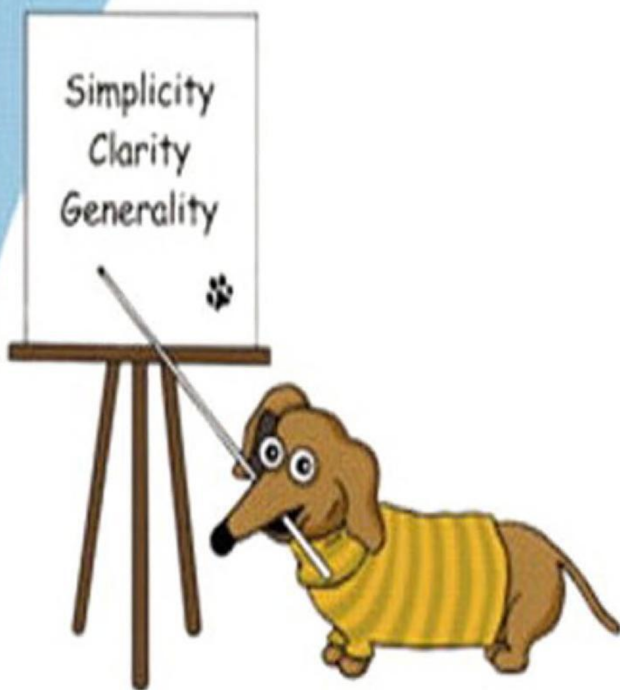
The Practice of Programming

Brian W. Kernighan
Rob Pike



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Practice of Programming

Brian W. Kernighan
Rob Pike

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Copyrighted material

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

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

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

Pearson Education Corporate Sales Division
201 W. 103rd Street
Indianapolis, IN 46290
(800) 428-5331
corpsales@pearsoned.com

Visit AW on the Web: www.awprofessional.com

This book was typeset (`grap|pic|tbl|eqn|troff - mpm`) in Times and Lucida Sans Typewriter by the authors.

Library of Congress Cataloging-in-Publication Data

Kernighan, Brian W.
The practice of programming / Brian W. Kernighan, Rob Pike.
p. cm. --(Addison-Wesley professional computing series)
Includes bibliographical references.
ISBN 0-201-61586-X
1. Computer programming. I. Pike, Rob. II. Title. III. Series.
QA76.6 .K48 1999
005.1--dc21 99-10131
CIP

Copyright © 1999 by Lucent Technologies.

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

ISBN 0-201-61586-X
Text printed in the United States on recycled paper at RR Donnelley in Harrisonburg, Virginia.
Twenty-second printing, February 2013

Copyrighted material

Contents

Preface ix

Chapter 1: Style 1
1.1 Names 3
1.2 Expressions and Statements 6
1.3 Consistency and Idioms 10
1.4 Function Macros 17
1.5 Magic Numbers 19
1.6 Comments 23
1.7 Why Bother? 27

Chapter 2: Algorithms and Data Structures 29
2.1 Searching 30
2.2 Sorting 32
2.3 Libraries 34
2.4 A Java Quicksort 37
2.5 O-Notation 40
2.6 Growing Arrays 41
2.7 Lists 44
2.8 Trees 50
2.9 Hash Tables 55
2.10 Summary 58

Chapter 3: Design and Implementation 61
3.1 The Markov Chain Algorithm 62
3.2 Data Structure Alternatives 64
3.3 Building the Data Structure in C 65
3.4 Generating Output 69

Copyrighted material

3.5 Java 71
3.6 C++ 76
3.7 Awk and Perl 78
3.8 Performance 80
3.9 Lessons 82

Chapter 4: Interfaces **85**
4.1 Comma-Separated Values 86
4.2 A Prototype Library 87
4.3 A Library for Others 91
4.4 A C++ Implementation 99
4.5 Interface Principles 103
4.6 Resource Management 106
4.7 Abort, Retry, Fail? 109
4.8 User Interfaces 113

Chapter 5: Debugging **117**
5.1 Debuggers 118
5.2 Good Clues, Easy Bugs 119
5.3 No Clues, Hard Bugs 123
5.4 Last Resorts 127
5.5 Non-reproducible Bugs 130
5.6 Debugging Tools 131
5.7 Other People's Bugs 135
5.8 Summary 136

Chapter 6: Testing **139**
6.1 Test as You Write the Code 140
6.2 Systematic Testing 145
6.3 Test Automation 149
6.4 Test Scaffolds 151
6.5 Stress Tests 155
6.6 Tips for Testing 158
6.7 Who Does the Testing? 159
6.8 Testing the Markov Program 160
6.9 Summary 162

Chapter 7: Performance **165**
7.1 A Bottleneck 166
7.2 Timing and Profiling 171
7.3 Strategies for Speed 175
7.4 Tuning the Code 178
7.5 Space Efficiency 182

Copyrighted material

7.6 Estimation 184
7.7 Summary 187

Chapter 8: Portability **189**
8.1 Language 190
8.2 Headers and Libraries 196
8.3 Program Organization 198
8.4 Isolation 202
8.5 Data Exchange 203
8.6 Byte Order 204
8.7 Portability and Upgrade 207
8.8 Internationalization 209
8.9 Summary 212

Chapter 9: Notation **215**
9.1 Formatting Data 216
9.2 Regular Expressions 222
9.3 Programmable Tools 228
9.4 Interpreters, Compilers, and Virtual Machines 231
9.5 Programs that Write Programs 237
9.6 Using Macros to Generate Code 240
9.7 Compiling on the Fly 241

Epilogue **247**

Appendix: Collected Rules **249**

Index **253**

Copyrighted material

This page intentionally left blank

Copyrighted material

Preface

Have you ever...

wasted a lot of time coding the wrong algorithm?
used a data structure that was much too complicated?
tested a program but missed an obvious problem?
spent a day looking for a bug you should have found in five minutes?
needed to make a program run three times faster and use less memory?
struggled to move a program from a workstation to a PC or vice versa?
tried to make a modest change in someone else's program?
rewritten a program because you couldn't understand it?

Was it fun?

These things happen to programmers all the time. But dealing with such problems is often harder than it should be because topics like testing, debugging, portability, performance, design alternatives, and style—the *practice* of programming—are not usually the focus of computer science or programming courses. Most programmers learn them haphazardly as their experience grows, and a few never learn them at all.

In a world of enormous and intricate interfaces, constantly changing tools and languages and systems, and relentless pressure for more of everything, one can lose sight of the basic principles—simplicity, clarity, generality—that form the bedrock of good software. One can also overlook the value of tools and notations that mechanize some of software creation and thus enlist the computer in its own programming.

Our approach in this book is based on these underlying, interrelated principles, which apply at all levels of computing. These include *simplicity*, which keeps programs short and manageable; *clarity*, which makes sure they are easy to understand, for people as well as machines; *generality*, which means they work well in a broad range of situations and adapt well as new situations arise; and *automation*, which lets the machine do the work for us, freeing us from mundane tasks. By looking at computer programming in a variety of languages, from algorithms and data structures through design, debugging, testing, and performance improvement, we can illustrate

Copyrighted material

universal engineering concepts that are independent of language, operating system, or programming paradigm.

This book comes from many years of experience writing and maintaining a lot of software, teaching programming courses, and working with a wide variety of programmers. We want to share lessons about practical issues, to pass on insights from our experience, and to suggest ways for programmers of all levels to be more proficient and productive.

We are writing for several kinds of readers. If you are a student who has taken a programming course or two and would like to be a better programmer, this book will expand on some of the topics for which there wasn't enough time in school. If you write programs as part of your work, but in support of other activities rather than as the goal in itself, the information will help you to program more effectively. If you are a professional programmer who didn't get enough exposure to such topics in school or who would like a refresher, or if you are a software manager who wants to guide your staff in the right direction, the material here should be of value.

We hope that the advice will help you to write better programs. The only prerequisite is that you have done some programming, preferably in C, C++ or Java. Of course the more experience you have, the easier it will be; nothing can take you from neophyte to expert in 21 days. Unix and Linux programmers will find some of the examples more familiar than will those who have used only Windows and Macintosh systems, but programmers from any environment should discover things to make their lives easier.

The presentation is organized into nine chapters, each focusing on one major aspect of programming practice.

Chapter 1 discusses programming style. Good style is so important to good programming that we have chosen to cover it first. Well-written programs are better than badly-written ones—they have fewer errors and are easier to debug and to modify—so it is important to think about style from the beginning. This chapter also introduces an important theme in good programming, the use of idioms appropriate to the language being used.

Algorithms and data structures, the topics of Chapter 2, are the core of the computer science curriculum and a major part of programming courses. Since most readers will already be familiar with this material, our treatment is intended as a brief review of the handful of algorithms and data structures that show up in almost every program. More complex algorithms and data structures usually evolve from these building blocks, so one should master the basics.

Chapter 3 describes the design and implementation of a small program that illustrates algorithm and data structure issues in a realistic setting. The program is implemented in five languages; comparing the versions shows how the same data structures are handled in each, and how expressiveness and performance vary across a spectrum of languages.

Copyrighted material

1

Style

It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules.

William Strunk and E. B. White, *The Elements of Style*

This fragment of code comes from a large program written many years ago:

```
if ( (country == SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY) )
{
        /*
         * If the country is Singapore, Brunei or Poland
         * then the current time is the answer time
         * rather than the off hook time.
         * Reset answer time and set day of week.
         */
        ...
```

It's carefully written, formatted, and commented, and the program it comes from works extremely well; the programmers who created this system are rightly proud of what they built. But this excerpt is puzzling to the casual reader. What relationship links Singapore, Brunei, Poland and Italy? Why isn't Italy mentioned in the comment? Since the comment and the code differ, one of them must be wrong. Maybe both are. The code is what gets executed and tested, so it's more likely to be right; probably the comment didn't get updated when the code did. The comment doesn't say enough about the relationship among the three countries it does mention; if you had to maintain this code, you would need to know more.

The few lines above are typical of much real code: mostly well done, but with some things that could be improved.

Copyrighted material

This book is about the practice of programming—how to write programs for real. Our purpose is to help you to write software that works at least as well as the program this example was taken from, while avoiding trouble spots and weaknesses. We will talk about writing better code from the beginning and improving it as it evolves.

We are going to start in an unusual place, however, by discussing programming style. The purpose of style is to make the code easy to read for yourself and others, and good style is crucial to good programming. We want to talk about it first so you will be sensitive to it as you read the code in the rest of the book.

There is more to writing a program than getting the syntax right, fixing the bugs, and making it run fast enough. Programs are read not only by computers but also by programmers. A well-written program is easier to understand and to modify than a poorly-written one. The discipline of writing well leads to code that is more likely to be correct. Fortunately, this discipline is not hard.

The principles of programming style are based on common sense guided by experience, not on arbitrary rules and prescriptions. Code should be clear and simple—straightforward logic, natural expression, conventional language use, meaningful names, neat formatting, helpful comments—and it should avoid clever tricks and unusual constructions. Consistency is important because others will find it easier to read your code, and you theirs, if you all stick to the same style. Details may be imposed by local conventions, management edict, or a program, but even if not, it is best to obey a set of widely shared conventions. We follow the style used in the book *The C Programming Language*, with minor adjustments for C++ and Java.

We will often illustrate rules of style by small examples of bad and good programming, since the contrast between two ways of saying the same thing is instructive. These examples are not artificial. The "bad" ones are all adapted from real code, written by ordinary programmers (occasionally ourselves) working under the common pressures of too much work and too little time. Some will be distilled for brevity, but they will not be misrepresented. Then we will rewrite the bad excerpts to show how they could be improved. Since they are real code, however, they may exhibit multiple problems. Addressing every shortcoming would take us too far off topic, so some of the good examples will still harbor other, unremarked flaws.

To distinguish bad examples from good, throughout the book we will place question marks in the margins of questionable code, as in this real excerpt:

```
?   #define ONE 1
?   #define TEN 10
?   #define TWENTY 20
```

Why are these `#defines` questionable? Consider the modifications that will be necessary if an array of `TWENTY` elements must be made larger. At the very least, each name should be replaced by one that indicates the role of the specific value in the program:

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

Copyrighted material

1.1 Names

What's in a name? A variable or function name labels an object and conveys information about its purpose. A name should be informative, concise, memorable, and pronounceable if possible. Much information comes from context and scope; the broader the scope of a variable, the more information should be conveyed by its name.

Use descriptive names for globals, short names for locals. Global variables, by definition, can crop up anywhere in a program, so they need names long enough and descriptive enough to remind the reader of their meaning. It's also helpful to include a brief comment with the declaration of each global:

```
int npending = 0;  // current length of input queue
```

Global functions, classes, and structures should also have descriptive names that suggest their role in a program.

By contrast, shorter names suffice for local variables; within a function, `n` may be sufficient, `npoints` is fine, and `numberOfPoints` is overkill.

Local variables used in conventional ways can have very short names. The use of `i` and `j` for loop indices, `p` and `q` for pointers, and `s` and `t` for strings is so frequent that there is little profit and perhaps some loss in longer names. Compare

```
?   for (theElementIndex = 0; theElementIndex < numberOfElements;
?           theElementIndex++)
?       elementArray[theElementIndex] = theElementIndex;
```

to

```
for (i = 0; i < nelems; i++)
    elem[i] = i;
```

Programmers are often encouraged to use long variable names regardless of context. That is a mistake: clarity is often achieved through brevity.

There are many naming conventions and local customs. Common ones include using names that begin or end with `p`, such as `nodep`, for pointers; initial capital letters for `Globals`; and all capitals for `CONSTANTS`. Some programming shops use more sweeping rules, such as notation to encode type and usage information in the variable, perhaps `pch` to mean a pointer to a character and `strTo` and `strFrom` to mean strings that will be written to and read from. As for the spelling of the names themselves, whether to use `npending` or `numPending` or `num_pending` is a matter of taste; specific rules are much less important than consistent adherence to a sensible convention.

Naming conventions make it easier to understand your own code, as well as code written by others. They also make it easier to invent new names as the code is being written. The longer the program, the more important is the choice of good, descriptive, systematic names.

Namespaces in C++ and packages in Java provide ways to manage the scope of names and help to keep meanings clear without unduly long names.

Copyrighted material

Be consistent. Give related things related names that show their relationship and highlight their difference.

Besides being much too long, the member names in this Java class are wildly inconsistent:

```
?    class UserQueue {
?        int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?        public int noOfUsersInQueue() {...}
?    }
```

The word ''queue'' appears as `Q`, `Queue` and `queue`. But since queues can only be accessed from a variable of type `UserQueue`, member names do not need to mention ''queue'' at all; context suffices, so

```
?    queue.queueCapacity
```

is redundant. This version is better:

```
class UserQueue {
    int nitems, front, capacity;
    public int nusers() {...}
}
```

since it leads to statements like

```
queue.capacity++;
n = queue.nusers();
```

No clarity is lost. This example still needs work, however: ''items'' and ''users'' are the same thing, so only one term should be used for a single concept.

Use active names for functions. Function names should be based on active verbs, perhaps followed by nouns:

```
now = date.getTime();
putchar('\n');
```

Functions that return a boolean (true or false) value should be named so that the return value is unambiguous. Thus

```
?    if (checkoctal(c)) ...
```

does not indicate which value is true and which is false, while

```
if (isoctal(c)) ...
```

makes it clear that the function returns true if the argument is octal and false if not.

Be accurate. A name not only labels, it conveys information to the reader. A misleading name can result in mystifying bugs.

One of us wrote and distributed for years a macro called `isoctal` with this incorrect implementation:

Copyrighted material

```
?    #define isoctal(c) ((c) >= '0' && (c) <= '8')
```

instead of the proper

```
     #define isoctal(c) ((c) >= '0' && (c) <= '7')
```

In this case, the name conveyed the correct intent but the implementation was wrong; it's easy for a sensible name to disguise a broken implementation.

Here's an example in which the name and the code are in complete contradiction:

```
?    public boolean inTable(Object obj) {
?        int j = this.getIndex(obj);
?        return (j == nTable);
?    }
```

The function `getIndex` returns a value between zero and `nTable-1` if it finds the object, and returns `nTable` if not. The boolean value returned by `inTable` is thus the opposite of what the name implies. At the time the code is written, this might not cause trouble, but if the program is modified later, perhaps by a different programmer, the name is sure to confuse.

Exercise 1-1. Comment on the choice of names and values in the following code.

```
?    #define TRUE 0
?    #define FALSE 1
?
?    if ((ch = getchar()) == EOF)
?        not_eof = FALSE;
```

□

Exercise 1-2. Improve this function:

```
?    int smaller(char *s, char *t) {
?        if (strcmp(s, t) < 1)
?            return 1;
?        else
?            return 0;
?    }
```

□

Exercise 1-3. Read this code aloud:

```
?    if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?        ...
```

□

Copyrighted material

What does this intricate calculation do?

```
?   subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

The innermost expression shifts `bitoff` three bits to the right. The result is shifted left again, thus replacing the three shifted bits by zeros. This result in turn is subtracted from the original value, yielding the bottom three bits of `bitoff`. These three bits are used to shift `subkey` to the right.

Thus the original expression is equivalent to

```
subkey = subkey >> (bitoff & 0x7);
```

It takes a while to puzzle out what the first version is doing; the second is shorter and clearer. Experienced programmers make it even shorter by using an assignment operator:

```
subkey >>= bitoff & 0x7;
```

Some constructs seem to invite abuse. The `?:` operator can lead to mysterious code:

```
?   child=(!LC&&!RC)?0:(!LC?RC:LC);
```

It's almost impossible to figure out what this does without following all the possible paths through the expression. This form is longer, but much easier to follow because it makes the paths explicit:

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

The `?:` operator is fine for short expressions where it can replace four lines of if-else with one, as in

```
max = (a > b) ? a : b;
```

or perhaps

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```

but it is not a general replacement for conditional statements.

Clarity is not the same as brevity. Often the clearer code will be shorter, as in the bit-shifting example, but it can also be longer, as in the conditional expression recast as an if-else. The proper criterion is ease of understanding.

Be careful with side effects. Operators like ++ have *side effects*: besides returning a value, they also modify an underlying variable. Side effects can be extremely convenient, but they can also cause trouble because the actions of retrieving the value and updating the variable might not happen at the same time. In C and C++, the order of

Copyrighted material

execution of side effects is undefined, so this multiple assignment is likely to produce the wrong answer:

```
?       str[i++] = str[i++] = ' ';
```

The intent is to store blanks at the next two positions in `str`. But depending on when `i` is updated, a position in `str` could be skipped and `i` might end up increased only by 1. Break it into two statements:

```
        str[i++] = ' ';
        str[i++] = ' ';
```

Even though it contains only one increment, this assignment can also give varying results:

```
?       array[i++] = i;
```

If `i` is initially 3, the array element might be set to 3 or 4.

It's not just increments and decrements that have side effects; I/O is another source of behind-the-scenes action. This example is an attempt to read two related numbers from standard input:

```
?       scanf("%d %d", &yr, &profit[yr]);
```

It is broken because part of the expression modifies `yr` and another part uses it. The value of `profit[yr]` can never be right unless the new value of `yr` is the same as the old one. You might think that the answer depends on the order in which the arguments are evaluated, but the real issue is that *all* the arguments to `scanf` are evaluated before the routine is called, so `&profit[yr]` will always be evaluated using the old value of `yr`. This sort of problem can occur in almost any language. The fix is, as usual, to break up the expression:

```
        scanf("%d", &yr);
        scanf("%d", &profit[yr]);
```

Exercise caution in any expression with side effects.

Exercise 1-4. Improve each of these fragments:

```
?       if ( !(c == 'y' || c == 'Y') )
?           return;
```

```
?       length = (length < BUFSIZE) ? length : BUFSIZE;
```

```
?       flag = flag ? 0 : 1;
```

```
?       quote = (*line == '"') ? 1 : 0;
```

Copyrighted material

```
?     if (val & 1)
?         bit = 1;
?     else
?         bit = 0;
```

□

Exercise 1-5. What is wrong with this excerpt?

```
?     int read(int *ip) {
?         scanf("%d", ip);
?         return *ip;
?     }
?         ...
?     insert(&graph[vert], read(&val), read(&ch));
```

□

Exercise 1-6. List all the different outputs this could produce with various orders of evaluation:

```
?     n = 1;
?     printf("%d %d\n", n++, n++);
```

Try it on as many compilers as you can, to see what happens in practice. □

1.3 Consistency and Idioms

Consistency leads to better programs. If formatting varies unpredictably, or a loop over an array runs uphill this time and downhill the next, or strings are copied with `strcpy` here and a `for` loop there, the variations make it harder to see what's really going on. But if the same computation is done the same way every time it appears, any variation suggests a genuine difference, one worth noting.

Use a consistent indentation and brace style. Indentation shows structure, but which indentation style is best? Should the opening brace go on the same line as the `if` or on the next? Programmers have always argued about the layout of programs, but the specific style is much less important than its consistent application. Pick one style, preferably ours, use it consistently, and don't waste time arguing.

Should you include braces even when they are not needed? Like parentheses, braces can resolve ambiguity and occasionally make the code clearer. For consistency, many experienced programmers always put braces around loop or `if` bodies. But if the body is a single statement they are unnecessary, so we tend to omit them. If you also choose to leave them out, make sure you don't drop them when they are needed to resolve the ''dangling else'' ambiguity exemplified by this excerpt:

Copyrighted material

```
?     if (month == FEB) {
?         if (year%4 == 0)
?             if (day > 29)
?                 legal = FALSE;
?         else
?             if (day > 28)
?                 legal = FALSE;
?     }
```

The indentation is misleading, since the `else` is actually attached to the line

```
?             if (day > 29)
```

and the code is wrong. When one `if` immediately follows another, always use braces:

```
?     if (month == FEB) {
?         if (year%4 == 0) {
?             if (day > 29)
?                 legal = FALSE;
?         } else {
?             if (day > 28)
?                 legal = FALSE;
?         }
?     }
```

Syntax-driven editing tools make this sort of mistake less likely.

Even with the bug fixed, though, the code is hard to follow. The computation is easier to grasp if we use a variable to hold the number of days in February:

```
?     if (month == FEB) {
?         int nday;
?
?         nday = 28;
?         if (year%4 == 0)
?             nday = 29;
?         if (day > nday)
?             legal = FALSE;
?     }
```

The code is still wrong—2000 is a leap year, while 1900 and 2100 are not—but this structure is much easier to adapt to make it absolutely right.

By the way, if you work on a program you didn't write, preserve the style you find there. When you make a change, don't use your own style even though you prefer it. The program's consistency is more important than your own, because it makes life easier for those who follow.

Use idioms for consistency. Like natural languages, programming languages have idioms, conventional ways that experienced programmers write common pieces of code. A central part of learning any language is developing a familiarity with its idioms.

Copyrighted material

One of the most common idioms is the form of a loop. Consider the C, C++, or Java code for stepping through the `n` elements of an array, for example to initialize them. Someone might write the loop like this:

```
?       i = 0;
?       while (i <= n-1)
?           array[i++] = 1.0;
```

or perhaps like this:

```
?       for (i = 0; i < n; )
?           array[i++] = 1.0;
```

or even:

```
?       for (i = n; --i >= 0; )
?           array[i] = 1.0;
```

All of these are correct, but the idiomatic form is like this:

```
        for (i = 0; i < n; i++)
            array[i] = 1.0;
```

This is not an arbitrary choice. It visits each member of an `n`-element array indexed from `0` to `n-1`. It places all the loop control in the `for` itself, runs in increasing order, and uses the very idiomatic `++` operator to update the loop variable. It leaves the index variable at a known value just beyond the last array element. Native speakers recognize it without study and write it correctly without a moment's thought.

In C++ or Java, a common variant includes the declaration of the loop variable:

```
        for (int i = 0; i < n; i++)
            array[i] = 1.0;
```

Here is the standard loop for walking along a list in C:

```
        for (p = list; p != NULL; p = p->next)
            ...
```

Again, all the loop control is in the `for`.

For an infinite loop, we prefer

```
        for (;;)
            ...
```

but

```
        while (1)
            ...
```

is also popular. Don't use anything other than these forms.

Indentation should be idiomatic, too. This unusual vertical layout detracts from readability; it looks like three statements, not a loop:

Copyrighted material

The last `else` handles the ''default'' situation, where none of the other alternatives was chosen. This trailing `else` part may be omitted if there is no action for the default, although leaving it in with an error message may help to catch conditions that ''can't happen.''

Align all of the `else` clauses vertically rather than lining up each `else` with the corresponding `if`. Vertical alignment emphasizes that the tests are done in sequence and keeps them from marching off the right side of the page.

A sequence of nested `if` statements is often a warning of awkward code, if not outright errors:

```
?   if (argc == 3)
?       if ((fin = fopen(argv[1], "r")) != NULL)
?           if ((fout = fopen(argv[2], "w")) != NULL) {
?               while ((c = getc(fin)) != EOF)
?                   putc(c, fout);
?               fclose(fin); fclose(fout);
?           } else
?               printf("Can't open output file %s\n", argv[2]);
?       else
?           printf("Can't open input file %s\n", argv[1]);
?   else
?       printf("Usage: cp inputfile outputfile\n");
```

The sequence of `if`s requires us to maintain a mental pushdown stack of what tests were made, so that at the appropriate point we can pop them until we determine the corresponding action (if we can still remember). Since at most one action will be performed, we really want an `else if`. Changing the order in which the decisions are made leads to a clearer version, in which we have also corrected the resource leak in the original:

```
    if (argc != 3)
        printf("Usage: cp inputfile outputfile\n");
    else if ((fin = fopen(argv[1], "r")) == NULL)
        printf("Can't open input file %s\n", argv[1]);
    else if ((fout = fopen(argv[2], "w")) == NULL) {
        printf("Can't open output file %s\n", argv[2]);
        fclose(fin);
    } else {
        while ((c = getc(fin)) != EOF)
            putc(c, fout);
        fclose(fin);
        fclose(fout);
    }
```

We read down the tests until the first one that is true, do the corresponding action, and continue after the last `else`. The rule is to follow each decision as closely as possible by its associated action. Or, to put it another way, each time you make a test, do something.

Attempts to re-use pieces of code often lead to tightly knotted programs:

Copyrighted material

```
?    switch (c) {
?    case '-':  sign = -1;
?    case '+':  c = getchar();
?    case '.':  break;
?    default:   if (!isdigit(c))
?                      return 0;
?    }
```

This uses a tricky sequence of fall-throughs in the switch statement to avoid duplicating one line of code. It's also not idiomatic; cases should almost always end with a `break`, with the rare exceptions commented. A more traditional layout and structure is easier to read, though longer:

```
?    switch (c) {
?    case '-':
?        sign = -1;
?        /* fall through */
?    case '+':
?        c = getchar();
?        break;
?    case '.':
?        break;
?    default:
?        if (!isdigit(c))
?            return 0;
?        break;
?    }
```

The increase in size is more than offset by the increase in clarity. However, for such an unusual structure a sequence of else-if statements is even clearer:

```
    if (c == '-') {
        sign = -1;
        c = getchar();
    } else if (c == '+') {
        c = getchar();
    } else if (c != '.' && !isdigit(c)) {
        return 0;
    }
```

The braces around the one-line blocks highlight the parallel structure.

An acceptable use of a fall-through occurs when several cases have identical code; the conventional layout is like this:

```
    case '0':
    case '1':
    case '2':
        ...
        break;
```

and no comment is required.

Copyrighted material

Exercise 1-7. Rewrite these C/C++ excerpts more clearly:

```
?     if (istty(stdin))  ;
?     else if (istty(stdout))  ;
?         else if (istty(stderr))  ;
?             else return(0);
```

```
?     if (retval != SUCCESS)
?     {
?         return (retval);
?     }
?     /* All went well! */
?     return SUCCESS;
```

```
?     for (k = 0; k++ < 5; x += dx)
?         scanf("%lf", &dx);
```

□

Exercise 1-8. Identify the errors in this Java fragment and repair it by rewriting with an idiomatic loop:

```
?     int count = 0;
?     while (count < total) {
?         count++;
?         if (this.getName(count) == nametable.userName()) {
?             return (true);
?         }
?     }
```

□

1.4 Function Macros

There is a tendency among older C programmers to write macros instead of functions for very short computations that will be executed frequently; I/O operations such as `getchar` and character tests like `isdigit` are officially sanctioned examples. The reason is performance: a macro avoids the overhead of a function call. This argument was weak even when C was first defined, a time of slow machines and expensive function calls; today it is irrelevant. With modern machines and compilers, the drawbacks of function macros outweigh their benefits.

Avoid function macros. In C++, inline functions render function macros unnecessary; in Java, there are no macros. In C, they cause more problems than they solve.

Copyrighted material

One of the most serious problems with function macros is that a parameter that appears more than once in the definition might be evaluated more than once; if the argument in the call includes an expression with side effects, the result is a subtle bug. This code attempts to implement one of the character tests from `<ctype.h>`:

```
?   #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

Note that the parameter `c` occurs twice in the body of the macro. If `isupper` is called in a context like this,

```
?   while (isupper(c = getchar()))
?       ...
```

then each time an input character is greater than or equal to `A`, it will be discarded and another character read to be tested against `Z`. The C standard is carefully written to permit `isupper` and analogous functions to be macros, but only if they guarantee to evaluate the argument only once, so this implementation is broken.

It's always better to use the `ctype` functions than to implement them yourself, and it's safer not to nest routines like `getchar` that have side effects. Rewriting the test to use two expressions rather than one makes it clearer and also gives an opportunity to catch end-of-file explicitly:

```
    while ((c = getchar()) != EOF && isupper(c))
        ...
```

Sometimes multiple evaluation causes a performance problem rather than an outright error. Consider this example:

```
?   #define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?0.5:-0.5)))
?       ...
?   size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

This will perform the square root computation twice as often as necessary. Even given simple arguments, a complex expression like the body of `ROUND_TO_INT` translates into many instructions, which should be housed in a single function to be called when needed. Instantiating a macro at every occurrence makes the compiled program larger. (C++ inline functions have this drawback, too.)

Parenthesize the macro body and arguments. If you insist on using function macros, be careful. Macros work by textual substitution: the parameters in the definition are replaced by the arguments of the call and the result replaces the original call, as text. This is a troublesome difference from functions. The expression

```
    1 / square(x)
```

works fine if `square` is a function, but if it's a macro like this,

```
?   #define square(x)   (x) * (x)
```

the expression will be expanded to the erroneous

Copyrighted material

```
?     1 / (x) * (x)
```

The macro should be rewritten as

```
#define square(x)   ((x) * (x))
```

All those parentheses are necessary. Even parenthesizing the macro properly does not address the multiple evaluation problem. If an operation is expensive or common enough to be wrapped up, use a function.

In C++, inline functions avoid the syntactic trouble while offering whatever performance advantage macros might provide. They are appropriate for short functions that set or retrieve a single value.

Exercise 1-9. Identify the problems with this macro definition:

```
?     #define ISDIGIT(c) ((c >= '0') && (c <= '9')) ? 1 : 0
```

□

1.5 Magic Numbers

Magic numbers are the constants, array sizes, character positions, conversion factors, and other literal numeric values that appear in programs.

Give names to magic numbers. As a guideline, any number other than 0 or 1 is likely to be magic and should have a name of its own. A raw number in program source gives no indication of its importance or derivation, making the program harder to understand and modify. This excerpt from a program to print a histogram of letter frequencies on a 24 by 80 cursor-addressed terminal is needlessly opaque because of a host of magic numbers:

```
?         fac = lim / 20;      /* set scale factor */
?         if (fac < 1)
?             fac = 1;
?                               /* generate histogram */
?         for (i = 0, col = 0; i < 27; i++, j++) {
?             col += 3;
?             k = 21 - (let[i] / fac);
?             star = (let[i] == 0) ? ' ' : '*';
?             for (j = k; j < 22; j++)
?                 draw(j, col, star);
?         }
?         draw(23, 2, ' '); /* label x axis */
?         for (i = 'A'; i <= 'Z'; i++)
?             printf("%c  ", i);
```

Copyrighted material

Use the language to calculate the size of an object. Don't use an explicit size for any data type; use `sizeof(int)` instead of 2 or 4, for instance. For similar reasons, `sizeof(array[0])` may be better than `sizeof(int)` because it's one less thing to change if the type of the array changes.

The `sizeof` operator is sometimes a convenient way to avoid inventing names for the numbers that determine array sizes. For example, if we write

```
char buf[1024];

fgets(buf, sizeof(buf), stdin);
```

the buffer size is still a magic number, but it occurs only once, in the declaration. It may not be worth inventing a name for the size of a local array, but it is definitely worth writing code that does not have to change if the size or type changes.

Java arrays have a `length` field that gives the number of elements:

```
char buf[] = new char[1024];

for (int i = 0; i < buf.length; i++)
    ...
```

There is no equivalent of `.length` in C and C++, but for an array (not a pointer) whose declaration is visible, this macro computes the number of elements in the array:

```
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))

double dbuf[100];

for (i = 0; i < NELEMS(dbuf); i++)
    ...
```

The array size is set in only one place; the rest of the code does not change if the size does. There is no problem with multiple evaluation of the macro argument here, since there can be no side effects, and in fact the computation is done as the program is being compiled. This is an appropriate use for a macro because it does something that a function cannot: compute the size of an array from its declaration.

Exercise 1-10. How would you rewrite these definitions to minimize potential errors?

```
?   #define FT2METER    0.3048
?   #define METER2FT    3.28084
?   #define MI2FT       5280.0
?   #define MI2KM       1.609344
?   #define SQMI2SQKM   2.589988
```

□

Copyrighted material

1.6 Comments

Comments are meant to help the reader of a program. They do not help by saying things the code already plainly says, or by contradicting the code, or by distracting the reader with elaborate typographical displays. The best comments aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings.

Don't belabor the obvious. Comments shouldn't report self-evident information, such as the fact that `i++` has incremented `i`. Here are some of our favorite worthless comments:

```
?	/*
?	 * default
?	 */
?	default:
?	    break;

?	/* return SUCCESS */
?	return SUCCESS;

?	zerocount++;    /* Increment zero entry counter */

?	/* Initialize "total" to "number_received" */
?	node->total = node->number_received;
```

All of these comments should be deleted; they're just clutter.

Comments should add something that is not immediately evident from the code, or collect into one place information that is spread through the source. When something subtle is happening, a comment may clarify, but if the actions are obvious already, restating them in words is pointless:

```
?	while ((c = getchar()) != EOF && isspace(c))
?	    ;                           /* skip white space */
?	if (c == EOF)                   /* end of file */
?	    type = endoffile;
?	else if (c == '(')              /* left paren */
?	    type = leftparen;
?	else if (c == ')')              /* right paren */
?	    type = rightparen;
?	else if (c == ';')              /* semicolon */
?	    type = semicolon;
?	else if (is_op(c))              /* operator */
?	    type = operator;
?	else if (isdigit(c))            /* number */
?	    ...
```

These comments should also be deleted, since the well-chosen names already convey the information.

Copyrighted material

Comment functions and global data. Comments *can* be useful, of course. We comment functions, global variables, constant definitions, fields in structures and classes, and anything else where a brief summary can aid understanding.

Global variables have a tendency to crop up intermittently throughout a program; a comment serves as a reminder to be referred to as needed. Here's an example from a program in Chapter 3 of this book:

```
struct State {  /* prefix + suffix list */
    char    *pref[NPREF];   /* prefix words */
    Suffix  *suf;           /* list of suffixes */
    State   *next;          /* next in hash table */
};
```

A comment that introduces each function sets the stage for reading the code itself. If the code isn't too long or technical, a single line is enough:

```
// random: return an integer in the range [0..r-1].
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

Sometimes code is genuinely difficult, perhaps because the algorithm is complicated or the data structures are intricate. In that case, a comment that points to a source of understanding can aid the reader. It may also be valuable to suggest why particular decisions were made. This comment introduces an extremely efficient implementation of an inverse discrete cosine transform (DCT) used in a JPEG image decoder.

```
/*
 * idct: Scaled integer implementation of
 * Inverse two dimensional 8x8 Discrete Cosine Transform,
 * Chen-Wang algorithm (IEEE ASSP-32, pp 803-816, Aug 1984)
 *
 * 32-bit integer arithmetic (8-bit coefficients)
 * 11 multiplies, 29 adds per DCT
 *
 * Coefficients extended to 12 bits for
 * IEEE 1180-1990 compliance
 */

static void idct(int b[8*8])
{
    ...
}
```

This helpful comment cites the reference, briefly describes the data used, indicates the performance of the algorithm, and tells how and why the original algorithm has been modified.

Copyrighted material

Don't comment bad code, rewrite it. Comment anything unusual or potentially confusing, but when the comment outweighs the code, the code probably needs fixing. This example uses a long, muddled comment and a conditionally-compiled debugging print statement to explain a single statement:

```
?     /* If "result" is 0 a match was found so return true (non-zero).
?        Otherwise, "result" is non-zero so return false (zero). */
?
?     #ifdef DEBUG
?     printf("*** isword returns !result = %d\n", !result);
?     fflush(stdout);
?     #endif
?
?     return(!result);
```

Negations are hard to understand and should be avoided. Part of the problem is the uninformative variable name, `result`. A more descriptive name, `matchfound`, makes the comment unnecessary and cleans up the print statement, too.

```
      #ifdef DEBUG
      printf("*** isword returns matchfound = %d\n", matchfound);
      fflush(stdout);
      #endif

      return matchfound;
```

Don't contradict the code. Most comments agree with the code when they are written, but as bugs are fixed and the program evolves, the comments are often left in their original form, resulting in disagreement with the code. This is the likely explanation for the inconsistency in the example that opens this chapter.

Whatever the source of the disagreement, a comment that contradicts the code is confusing, and many a debugging session has been needlessly protracted because a mistaken comment was taken as truth. When you change code, make sure the comments are still accurate.

Comments should not only agree with code, they should support it. The comment in this example is correct—it explains the purpose of the next two lines—but it appears to contradict the code; the comment talks about newline and the code talks about blanks:

```
?     time(&now);
?     strcpy(date, ctime(&now));
?     /* get rid of trailing newline character copied from ctime */
?     i = 0;
?     while(date[i] >= ' ') i++;
?     date[i] = 0;
```

One improvement is to rewrite the code more idiomatically:

Copyrighted material

```
?       time(&now);
?       strcpy(date, ctime(&now));
?       /* get rid of trailing newline character copied from ctime */
?       for (i = 0; date[i] != '\n'; i++)
?               ;
?       date[i] = '\0';
```

Code and comment now agree, but both can be improved by being made more direct. The problem is to delete the newline that `ctime` puts on the end of the string it returns. The comment should say so, and the code should do so:

```
        time(&now);
        strcpy(date, ctime(&now));
        /* ctime() puts newline at end of string; delete it */
        date[strlen(date)-1] = '\0';
```

This last expression is the C idiom for removing the last character from a string. The code is now short, idiomatic, and clear, and the comment supports it by explaining why it needs to be there.

Clarify, don't confuse. Comments are supposed to help readers over the hard parts, not create more obstacles. This example follows our guidelines of commenting the function and explaining unusual properties; on the other hand, the function is `strcmp` and the unusual properties are peripheral to the job at hand, which is the implementation of a standard and familiar interface:

```
?       int strcmp(char *s1, char *s2)
?       /* string comparison routine returns -1 if s1 is */
?       /* above s2 in an ascending order list, 0 if equal */
?       /* 1 if s1 below s2 */
?       {
?           while(*s1==*s2)  {
?               if(*s1=='\0')  return(0);
?               s1++;
?               s2++;
?           }
?           if(*s1>*s2) return(1);
?           return(-1);
?       }
```

When it takes more than a few words to explain what's happening, it's often an indication that the code should be rewritten. Here, the code could perhaps be improved but the real problem is the comment, which is nearly as long as the implementation and confusing, too (which way is "above"?). We're stretching the point to say this routine is hard to understand, but since it implements a standard function, its comment can help by summarizing the behavior and telling us where the definition originates; that's all that's needed:

Copyrighted material

2

Algorithms and Data Structures

In the end, only familiarity with the tools and techniques of the field will provide the right solution for a particular problem, and only a certain amount of experience will provide consistently professional results.

Raymond Fielding, *The Technique of Special Effects Cinematography*

The study of algorithms and data structures is one of the foundations of computer science, a rich field of elegant techniques and sophisticated mathematical analyses. And it's more than just fun and games for the theoretically inclined: a good algorithm or data structure might make it possible to solve a problem in seconds that could otherwise take years.

In specialized areas like graphics, databases, parsing, numerical analysis, and simulation, the ability to solve problems depends critically on state-of-the-art algorithms and data structures. If you are developing programs in a field that's new to you, you *must* find out what is already known, lest you waste your time doing poorly what others have already done well.

Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones. Even within an intricate program like a compiler or a web browser, most of the data structures are arrays, lists, trees, and hash tables. When a program needs something more elaborate, it will likely be based on these simpler ones. Accordingly, for most programmers, the task is to know what appropriate algorithms and data structures are available and to understand how to choose among alternatives.

Here is the story in a nutshell. There are only a handful of basic algorithms that show up in almost every program—primarily searching and sorting—and even those are often included in libraries. Similarly, almost every data structure is derived from a few fundamental ones. Thus the material covered in this chapter will be familiar to almost all programmers. We have written working versions to make the discussion

Copyrighted material

concrete, and you can lift code verbatim if necessary, but do so only after you have investigated what the programming language and its libraries have to offer.

2.1 Searching

Nothing beats an array for storing static tabular data. Compile-time initialization makes it cheap and easy to construct such arrays. (In Java, the initialization occurs at run-time, but this is an unimportant implementation detail unless the arrays are large.) In a program to detect words that are used rather too much in bad prose, we can write

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

The search routine needs to know how many elements are in the array. One way to tell it is to pass the length as an argument; another, used here, is to place a `NULL` marker at the end of the array:

```
/* lookup: sequential search for word in array */
int lookup(char *word, char *array[])
{
    int i;

    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

In C and C++, a parameter that is an array of strings can be declared as `char *array[]` or `char **array`. Although these forms are equivalent, the first makes it clearer how the parameter will be used.

This search algorithm is called *sequential search* because it looks at each element in turn to see if it's the desired one. When the amount of data is small, sequential search is fast enough. There are standard library routines to do sequential search for specific data types; for example, functions like `strchr` and `strstr` search for the first instance of a given character or substring in a C or C++ string, the Java `String` class has an `indexOf` method, and the generic C++ `find` algorithms apply to most data types. If such a function exists for the data type you've got, use it.

Sequential search is easy but the amount of work is directly proportional to the amount of data to be searched; doubling the number of elements will double the time to search if the desired item is not present. This is a linear relationship—run-time is a linear function of data size—so this method is also known as *linear search*.

Copyrighted material

Here's an excerpt from an array of more realistic size from a program that parses HTML, which defines textual names for well over a hundred individual characters:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

/* HTML characters, e.g. AElig is ligature of A and E. */
/* Values are Unicode/ISO10646 encoding. */
Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta",     0x03b6,
};
```

For a larger array like this, it's more efficient to use *binary search.* The binary search algorithm is an orderly version of the way we look up words in a dictionary. Check the middle element. If that value is bigger than what we are looking for, look in the first half; otherwise, look in the second half. Repeat until the desired item is found or determined not to be present.

For binary search, the table must be sorted, as it is here (that's good style anyway; people find things faster in sorted tables too), and we must know how long the table is. The NELEMS macro from Chapter 1 can help:

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

A binary search function for this table might look like this:

```
/* lookup: binary search for name in tab; return index */
int lookup(char *name, Nameval tab[], int ntab)
{
    int low, high, mid, cmp;

    low = 0;
    high = ntab - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        cmp = strcmp(name, tab[mid].name);
        if (cmp < 0)
            high = mid - 1;
        else if (cmp > 0)
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;  /* no match */
}
```

Copyrighted material

Putting all this together, to search `htmlchars` we write

```
half = lookup("frac12", htmlchars, NELEMS(htmlchars));
```

to find the array index of the character ½.

Binary search eliminates half the data at each step. The number of steps is therefore proportional to the number of times we can divide n by 2 before we're left with a single element. Ignoring roundoff, this is $\log_2 n$. If we have 1000 items to search, linear search takes up to 1000 steps, while binary search takes about 10; if we have a million items, linear takes a million steps and binary takes 20. The more items, the greater the advantage of binary search. Beyond some size of input (which varies with the implementation), binary search is faster than linear search.

2.2 Sorting

Binary search works only if the elements are sorted. If repeated searches are going to be made in some data set, it will be profitable to sort once and then use binary search. If the data set is known in advance, it can be sorted when the program is written and built using compile-time initialization. If not, it must be sorted when the program is run.

One of the best all-round sorting algorithms is *quicksort*, which was invented in 1960 by C. A. R. Hoare. Quicksort is a fine example of how to avoid extra computing. It works by partitioning an array into little and big elements:

> pick one element of the array (the ''pivot'').
> partition the other elements into two groups:
>   ''little ones'' that are less than the pivot value, and
>   ''big ones'' that are greater than or equal to the pivot value.
> recursively sort each group.

When this process is finished, the array is in order. Quicksort is fast because once an element is known to be less than the pivot value, we don't have to compare it to any of the big ones; similarly, big ones are not compared to little ones. This makes it much faster than the simple sorting methods such as insertion sort and bubble sort that compare each element directly to all the others.

Quicksort is practical and efficient; it has been extensively studied and myriad variations exist. The version that we present here is just about the simplest implementation but it is certainly not the quickest.

This `quicksort` function sorts an array of integers:

Copyrighted material