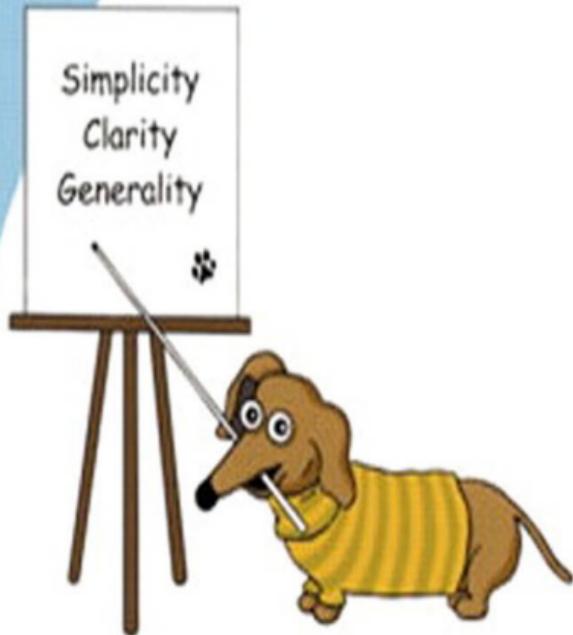




ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# The Practice of Programming

Brian W. Kernighan  
Rob Pike



FREE SAMPLE CHAPTER



SHARE WITH OTHERS

# The Practice of Programming

---

*This page intentionally left blank*

# 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

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](mailto:corpsales@pearsoned.com)

Visit AW on the Web: [www.awprofessional.com](http://www.awprofessional.com)

This book was typeset (graph|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

I. 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

---

---

# Contents

|  |           |
|--|-----------|
| <b>Preface</b>                                   | <b>ix</b> |
| <b>Chapter 1: Style</b>                          | <b>1</b>  |
| 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        |
| <b>Chapter 2: Algorithms and Data Structures</b> | <b>29</b> |
| 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        |
| <b>Chapter 3: Design and Implementation</b>      | <b>61</b> |
| 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        |

|                                |            |
|--------------------------------|------------|
| 3.5 Java                       | 71         |
| 3.6 C++                        | 76         |
| 3.7 Awk and Perl               | 78         |
| 3.8 Performance                | 80         |
| 3.9 Lessons                    | 82         |
| <b>Chapter 4: Interfaces</b>   | <b>85</b>  |
| 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        |
| <b>Chapter 5: Debugging</b>    | <b>117</b> |
| 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        |
| <b>Chapter 6: Testing</b>      | <b>139</b> |
| 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        |
| <b>Chapter 7: Performance</b>  | <b>165</b> |
| 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        |

|   |            |
|---|------------|
| 7.6 Estimation                                    | 184        |
| 7.7 Summary                                       | 187        |
| <b>Chapter 8: Portability</b>                     | <b>189</b> |
| 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        |
| <b>Chapter 9: Notation</b>                        | <b>215</b> |
| 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        |
| <b>Epilogue</b>                                   | <b>247</b> |
| <b>Appendix: Collected Rules</b>                  | <b>249</b> |
| <b>Index</b>                                      | <b>253</b> |

*This page intentionally left blank*

---

---

# 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

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.

Interfaces between users, programs, and parts of programs are fundamental in programming and much of the success of software is determined by how well interfaces are designed and implemented. Chapter 4 shows the evolution of a small library for parsing a widely used data format. Even though the example is small, it illustrates many of the concerns of interface design: abstraction, information hiding, resource management, and error handling.

Much as we try to write programs correctly the first time, bugs, and therefore debugging, are inevitable. Chapter 5 gives strategies and tactics for systematic and effective debugging. Among the topics are the signatures of common bugs and the importance of “numerology,” where patterns in debugging output often indicate where a problem lies.

Testing is an attempt to develop a reasonable assurance that a program is working correctly and that it stays correct as it evolves. The emphasis in Chapter 6 is on systematic testing by hand and machine. Boundary condition tests probe at potential weak spots. Mechanization and test scaffolds make it easy to do extensive testing with modest effort. Stress tests provide a different kind of testing than typical users do and ferret out a different class of bugs.

Computers are so fast and compilers are so good that many programs are fast enough the day they are written. But others are too slow, or they use too much memory, or both. Chapter 7 presents an orderly way to approach the task of making a program use resources efficiently, so that the program remains correct and sound as it is made more efficient.

Chapter 8 covers portability. Successful programs live long enough that their environment changes, or they must be moved to new systems or new hardware or new countries. The goal of portability is to reduce the maintenance of a program by minimizing the amount of change necessary to adapt it to a new environment.

Computing is rich in languages, not just the general-purpose ones that we use for the bulk of programming, but also many specialized languages that focus on narrow domains. Chapter 9 presents several examples of the importance of notation in computing, and shows how we can use it to simplify programs, to guide implementations, and even to help us write programs that write programs.

To talk about programming, we have to show a lot of code. Most of the examples were written expressly for the book, although some small ones were adapted from other sources. We’ve tried hard to write our own code well, and have tested it on half a dozen systems directly from the machine-readable text. More information is available at the web site for *The Practice of Programming*:

<http://tpop.awl.com>

The majority of the programs are in C, with a number of examples in C++ and Java and some brief excursions into scripting languages. At the lowest level, C and C++ are almost identical and our C programs are valid C++ programs as well. C++ and Java are lineal descendants of C, sharing more than a little of its syntax and much of its efficiency and expressiveness, while adding richer type systems and libraries.

In our own work, we routinely use all three of these languages, and many others. The choice of language depends on the problem: operating systems are best written in an efficient and unrestrictive language like C or C++; quick prototypes are often easiest in a command interpreter or a scripting language like Awk or Perl; for user interfaces, Visual Basic and Tcl/Tk are strong contenders, along with Java.

There is an important pedagogical issue in choosing a language for our examples. Just as no language solves all problems equally well, no single language is best for presenting all topics. Higher-level languages preempt some design decisions. If we use a lower-level language, we get to consider alternative answers to the questions; by exposing more of the details, we can talk about them better. Experience shows that even when we use the facilities of high-level languages, it's invaluable to know how they relate to lower-level issues; without that insight, it's easy to run into performance problems and mysterious behavior. So we will often use C for our examples, even though in practice we might choose something else.

For the most part, however, the lessons are independent of any particular programming language. The choice of data structure is affected by the language at hand; there may be few options in some languages while others might support a variety of alternatives. But the way to approach making the choice will be the same. The details of how to test and debug are different in different languages, but strategies and tactics are similar in all. Most of the techniques for making a program efficient can be applied in any language.

Whatever language you write in, your task as a programmer is to do the best you can with the tools at hand. A good programmer can overcome a poor language or a clumsy operating system, but even a great programming environment will not rescue a bad programmer. We hope that, no matter what your current experience and skill, this book will help you to program better and enjoy it more.

We are deeply grateful to friends and colleagues who read drafts of the manuscript and gave us many helpful comments. Jon Bentley, Russ Cox, John Lakos, John Linderman, Peter Memishian, Ian Lance Taylor, Howard Trickey, and Chris Van Wyk read the manuscript, some more than once, with exceptional care and thoroughness. We are indebted to Tom Cargill, Chris Cleeland, Steve Dewhurst, Eric Grosse, Andrew Herron, Gerard Holzmann, Doug McIlroy, Paul McNamee, Peter Nelson, Dennis Ritchie, Rich Stevens, Tom Szymanski, Kentaro Toyama, John Wait, Daniel C. Wang, Peter Weinberger, Margaret Wright, and Cliff Young for invaluable comments on drafts at various stages. We also appreciate good advice and thoughtful suggestions from Al Aho, Ken Arnold, Chuck Bigelow, Joshua Bloch, Bill Coughran, Bob Flandrena, Renée French, Mark Kernighan, Andy Koenig, Sape Mullender, Evi Nemeth, Marty Rabinowitz, Mark V. Shaney, Bjarne Stroustrup, Ken Thompson, and Phil Wadler. Thank you all.

*Brian W. Kernighan*

*Rob Pike*

*This page intentionally left blank*

# 3

---

## Design and Implementation

*Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*

Frederick P. Brooks, Jr., *The Mythical Man Month*

As the quotation from Brooks's classic book suggests, the design of the data structures is the central decision in the creation of a program. Once the data structures are laid out, the algorithms tend to fall into place, and the coding is comparatively easy.

This point of view is oversimplified but not misleading. In the previous chapter we examined the basic data structures that are the building blocks of most programs. In this chapter we will combine such structures as we work through the design and implementation of a modest-sized program. We will show how the problem influences the data structures, and how the code that follows is straightforward once we have the data structures mapped out.

One aspect of this point of view is that the choice of programming language is relatively unimportant to the overall design. We will design the program in the abstract and then write it in C, Java, C++, Awk, and Perl. Comparing the implementations demonstrates how languages can help or hinder, and ways in which they are unimportant. Program design can certainly be colored by a language but is not usually dominated by it.

The problem we have chosen is unusual, but in basic form it is typical of many programs: some data comes in, some data goes out, and the processing depends on a little ingenuity.

Specifically, we're going to generate random English text that reads well. If we emit random letters or random words, the result will be nonsense. For example, a program that randomly selects letters (and blanks, to separate words) might produce this:

```
xptmxgn xusaja afqnzgx1 lhidlwcd rjdjuvpydrlnwjy
```

which is not very convincing. If we weight the letters by their frequency of appearance in English text, we might get this:

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

which isn't a great deal better. Words chosen from the dictionary at random don't make much more sense:

```
polydactyl equatorial splashily jowl verandah circumscribe
```

For better results, we need a statistical model with more structure, such as the frequency of appearance of whole phrases. But where can we find such statistics?

We could grab a large body of English and study it in detail, but there is an easier and more entertaining approach. The key observation is that we can use any existing text to construct a statistical model of the language *as used in that text*, and from that generate random text that has similar statistics to the original.

## 3.1 The Markov Chain Algorithm

An elegant way to do this sort of processing is a technique called a *Markov chain algorithm*. If we imagine the input as a sequence of overlapping phrases, the algorithm divides each phrase into two parts, a multi-word *prefix* and a single *suffix* word that follows the prefix. A Markov chain algorithm emits output phrases by randomly choosing the suffix that follows the prefix, according to the statistics of (in our case) the original text. Three-word phrases work well—a two-word prefix is used to select the suffix word:

```
set  $w_1$  and  $w_2$  to the first two words in the text  
print  $w_1$  and  $w_2$   
loop:  
    randomly choose  $w_3$ , one of the successors of prefix  $w_1 w_2$  in the text  
    print  $w_3$   
    replace  $w_1$  and  $w_2$  by  $w_2$  and  $w_3$   
    repeat loop
```

To illustrate, suppose we want to generate random text based on a few sentences paraphrased from the epigraph above, using two-word prefixes:

Show your flowcharts and conceal your tables and I will be mystified. Show your tables and your flowcharts will be obvious. (*end*)

These are some of the pairs of input words and the words that follow them:

*Input prefix:*

Show your  
your flowcharts  
flowcharts and  
flowcharts will  
your tables  
will be  
be mystified.  
be obvious.

*Suffix words that follow:*

flowcharts tables  
and will  
conceal  
be  
and and  
mystified. obvious.  
Show  
(end)

A Markov algorithm processing this text will begin by printing Show your and will then randomly pick either flowcharts or tables. If it chooses the former, the current prefix becomes your flowcharts and the next word will be and or will. If it chooses tables, the next word will be and. This continues until enough output has been generated or until the end-marker is encountered as a suffix.

Our program will read a piece of English text and use a Markov chain algorithm to generate new text based on the frequency of appearance of phrases of a fixed length. The number of words in the prefix, which is two in our example, is a parameter. Making the prefix shorter tends to produce less coherent prose; making it longer tends to reproduce the input text verbatim. For English text, using two words to select a third is a good compromise; it seems to recreate the flavor of the input while adding its own whimsical touch.

What is a word? The obvious answer is a sequence of alphabetic characters, but it is desirable to leave punctuation attached to the words so “words” and “words.” are different. This helps to improve the quality of the generated prose by letting punctuation, and therefore (indirectly) grammar, influence the word choice, although it also permits unbalanced quotes and parentheses to sneak in. We will therefore define a “word” as anything between white space, a decision that places no restriction on input language and leaves punctuation attached to the words. Since most programming languages have facilities to split text into white-space-separated words, this is also easy to implement.

Because of the method, all words, all two-word phrases, and all three-word phrases in the output must have appeared in the input, but there should be many four-word and longer phrases that are synthesized. Here are a few sentences produced by the program we will develop in this chapter, when given the text of Chapter VII of *The Sun Also Rises* by Ernest Hemingway:

As I started up the undershirt onto his chest black, and big stomach muscles bulging under the light. "You see them?" Below the line where his ribs stopped were two raised white welts. "See on the forehead." "Oh, Brett, I love you." "Let's not talk. Talking's all bilge. I'm going away tomorrow." "Tomorrow?" "Yes. Didn't I say so? I am." "Let's have a drink, then."

We were lucky here that punctuation came out correctly; that need not happen.

## 3.2 Data Structure Alternatives

How much input do we intend to deal with? How fast must the program run? It seems reasonable to ask our program to read in a whole book, so we should be prepared for input sizes of  $n = 100,000$  words or more. The output will be hundreds or perhaps thousands of words, and the program should run in a few seconds instead of minutes. With 100,000 words of input text,  $n$  is fairly large so the algorithms can't be too simplistic if we want the program to be fast.

The Markov algorithm must see all the input before it can begin to generate output, so it must store the entire input in some form. One possibility is to read the whole input and store it in a long string, but we clearly want the input broken down into words. If we store it as an array of pointers to words, output generation is simple: to produce each word, scan the input text to see what possible suffix words follow the prefix that was just emitted, and then choose one at random. However, that means scanning all 100,000 input words for each word we generate; 1,000 words of output means hundreds of millions of string comparisons, which will not be fast.

Another possibility is to store only unique input words, together with a list of where they appear in the input so that we can locate successor words more quickly. We could use a hash table like the one in Chapter 2, but that version doesn't directly address the needs of the Markov algorithm, which must quickly locate all the suffixes of a given prefix.

We need a data structure that better represents a prefix and its associated suffixes. The program will have two passes, an input pass that builds the data structure representing the phrases, and an output pass that uses the data structure to generate the random output. In both passes, we need to look up a prefix (quickly): in the input pass to update its suffixes, and in the output pass to select at random from the possible suffixes. This suggests a hash table whose keys are prefixes and whose values are the sets of suffixes for the corresponding prefixes.

For purposes of description, we'll assume a two-word prefix, so each output word is based on the pair of words that precede it. The number of words in the prefix doesn't affect the design and the programs should handle any prefix length, but selecting a number makes the discussion concrete. The prefix and the set of all its possible suffixes we'll call a *state*, which is standard terminology for Markov algorithms.

Given a prefix, we need to store all the suffixes that follow it so we can access them later. The suffixes are unordered and added one at a time. We don't know how many there will be, so we need a data structure that grows easily and efficiently, such as a list or a dynamic array. When we are generating output, we need to be able to choose one suffix at random from the set of suffixes associated with a particular prefix. Items are never deleted.

What happens if a phrase appears more than once? For example, ‘might appear twice’ might appear twice but ‘might appear once’ only once. This could be represented by putting ‘twice’ twice in the suffix list for ‘might appear’ or by putting it in once, with an associated counter set to 2. We've tried it with and without counters;

without is easier, since adding a suffix doesn't require checking whether it's there already, and experiments showed that the difference in run-time was negligible.

In summary, each state comprises a prefix and a list of suffixes. This information is stored in a hash table, with prefix as key. Each prefix is a fixed-size set of words. If a suffix occurs more than once for a given prefix, each occurrence will be included separately in the list.

The next decision is how to represent the words themselves. The easy way is to store them as individual strings. Since most text has many words appearing multiple times, it would probably save storage if we kept a second hash table of single words, so the text of each word was stored only once. This would also speed up hashing of prefixes, since we could compare pointers rather than individual characters: unique strings have unique addresses. We'll leave that design as an exercise; for now, strings will be stored individually.

### 3.3 Building the Data Structure in C

Let's begin with a C implementation. The first step is to define some constants.

```
enum {
    NPREF = 2,      /* number of prefix words */
    NHASH = 4093,   /* size of state hash table array */
    MAXGEN = 10000 /* maximum words generated */
};
```

This declaration defines the number of words (NPREF) for the prefix, the size of the hash table array (NHASH), and an upper limit on the number of words to generate (MAXGEN). If NPREF is a compile-time constant rather than a run-time variable, storage management is simpler. The array size is set fairly large because we expect to give the program large input documents, perhaps a whole book. We chose NHASH = 4093 so that if the input has 10,000 distinct prefixes (word pairs), the average chain will be very short, two or three prefixes. The larger the size, the shorter the expected length of the chains and thus the faster the lookup. This program is really a toy, so the performance isn't critical, but if we make the array too small the program will not handle our expected input in reasonable time; on the other hand, if we make it too big it might not fit in the available memory.

The prefix can be stored as an array of words. The elements of the hash table will be represented as a State data type, associating the Suffix list with the prefix:

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

```

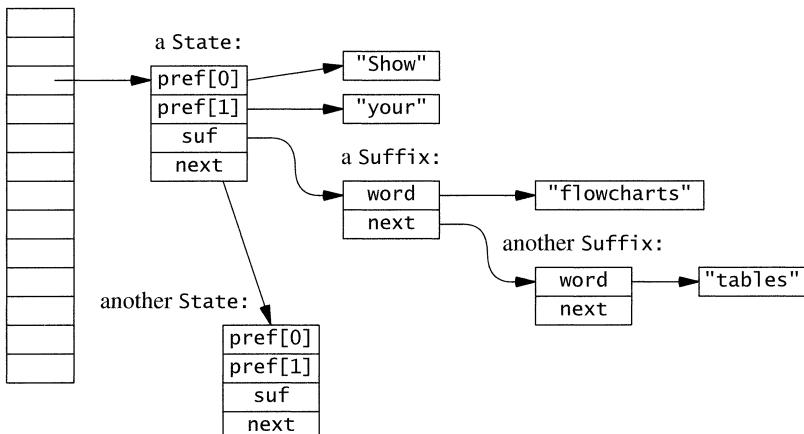
struct Suffix { /* list of suffixes */
    char    *word;           /* suffix */
    Suffix  *next;          /* next in list of suffixes */
};

State  *statetab[NHASH]; /* hash table of states */

```

Pictorially, the data structures look like this:

statetab:



We need a hash function for prefixes, which are arrays of strings. It is simple to modify the string hash function from Chapter 2 to loop over the strings in the array, thus in effect hashing the concatenation of the strings:

```

/* hash: compute hash value for array of NPREF strings */
unsigned int hash(char *s[NPREF])
{
    unsigned int h;
    unsigned char *p;
    int i;

    h = 0;
    for (i = 0; i < NPREF; i++)
        for (p = (unsigned char *) s[i]; *p != '\0'; p++)
            h = MULTIPLIER * h + *p;
    return h % NHASH;
}

```

A similar modification to the lookup routine completes the implementation of the hash table:

```
/* Lookup: search for prefix; create if requested. */
/* returns pointer if present or created; NULL if not. */
/* creation doesn't strdup so strings mustn't change later. */
State* lookup(char *prefix[NPREF], int create)
{
    int i, h;
    State *sp;

    h = hash(prefix);
    for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
            if (strcmp(prefix[i], sp->pref[i]) != 0)
                break;
        if (i == NPREF) /* found it */
            return sp;
    }
    if (create) {
        sp = (State *) emalloc(sizeof(State));
        for (i = 0; i < NPREF; i++)
            sp->pref[i] = prefix[i];
        sp->suf = NULL;
        sp->next = statetab[h];
        statetab[h] = sp;
    }
    return sp;
}
```

Notice that `lookup` doesn't make a copy of the incoming strings when it creates a new state; it just stores pointers in `sp->pref[]`. Callers of `lookup` must guarantee that the data won't be overwritten later. For example, if the strings are in an I/O buffer, a copy must be made before `lookup` is called; otherwise, subsequent input could overwrite the data that the hash table points to. Decisions about who owns a resource shared across an interface arise often. We will explore this topic at length in the next chapter.

Next we need to build the hash table as the file is read:

```
/* build: read input, build prefix table */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* create a format string; %s could overflow buf */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

The peculiar call to `sprintf` gets around an irritating problem with `fscanf`, which is otherwise perfect for the job. A call to `fscanf` with format `%s` will read the next white-space-delimited word from the file into the buffer, but there is no limit on size: a long word might overflow the input buffer, wreaking havoc. If the buffer is 100

bytes long (which is far beyond what we expect ever to appear in normal text), we can use the format `%99s` (leaving one byte for the terminal '`\0`'), which tells `fscanf` to stop after 99 bytes. A long word will be broken into pieces, which is unfortunate but safe. We could declare

```
? enum { BUFSIZE = 100 };
?     char   fmt[] = "%99s"; /* BUFSIZE-1 */
```

but that requires two constants for one arbitrary decision—the size of the buffer—and introduces the need to maintain their relationship. The problem can be solved once and for all by creating the format string dynamically with `sprintf`, so that's the approach we take.

The two arguments to `build` are the `prefix` array holding the previous `NPREF` words of input and a `FILE` pointer. It passes the `prefix` and a copy of the input word to `add`, which adds the new entry to the hash table and advances the `prefix`:

```
/* add: add word to suffix list, update prefix */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;
    sp = lookup(prefix, 1); /* create if not found */
    addsuffix(sp, suffix);
    /* move the words down the prefix */
    memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
    prefix[NPREF-1] = suffix;
}
```

The call to `memmove` is the idiom for deleting from an array. It shifts elements 1 through `NPREF-1` in the `prefix` down to positions 0 through `NPREF-2`, deleting the first `prefix` word and opening a space for a new one at the end.

The `addsuffix` routine adds the new suffix:

```
/* addsuffix: add to state. suffix must not change later */
void addsuffix(State *sp, char *suffix)
{
    Suffix *suf;
    suf = (Suffix *) malloc(sizeof(Suffix));
    suf->word = suffix;
    suf->next = sp->suf;
    sp->suf = suf;
}
```

We split the action of updating the state into two functions: `add` performs the general service of adding a suffix to a `prefix`, while `addsuffix` performs the implementation-specific action of adding a word to a suffix list. The `add` routine is used by `build`, but `addsuffix` is used internally only by `add`; it is an implementation detail that might change and it seems better to have it in a separate function, even though it is called in only one place.

## 3.4 Generating Output

With the data structure built, the next step is to generate the output. The basic idea is as before: given a prefix, select one of its suffixes at random, print it, then advance the prefix. This is the steady state of processing; we must still figure out how to start and stop the algorithm. Starting is easy if we remember the words of the first prefix and begin with them. Stopping is easy, too. We need a marker word to terminate the algorithm. After all the regular input, we can add a terminator, a “word” that is guaranteed not to appear in any input:

```
build(prefix, stdin);
add(prefix, NONWORD);
```

NONWORD should be some value that will never be encountered in regular input. Since the input words are delimited by white space, a “word” of white space will serve, such as a newline character:

```
char NONWORD[] = "\n"; /* cannot appear as real word */
```

One more worry: what happens if there is insufficient input to start the algorithm? There are two approaches to this sort of problem, either exit prematurely if there is insufficient input, or arrange that there is always enough and don’t bother to check. In this program, the latter approach works well.

We can initialize building and generating with a fabricated prefix, which guarantees there is always enough input for the program. To prime the loops, initialize the prefix array to be all NONWORD words. This has the nice benefit that the first word of the input file will be the first *suffix* of the fake prefix, so the generation loop needs to print only the suffixes it produces.

In case the output is unmanageably long, we can terminate the algorithm after some number of words are produced or when we hit NONWORD as a suffix, whichever comes first.

Adding a few NONWORDS to the ends of the data simplifies the main processing loops of the program significantly; it is an example of the technique of adding *sentinel* values to mark boundaries.

As a rule, try to handle irregularities and exceptions and special cases in data. Code is harder to get right so the control flow should be as simple and regular as possible.

The generate function uses the algorithm we sketched originally. It produces one word per line of output, which can be grouped into longer lines with a word processor; Chapter 9 shows a simple formatter called `fmt` for this task.

With the use of the initial and final NONWORD strings, generate starts and stops properly:

```

/* generate: produce output, one word per line */
void generate(int nwords)
{
    State *sp;
    Suffix *suf;
    char *prefix[NPREF], *w;
    int i, nmatch;

    for (i = 0; i < NPREF; i++) /* reset initial prefix */
        prefix[i] = NONWORD;

    for (i = 0; i < nwords; i++) {
        sp = lookup(prefix, 0);
        nmatch = 0;
        for (suf = sp->suf; suf != NULL; suf = suf->next)
            if (rand() % ++nmatch == 0) /* prob = 1/nmatch */
                w = suf->word;
        if (strcmp(w, NONWORD) == 0)
            break;
        printf("%s\n", w);
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = w;
    }
}

```

Notice the algorithm for selecting one item at random when we don't know how many items there are. The variable `nmatch` counts the number of matches as the list is scanned. The expression

```
rand() % ++nmatch == 0
```

increments `nmatch` and is then true with probability  $1/nmatch$ . Thus the first matching item is selected with probability 1, the second will replace it with probability 1/2, the third will replace the survivor with probability 1/3, and so on. At any time, each one of the  $k$  matching items seen so far has been selected with probability 1/ $k$ .

At the beginning, we set the `prefix` to the starting value, which is guaranteed to be installed in the hash table. The first `Suffix` values we find will be the first words of the document, since they are the unique follow-on to the starting prefix. After that, random suffixes will be chosen. The loop calls `lookup` to find the hash table entry for the current `prefix`, then chooses a random suffix, prints it, and advances the `prefix`.

If the suffix we choose is `NONWORD`, we're done, because we have chosen the state that corresponds to the end of the input. If the suffix is not `NONWORD`, we print it, then drop the first word of the `prefix` with a call to `memmove`, promote the suffix to be the last word of the `prefix`, and loop.

Now we can put all this together into a `main` routine that reads the standard input and generates at most a specified number of words:

```
/* markov main: markov-chain random text generation */
int main(void)
{
    int i, nwords = MAXGEN;
    char *prefix[NPREF];           /* current input prefix */
    for (i = 0; i < NPREF; i++) /* set up initial prefix */
        prefix[i] = NONWORD;
    build(prefix, stdin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

This completes our C implementation. We will return at the end of the chapter to a comparison of programs in different languages. The great strengths of C are that it gives the programmer complete control over implementation, and programs written in it tend to be fast. The cost, however, is that the C programmer must do more of the work, allocating and reclaiming memory, creating hash tables and linked lists, and the like. C is a razor-sharp tool, with which one can create an elegant and efficient program or a bloody mess.

**Exercise 3-1.** The algorithm for selecting a random item from a list of unknown length depends on having a good random number generator. Design and carry out experiments to determine how well the method works in practice. □

**Exercise 3-2.** If each input word is stored in a second hash table, the text is only stored once, which should save space. Measure some documents to estimate how much. This organization would allow us to compare pointers rather than strings in the hash chains for prefixes, which should run faster. Implement this version and measure the change in speed and memory consumption. □

**Exercise 3-3.** Remove the statements that place sentinel NONWORDS at the beginning and end of the data, and modify generate so it starts and stops properly without them. Make sure it produces correct output for input with 0, 1, 2, 3, and 4 words. Compare this implementation to the version using sentinels. □

## 3.5 Java

Our second implementation of the Markov chain algorithm is in Java. Object-oriented languages like Java encourage one to pay particular attention to the interfaces between the components of the program, which are then encapsulated as independent data items called objects or classes, with associated functions called methods.

Java has a richer library than C, including a set of *container classes* to group existing objects in various ways. One example is a `Vector` that provides a dynamically-growable array that can store any `Object` type. Another example is the `Hashtable`

class, with which one can store and retrieve values of one type using objects of another type as keys.

In our application, Vectors of strings are the natural choice to hold prefixes and suffixes. We can use a `Hashtable` whose keys are prefix vectors and whose values are suffix vectors. The terminology for this type of construction is a *map* from prefixes to suffixes; in Java, we need no explicit `State` type because `Hashtable` implicitly connects (maps) prefixes to suffixes. This design is different from the C version, in which we installed `State` structures that held both prefix and suffix list, and hashed on the prefix to recover the full `State`.

A `Hashtable` provides a `put` method to store a key-value pair, and a `get` method to retrieve the value for a key:

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

Our implementation has three classes. The first class, `Prefix`, holds the words of the prefix:

```
class Prefix {
    public Vector pref; // NPREF adjacent words from input
    ...
}
```

The second class, `Chain`, reads the input, builds the hash table, and generates the output; here are its class variables:

```
class Chain {
    static final int NPREF = 2; // size of prefix
    static final String NONWORD = "\n";
        // "word" that can't appear
    Hashtable statetab = new Hashtable();
        // key = Prefix, value = suffix Vector
    Prefix prefix = new Prefix(NPREF, NONWORD);
        // initial prefix
    Random rand = new Random();
    ...
}
```

The third class is the public interface; it holds `main` and instantiates a `Chain`:

```
class Markov {
    static final int MAXGEN = 10000; // maximum words generated
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;
        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

When an instance of class `Chain` is created, it in turn creates a hash table and sets up the initial prefix of NPREF NONWORDs. The `build` function uses the library function `StreamTokenizer` to parse the input into words separated by white space characters. The three calls before the loop set the tokenizer into the proper state for our definition of “word.”

```
// Chain build: build State table from input stream
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();           // remove default rules
    st.wordChars(0, Character.MAX_VALUE); // turn on all chars
    st.whitespaceChars(0, ' ');      // except up to blank
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

The `add` function retrieves the vector of suffixes for the current prefix from the hash table; if there are none (the vector is null), `add` creates a new vector and a new prefix to store in the hash table. In either case, it adds the new word to the suffix vector and advances the prefix by dropping the first word and adding the new word at the end.

```
// Chain add: add word to suffix list, update prefix
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);
    if (suf == null) {
        suf = new Vector();
        statetab.put(new Prefix(prefix), suf);
    }
    suf.addElement(word);
    prefix.pref.removeElementAt(0);
    prefix.pref.addElement(word);
}
```

Notice that if `suf` is null, `add` installs a new `Prefix` in the hash table, rather than `prefix` itself. This is because the `Hashtable` class stores items by reference, and if we don't make a copy, we could overwrite data in the table. This is the same issue that we had to deal with in the C program.

The generation function is similar to the C version, but slightly more compact because it can index a random vector element directly instead of looping through a list.

```

// Chain generate: generate output words
void generate(int nwords)
{
    prefix = new Prefix(NPREF, NONWORD);
    for (int i = 0; i < nwords; i++) {
        Vector s = (Vector) statetab.get(prefix);
        int r = Math.abs(rand.nextInt()) % s.size();
        String suf = (String) s.elementAt(r);
        if (suf.equals(NONWORD))
            break;
        System.out.println(suf);
        prefix.pref.removeElementAt(0);
        prefix.pref.addElement(suf);
    }
}

```

The two constructors of `Prefix` create new instances from supplied data. The first copies an existing `Prefix`, and the second creates a prefix from `n` copies of a string; we use it to make `NPREF` copies of `NONWORD` when initializing:

```

// Prefix constructor: duplicate existing prefix
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

// Prefix constructor: n copies of str
Prefix(int n, String str)
{
    pref = new Vector();
    for (int i = 0; i < n; i++)
        pref.addElement(str);
}

```

`Prefix` also has two methods, `hashCode` and `equals`, that are called implicitly by the implementation of `Hashtable` to index and search the table. It is the need to have an explicit class for these two methods for `Hashtable` that forced us to make `Prefix` a full-fledged class, rather than just a `Vector` like the suffix.

The `hashCode` method builds a single hash value by combining the set of `hashCodes` for the elements of the vector:

```

static final int MULTIPLIER = 31; // for hashCode()

// Prefix hashCode: generate hash from all prefix words
public int hashCode()
{
    int h = 0;
    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}

```

and `equals` does an elementwise comparison of the words in two prefixes:

```
// Prefix equals: compare two prefixes for equal words
public boolean equals(Object o)
{
    Prefix p = (Prefix) o;
    for (int i = 0; i < pref.size(); i++)
        if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
            return false;
    return true;
}
```

The Java program is significantly smaller than the C program and takes care of more details; `Vectors` and the `Hashtable` are the obvious examples. In general, storage management is easy since vectors grow as needed and garbage collection takes care of reclaiming memory that is no longer referenced. But to use the `Hashtable` class, we still need to write functions `hashCode` and `equals`, so Java isn't taking care of all the details.

Comparing the way the C and Java programs represent and operate on the same basic data structure, we see that the Java version has better separation of functionality. For example, to switch from `Vectors` to arrays would be easy. In the C version, everything knows what everything else is doing: the hash table operates on arrays that are maintained in various places, `lookup` knows the layout of the `State` and `Suffix` structures, and everyone knows the size of the prefix array.

```
% java Markov <jr_chemistry.txt | fmt
Wash the blackboard. Watch it dry. The water goes
into the air. When water goes into the air it
evaporates. Tie a damp cloth to one end of a solid or
liquid. Look around. What are the solid things?
Chemical changes take place when something burns. If
the burning material has liquids, they are stable and
the sponge rise. It looked like dough, but it is
burning. Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
```

**Exercise 3-4.** Revise the Java version of `markov` to use an array instead of a `Vector` for the prefix in the `State` class. □

## 3.6 C++

Our third implementation is in C++. Since C++ is almost a superset of C, it can be used as if it were C with a few notational conveniences, and our original C version of `markov` is also a legal C++ program. A more appropriate use of C++, however, would be to define classes for the objects in the program, more or less as we did in Java; this would let us hide implementation details. We decided to go even further by using the Standard Template Library or STL, since the STL has built-in mechanisms that will do much of what we need. The ISO standard for C++ includes the STL as part of the language definition.

The STL provides containers such as vectors, lists, and sets, and a family of fundamental algorithms for searching, sorting, inserting, and deleting. Using the template features of C++, every STL algorithm works on a variety of containers, including both user-defined types and built-in types like integers. Containers are expressed as C++ templates that are instantiated for specific data types; for example, there is a `vector` container that can be used to make particular types like `vector<int>` or `vector<string>`. All `vector` operations, including standard algorithms for sorting, can be used on such data types.

In addition to a `vector` container that is similar to Java's `Vector`, the STL provides a `deque` container. A `deque` (pronounced "deck") is a double-ended queue that matches what we do with prefixes: it holds NPREF elements, and lets us pop the first element and add a new one to the end, in  $O(1)$  time for both. The STL `deque` is more general than we need, since it permits push and pop at either end, but the performance guarantees make it an obvious choice.

The STL also provides an explicit `map` container, based on balanced trees, that stores key-value pairs and provides  $O(\log n)$  retrieval of the value associated with any key. Maps might not be as efficient as  $O(1)$  hash tables, but it's nice not to have to write any code whatsoever to use them. (Some non-standard C++ libraries include a `hash` or `hash_map` container whose performance may be better.)

We also use the built-in comparison functions, which in this case will do string comparisons using the individual strings in the prefix.

With these components in hand, the code goes together smoothly. Here are the declarations:

```
typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab; // prefix -> suffixes
```

The STL provides a template for deques; the notation `deque<string>` specializes it to a deque whose elements are strings. Since this type appears several times in the program, we used a `typedef` to give it the name `Prefix`. The map type that stores prefixes and suffixes occurs only once, however, so we did not give it a separate name; the `map` declaration declares a variable `statetab` that is a map from prefixes to vectors of strings. This is more convenient than either C or Java, because we don't need to provide a hash function or `equals` method.

The main routine initializes the prefix, reads the input (from standard input, called `cin` in the C++ `iostream` library), adds a tail, and generates the output, exactly as in the earlier versions:

```
// markov main: markov-chain random text generation
int main(void)
{
    int nwords = MAXGEN;
    Prefix prefix; // current input prefix
    for (int i = 0; i < NPREF; i++) // set up initial prefix
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

The function `build` uses the `iostream` library to read the input one word at a time:

```
// build: read input words, build state table
void build(Prefix& prefix, istream& in)
{
    string buf;
    while (in >> buf)
        add(prefix, buf);
}
```

The string `buf` will grow as necessary to handle input words of arbitrary length.

The `add` function shows more of the advantages of using the STL:

```
// add: add word to suffix list, update prefix
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

Quite a bit is going on under these apparently simple statements. The `map` container overloads subscripting (the `[]` operator) to behave as a lookup operation. The expression `statetab[prefix]` does a lookup in `statetab` with `prefix` as key and returns a reference to the desired entry; the vector is created if it does not exist already. The `push_back` member functions of `vector` and `deque` push a new string onto the back end of the vector or deque; `pop_front` pops the first element off the deque.

Generation is similar to the previous versions:

```
// generate: produce output, one word per line
void generate(int nwords)
{
    Prefix prefix;
    int i;
    for (i = 0; i < NPREF; i++) // reset initial prefix
        add(prefix, NONWORD);
    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();      // advance
        prefix.push_back(w);
    }
}
```

Overall, this version seems especially clear and elegant—the code is compact, the data structure is visible and the algorithm is completely transparent. Sadly, there is a price to pay: this version runs much slower than the original C version, though it is not the slowest. We'll come back to performance measurements shortly.

**Exercise 3-5.** The great strength of the STL is the ease with which one can experiment with different data structures. Modify the C++ version of Markov to use various structures to represent the prefix, suffix list, and state table. How does performance change for the different structures? □

**Exercise 3-6.** Write a C++ version that uses only classes and the `string` data type but no other advanced library facilities. Compare it in style and speed to the STL versions. □

## 3.7 Awk and Perl

To round out the exercise, we also wrote the program in two popular scripting languages, Awk and Perl. These provide the necessary features for this application, associative arrays and string handling.

An *associative array* is a convenient packaging of a hash table; it looks like an array but its subscripts are arbitrary strings or numbers, or comma-separated lists of them. It is a form of map from one data type to another. In Awk, all arrays are associative; Perl has both conventional indexed arrays with integer subscripts and associative arrays, which are called “hashes,” a name that suggests how they are implemented.

The Awk and Perl implementations are specialized to prefixes of length 2.

```

# markov.awk: markov chain algorithm for 2-word prefixes
BEGIN { MAXGEN = 10000; NONWORD = "\n"; w1 = w2 = NONWORD }
{
    for (i = 1; i <= NF; i++) {      # read all words
        statetab[w1,w2,++nsuffix[w1,w2]] = $i
        w1 = w2
        w2 = $i
    }
}
END {
    statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD  # add tail
    w1 = w2 = NONWORD
    for (i = 0; i < MAXGEN; i++) { # generate
        r = int(rand()*nsuffix[w1,w2]) + 1  # nsuffix >= 1
        p = statetab[w1,w2,r]
        if (p == NONWORD)
            exit
        print p
        w1 = w2          # advance chain
        w2 = p
    }
}

```

Awk is a pattern-action language: the input is read a line at a time, each line is matched against the patterns, and for each match the corresponding action is executed. There are two special patterns, BEGIN and END, that match before the first line of input and after the last.

An action is a block of statements enclosed in braces. In the Awk version of Markov, the BEGIN block initializes the prefix and a couple of other variables.

The next block has no pattern, so by default it is executed once for each input line. Awk automatically splits each input line into fields (white-space delimited words) called \$1 through \$NF; the variable NF is the number of fields. The statement

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

builds the map from prefix to suffixes. The array nsuffix counts suffixes and the element nsuffix[w1,w2] counts the number of suffixes associated with that prefix. The suffixes themselves are stored in array elements statetab[w1,w2,1], statetab[w1,w2,2], and so on.

When the END block is executed, all the input has been read. At that point, for each prefix there is an element of nsuffix containing the suffix count, and there are that many elements of statetab containing the suffixes.

The Perl version is similar, but uses an anonymous array instead of a third subscript to keep track of suffixes; it also uses multiple assignment to update the prefix. Perl uses special characters to indicate the types of variables: \$ marks a scalar and @ an indexed array, while brackets [] are used to index arrays and braces {} to index hashes.

```

# markov.pl: markov chain algorithm for 2-word prefixes

$MAXGEN = 10000;
$NONWORD = "\n";
$w1 = $w2 = $NONWORD;                      # initial state
while (<>) {                                # read each line of input
    foreach (split) {
        push(@{$statetab{$w1}{$w2}}, $_);
        ($w1, $w2) = ($w2, $_); # multiple assignment
    }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD);      # add tail

$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
    $suf = $statetab{$w1}{$w2}; # array reference
    $r = int(rand @$suf);       # @$suf is number of elems
    exit if (($t = $suf->[$r]) eq $NONWORD);
    print "$t\n";
    ($w1, $w2) = ($w2, $t);     # advance chain
}

```

As in the previous programs, the map is stored using the variable `statetab`. The heart of the program is the line

```
push(@{$statetab{$w1}{$w2}}, $_);
```

which pushes a new suffix onto the end of the (anonymous) array stored at `statetab{$w1}{$w2}`. In the generation phase, `$statetab{$w1}{$w2}` is a reference to an array of suffixes, and `$suf->[$r]` points to the `r`-th suffix.

Both the Perl and Awk programs are short compared to the three earlier versions, but they are harder to adapt to handle prefixes that are not exactly two words. The core of the C++ STL implementation (the `add` and `generate` functions) is of comparable length and seems clearer. Nevertheless, scripting languages are often a good choice for experimental programming, for making prototypes, and even for production use if run-time is not a major issue.

**Exercise 3-7.** Modify the Awk and Perl versions to handle prefixes of any length. Experiment to determine what effect this change has on performance. □

## 3.8 Performance

We have several implementations to compare. We timed the programs on the Book of Psalms from the King James Bible, which has 42,685 words (5,238 distinct words, 22,482 prefixes). This text has enough repeated phrases (“Blessed is the ...”)

that one suffix list has more than 400 elements, and there are a few hundred chains with dozens of suffixes, so it is a good test data set.

Blessed is the man of the net. Turn thee unto me, and raise me up, that I may tell all my fears. They looked unto him, he heard. My praise shall be blessed. Wealth and riches shall be saved. Thou hast dealt well with thy hid treasure: they are cast into a standing water, the flint into a standing water, and dry ground into watersprings.

The times in the following table are the number of seconds for generating 10,000 words of output; one machine is a 250MHz MIPS R10000 running Irix 6.4 and the other is a 400MHz Pentium II with 128 megabytes of memory running Windows NT. Run-time is almost entirely determined by the input size; generation is very fast by comparison. The table also includes the approximate program size in lines of source code.

|               | 250MHz<br>R10000 | 400MHz<br>Pentium II | Lines of<br>source code |
|---------------|------------------|----------------------|-------------------------|
| C             | 0.36 sec         | 0.30 sec             | 150                     |
| Java          | 4.9              | 9.2                  | 105                     |
| C++/STL/deque | 2.6              | 11.2                 | 70                      |
| C++/STL/list  | 1.7              | 1.5                  | 70                      |
| Awk           | 2.2              | 2.1                  | 20                      |
| Perl          | 1.8              | 1.0                  | 18                      |

The C and C++ versions were compiled with optimizing compilers, while the Java runs had just-in-time compilers enabled. The Irix C and C++ times are the fastest obtained from three different compilers; similar results were observed on Sun SPARC and DEC Alpha machines. The C version of the program is fastest by a large factor; Perl comes second. The times in the table are a snapshot of our experience with a particular set of compilers and libraries, however, so you may see very different results in your environment.

Something is clearly wrong with the STL deque version on Windows. Experiments showed that the deque that represents the prefix accounts for most of the run-time, although it never holds more than two elements; we would expect the central data structure, the map, to dominate. Switching from a deque to a list (which is a doubly-linked list in the STL) improves the time dramatically. On the other hand, switching from a map to a (non-standard) hash container made no difference on Irix; hashes were not available on our Windows machine. It is a testament to the fundamental soundness of the STL design that these changes required only substituting the word `list` for the word `deque` or `hash` for `map` in two places and recompiling. We conclude that the STL, which is a new component of C++, still suffers from immature implementations. The performance is unpredictable between implementations of the STL and between individual data structures. The same is true of Java, where implementations are also changing rapidly.

There are some interesting challenges in testing a program that is meant to produce voluminous random output. How do we know it works at all? How do we know it works all the time? Chapter 6, which discusses testing, contains some suggestions and describes how we tested the Markov programs.

## 3.9 Lessons

The Markov program has a long history. The first version was written by Don P. Mitchell, adapted by Bruce Ellis, and applied to humorous deconstructionist activities throughout the 1980s. It lay dormant until we thought to use it in a university course as an illustration of program design. Rather than dusting off the original, we rewrote it from scratch in C to refresh our memories of the various issues that arise, and then wrote it again in several other languages, using each language's unique idioms to express the same basic idea. After the course, we reworked the programs many times to improve clarity and presentation.

Over all that time, however, the basic design has remained the same. The earliest version used the same approach as the ones we have presented here, although it did employ a second hash table to represent individual words. If we were to rewrite it again, we would probably not change much. The design of a program is rooted in the layout of its data. The data structures don't define every detail, but they do shape the overall solution.

Some data structure choices make little difference, such as lists versus growable arrays. Some implementations generalize better than others—the Perl and Awk code could be readily modified to one- or three-word prefixes but parameterizing the choice would be awkward. As befits object-oriented languages, tiny changes to the C++ and Java implementations would make the data structures suitable for objects other than English text, for instance programs (where white space would be significant), or notes of music, or even mouse clicks and menu selections for generating test sequences.

Of course, while the data structures are much the same, there is a wide variation in the general appearance of the programs, in the size of the source code, and in performance. Very roughly, higher-level languages give slower programs than lower level ones, although it's unwise to generalize other than qualitatively. Big building-blocks like the C++ STL or the associative arrays and string handling of scripting languages can lead to more compact code and shorter development time. These are not without price, although the performance penalty may not matter much for programs, like Markov, that run for only a few seconds.

Less clear, however, is how to assess the loss of control and insight when the pile of system-supplied code gets so big that one no longer knows what's going on underneath. This is the case with the STL version; its performance is unpredictable and there is no easy way to address that. One immature implementation we used needed

to be repaired before it would run our program. Few of us have the resources or the energy to track down such problems and fix them.

This is a pervasive and growing concern in software: as libraries, interfaces, and tools become more complicated, they become less understood and less controllable. When everything works, rich programming environments can be very productive, but when they fail, there is little recourse. Indeed, we may not even realize that something is wrong if the problems involve performance or subtle logic errors.

The design and implementation of this program illustrate a number of lessons for larger programs. First is the importance of choosing simple algorithms and data structures, the simplest that will do the job in reasonable time for the expected problem size. If someone else has already written them and put them in a library for you, that's even better; our C++ implementation profited from that.

Following Brooks's advice, we find it best to start detailed design with data structures, guided by knowledge of what algorithms might be used; with the data structures settled, the code goes together easily.

It's hard to design a program completely and then build it; constructing real programs involves iteration and experimentation. The act of building forces one to clarify decisions that had previously been glossed over. That was certainly the case with our programs here, which have gone through many changes of detail. As much as possible, start with something simple and evolve it as experience dictates. If our goal had been just to write a personal version of the Markov chain algorithm for fun, we would almost surely have written it in Awk or Perl—though not with as much polishing as the ones we showed here—and let it go at that.

Production code takes much more effort than prototypes do, however. If we think of the programs presented here as *production code* (since they have been polished and thoroughly tested), production quality requires one or two orders of magnitude more effort than a program intended for personal use.

**Exercise 3-8.** We have seen versions of the Markov program in a wide variety of languages, including Scheme, Tcl, Prolog, Python, Generic Java, ML, and Haskell; each presents its own challenges and advantages. Implement the program in your favorite language and compare its general flavor and performance. □

## Supplementary Reading

The Standard Template Library is described in a variety of books, including *Generic Programming and the STL*, by Matthew Austern (Addison-Wesley, 1998). The definitive reference on C++ itself is *The C++ Programming Language*, by Bjarne Stroustrup (3rd edition, Addison-Wesley, 1997). For Java, we refer to *The Java Programming Language, 2nd Edition* by Ken Arnold and James Gosling (Addison-Wesley, 1998). The best description of Perl is *Programming Perl, 2nd Edition*, by Larry Wall, Tom Christiansen, and Randal Schwartz (O'Reilly, 1996).

The idea behind *design patterns* is that there are only a few distinct design constructs in most programs in the same way that there are only a few basic data structures; very loosely, it is the design analog of the code idioms that we discussed in Chapter 1. The standard reference is *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

The picaresque adventures of the `markov` program, originally called `shaney`, were described in the “Computing Recreations” column of the June, 1989 *Scientific American*. The article was republished in *The Magic Machine*, by A. K. Dewdney (W. H. Freeman, 1990).

*This page intentionally left blank*

---

---

# Index

Woman: *Is my Aunt Minnie in here?*

Driftwood: *Well, you can come in and prowl around if you want to.  
If she isn't in here, you can probably find somebody just as good.*

The Marx Brothers, *A Night at the Opera*

- 0, *see* zero, notation for
  - 1/k random selection, 70
  - \_\_ naming convention, 104
  - \$ end of string metacharacter, 222
  - & bitwise operator, 7, 127
  - && logical operator, 6, 193
  - '\0' null byte, 21
  - \*
    - wildcards, 106, 222
    - zero or more metacharacter, 223, 225, 227  - + one or more metacharacter, 223, 228
  - ++ increment operator, 9
  - . any character metacharacter, 223
  - ... ellipsis function parameter, 109, 218
  - = assignment operator, 9, 13
  - >> right shift operator, 8, 135, 194
  - >>= assignment operator, 8
  - >>> Java logical right shift operator, 194
  - ?
    - questionable code notation, 2, 88
    - zero or one metacharacter, 223, 228  - ? : conditional operator, 8, 193
  - [] character class metacharacter, 223, 228
  - \
    - line continuation character, 240
    - quote metacharacter, 223, 228  - ^ start of string metacharacter, 222
  - { } braces, position of, 10
  - |
    - OR metacharacter, 223
    - bitwise operator, 7, 127
- || logical operator, 6, 193
  - abort library function, 125
  - abstraction, 104, 202
  - add function, Markov C, 68
  - addend list function, 46
  - addfront list function, 46
  - addname list function, 42
  - addop function, 233, 244
  - addsuffix function, Markov C, 68
  - advquoted function, CSV, 97–98
  - Aho, Al, xii
  - algorithm
    - binary search, 31, 52
    - constant-time, 41, 44, 49, 55, 76
    - cubic, 41
    - exponential, 41
    - linear, 30, 41, 46–47
    - $\log n$ , 32, 41, 51–52, 76
    - Markov chain, 62–63
    - $n \log n$ , 34, 41
    - quadratic, 40, 43, 176
    - quicksort, 32
    - sequential search, 30
    - tree sort, 53
  - alignment, 206
    - structure member, 195
  - alloca function, 180
  - allocation
    - error, memory, 130
    - memory, 48, 67, 92

allocator, special-purpose, 180, 182  
 ambiguity  
     and parenthesization, 6  
     *if-else*, 10  
 analysis of algorithms, *see O*-notation  
 ANSI/ISO C standard, 190, 212  
 any character metacharacter, ., 223  
 application program interface (API), 105, 198  
*apply* list function, 47  
*applyinorder* tree function, 53  
*applypostorder* tree function, 54  
 approximate values, 181  
 Ariane 5 rocket, 157  
 arithmetic  
     IEEE floating-point, 112, 181, 193  
     shift, 135, 194  
 Arnold, Ken, xii, 83  
 array bounds, 14  
**Array**  
     Java, 39  
         `length` field, Java, 22  
 array, `static`, 131  
`*array[]` vs. `**array`, 30  
 arrays, growing, 41–44, 58, 92, 95, 97, 158  
 ASCII encoding, 210  
 assembly language, 152, 181, 237  
`assert` macro, 142  
`<assert.h>` header, 142  
 assignment  
     multiple, 9  
     operator, `=`, 9, 13  
     operator, `>=`, 8  
 associative array, *see also* hash table  
 associative array, 78, 82  
`atexit` library function, 107  
 Austern, Matthew, 83  
`avg` function, 141  
 Awk, 229  
     profile, 174  
     program, `fmt`, 229  
     program, Markov, 79  
     program, `split.awk`, 229  
     test, 150  
 backwards compatibility, 209, 211  
 balanced tree, 52, 76  
 benchmarking, 187  
 Bentley, Jon, xii, 59, 163, 188  
 beta release test, 160  
 Bigelow, Chuck, xii  
 big-endian, 204, 213  
 binary  
     files, 132, 157, 203  
     mode I/O, 134, 207  
 binary search  
     algorithm, 31, 52  
     for error, 124  
     function, lookup, 31, 36  
     testing, 146  
     tree, 50  
     tree diagram, 51  
`b1nhex` program, 203  
`bison` compiler-compiler, 232  
`bitblt` operator, 241  
 bitfields, 183, 191, 195  
 bitwise operator  
     &, 7, 127  
     |, 7, 127  
 black box testing, 159  
 Bloch, Joshua, xii  
 block, `try`, 113  
 Booth, Rick, 188  
 boundary condition testing, 140–141, 152, 159–160  
 Bourne, Steven R., 158  
 braces, position of {}, 10  
 Brooks, Frederick P., Jr., 61, 83, 87, 115  
`bsearch` library function, 36  
 B-tree, 54  
 buffer  
     flush, 107, 126  
     overflow error, 67, 156–157  
 buffering, I/O, 180  
 bug, *see also* error  
 bug  
     environment dependent, 131  
     header file, 129  
     *isprint*, 129, 136  
     list, 128  
     mental model, 127  
     non-reproducible, 130–131  
     performance, 18, 82, 175  
     reports, 136  
     test program, 129  
     typographical, 128  
**build** function  
     Markov C, 67  
     Markov C++, 77  
 byte order, 194, 204–207  
     diagram, 204  
`byteorder` program, 205

**C**

    function prototype, 191  
     standard, ANSI/ISO, 190, 212  
**C++**  
     inline function, 17, 19  
     *iostream* library, 77  
     *sort* function, 37  
     standard, ISO, 76, 190, 212  
     *string* class, 100  
     caching, 179, 186, 243  
     *can't get here* message, 124  
     *can't happen* message, 15, 142, 155

Cargill, Tom, xii  
 carriage return, \r, 89, 96, 203–204  
 cast, 35, 40, 43, 244  
 C/C++ preprocessor, *see* preprocessor directive  
 C/C++ data type sizes, 192, 216  
`cerr` error stream, 126  
`Chain` class, Markov Java, 72  
`Chain.add` function, Markov Java, 73  
`Chain.build` function, Markov Java, 73  
`Chain.generate` function, Markov Java, 74  
 character set, *see* encoding  
 character class metacharacter, [], 223, 228  
 characters  
   HTML, 31  
   non-printing, 132  
   unsigned, 57, 152, 193  
 check function, 125  
 Christiansen, Tom, 83  
`cin` input stream, 77  
 class  
   C++ string, 100  
   container, 71, 76  
   Csv, 100  
   Java Date, 172  
   Java DecimalFormat, 221  
   Java Hashtable, 71  
   Java Random, 39  
   Java StringTokenizer, 73  
   Java Vector, 71  
   Markov, 72  
     Markov Java Chain, 72  
     Markov Java Prefix, 72  
 Cleeland, Chris, xii  
 clock library function, 171  
 CLOCKS\_PER\_SEC timer resolution, 172  
`clone` method, *see* object copy  
`Cmp` interface, 38  
 code generation by macro, 240  
 Code structure, 234  
 code tuning, 176, 178–182  
 Cohen, Danny, 213  
 Coleridge, Samuel Taylor, 247  
 command  
   echo, 207  
   interpreter, 106, 228  
   status return, 109, 225  
   sum, 208  
   time, 171  
 comma-separated values, *see also* CSV  
 comma-separated values, 86–87  
 comments, 23–27, 203  
   semantic, 239  
 common subexpression elimination, 178  
 Comparable interface, 37  
 compatibility, backwards, 209, 211  
 compiler  
   gcc, 120  
   just-in-time, 81, 241, 243  
   optimization, 176, 186  
   testing, 147, 239  
 compiler-compiler  
   **bison**, 232  
   **yacc**, 232, 245  
 compile-time control flow, 199  
 complex expressions, 7  
 complexity, 40  
 conditional  
   compilation, 25, 199  
   operator, ?:, 8, 193  
 configuration script, 201  
 conservation properties, testing, 147, 161  
 consistency, 4, 11, 105  
 const declaration, 20  
 constant-time algorithm, 41, 44, 49, 55, 76  
 constructor, 100, 107–108  
   Markov Java Prefix, 74  
 container  
   class, 71, 76  
   deque, 76, 81  
   hash, 76, 81  
   list, 81  
   map, 72, 76, 81  
   pair, 112  
   vector, 76, 100  
 control flow, compile-time, 199  
 control-Z end of file, 134, 207  
 convention  
   \_\_ naming, 104  
   naming, 3–5, 104  
 conversion error, `printf`, 120  
 Cooper, Alan, 115  
 coordinate hashing, 57–58  
 copy, object, 67, 73, 107–108, 161  
 cost model, performance, 184  
 Coughran, Bill, xii  
 coverage, test, 148  
 Cox, Russ, xii  
 CPU pipeline, 179, 244  
 CRLF, 204  
 CSV  
   advquoted function, 97–98  
   csvfield function, 98  
   csvnfield function, 98  
   endofline function, 96  
   main function, 89, 98, 103  
   reset function, 96  
   split function, 97  
   field diagram, 95  
   format, 91, 93, 96  
   in C, 91–99  
   in C++, 99–103  
   prototype, 87–91  
   specification, 93  
   "csv.h" header, 94

`Csv::advplain` function, 102  
`Csv::advquoted` function, 102  
`Csv::endofline` function, 101  
`Csv::getfield` function, 102  
`Csv::getline` function, 100  
`Csv::getnfield` function, 102  
`Csv::split` function, 101  
`Csv` class, 100  
`csvfield` function, CSV, 98  
`csvgetline`  
  function, 95  
  prototype, 88  
  variables, 94  
`csvnfield` function, CSV, 98  
`ctime` library function, 25, 144  
`<ctype.h>` header, 18, 21, 129, 210  
cubic algorithm, 41  
cyclic redundancy check, 58

dangling `else`, *see if-else* ambiguity  
dangling pointer, 130  
data  
  exchange, 203–204, 216  
  structure diagram, Markov, 66  
  structure diagram, spam filter, 170  
  structure, trie, 171  
  type sizes, C/C++, 192, 216  
  type sizes, Java, 193  
`Date` class, Java, 172  
`Date.getTime` Java library function, 172  
`dbx` debugger, 122  
`0xDEADBEEF`, 159  
debuggers, 118–119  
debugging  
  code, 200, 202  
  `malloc`, 131  
  output, 123  
`DecimalFormat` class, Java, 221  
decisions, multi-way, 14  
declaration  
  `const`, 20  
  `enum`, 20  
  `final`, 21  
  Java `synchronized`, 108  
  loop variable, 12  
  `static`, 94  
  `typedef`, 76, 217  
deconstruction, 82, 114  
default parameters, 100  
defensive programming, 114, 142  
`#define`, *see also* macro, function macro  
`#define` preprocessor directive, 2, 20, 240  
`delitem` list function, 49  
`delname` function, 43  
deque container, 76, 81  
derived type, 38  
Descartes, René, 249

descriptive names, 3  
design tradeoffs, 90  
destructor, 108  
Dewdney, A. K., 84  
Dewhurst, Steve, xii  
diagram  
  binary search tree, 51  
  byte order, 204  
  CSV field, 95  
  hash table, 55  
  list, 45  
  Markov data structure, 66  
  Markov hash table, 66  
  packet format, 216  
  parse tree, 54, 232  
  quicksort, 33  
  spam filter data structure, 170  
Dijkstra, Edsger, 139  
directive, *see* preprocessor directive  
discrete cosine transform, 24  
divide and conquer, 52, 124  
division by zero, 141–142, 236, 241  
`divop` function, 236  
Dorward, Sean, 213  
`double` vs. `float`, 183  
doubly-linked list, 49, 81  
`do-while` loop, 13, 133, 225  
dynamic `printf` format, 68

eager evaluation, 181  
`echo` command, 207  
Edison, Thomas A., 117  
`#elif` preprocessor directive, 199  
elimination, common subexpression, 178  
ellipsis function parameter, ..., 109, 218  
Ellis, Bruce, 82  
`else if`, 14  
`emalloc` function, 46, 110  
`emit` function, 244  
empty string, 91, 100  
encapsulation, 104  
encoding  
  ASCII, 210  
  GIF, 184  
  ISO 10646, 31, 210  
  Latin-1, 210  
  MIME, 203  
  PPM, 184  
  Unicode, 31, 210, 228  
  UTF-8, 211, 213, 228

`#endif` preprocessor directive, 199  
end of file, control-Z, 134, 207  
`endofline` function, CSV, 96  
end of string metacharacter, \$, 222  
`enum` declaration, 20  
`enum.p1` Perl program, 239  
environment dependent bug, 131

EOF value, 194  
**eprintf** function, 49, 109  
 "eprintf.h" header, 110  
 eqn language, 229  
 errno variable, 112, 193  
 <errno.h> header, 109  
 error message, *see also eprintf, weprintf*  
 error
 

- binary search for, 124
- buffer overflow, 67, 156–157
- gets, 14, 156
- handling, 109
- hardware, 130
- memory allocation, 130
- message format, 114
- message, misleading, 134
- numeric patterns of, 124
- off-by-one, 13, 124, 141
- order of evaluation, 9, 193
- out of bounds, 153
- patterns, 120
- Pentium floating-point, 130
- printf conversion, 120
- qsort argument, 122
- recent change, 120
- recovery, 92, 109–113
- reproducible, 123
- return values, 91, 111, 141, 143
- scanf, 120
- status return, 109
- stream, cerr, 126
- stream stderr, 104, 126
- stream, System.err, 126
- subscript out of range, 14, 140, 157

 "errors.h" header, 238  
 estimation, performance, 184–187  
**estrndup** function, 110, 114  
**eval** function, 233–234, 236  
 evaluation
 

- eager, 181
- expression, 233
- lazy, 92, 99
- multiple, 18–19, 22
  - of macro argument, multiple, 18, 129

 examples, regular expression, 223, 230, 239  
 Excel spreadsheet, 97  
 exhaustive testing, 154  
 expected performance, 40  
 exponential algorithm, 41  
 expression, *see also* regular expression  
 expression
 

- evaluation, 233
- format, 7
- style, 6–8

 expressions
 

- complex, 7

 negated, 6, 8, 25  
 readability of, 6  
 extensions, printf, 216  
**falloc** symbol, 5  
 fall-through, switch, 16  
 far pointer, 192  
**fdopen** function, 134  
**fflush** library function, 126  
**fgets** library function, 22, 88, 92, 140, 156  
 Fielding, Raymond, 29  
 file, *see also* header  
 files
 

- binary, 132, 157, 203
- test data, 157

 final declaration, 21  
**find** library function, 30  
**find\_first\_of** library function, 101–102  
 Flandrena, Bob, xii, 188  
**float** vs. **double**, 183  
 floating-point
 

- arithmetic, IEEE, 112, 181, 193
- error, Pentium, 130

 flush, buffer, 107, 126  
**fmt** Awk program, 229  
**for** loop idioms, 12, 194  
 format
 

- CSV, 91, 93, 96
- dynamic printf, 68
- output, 89
- printf%.\*s, 133
- string, printf, 216

 Fraser, Chris, 245  
**freadd** library function, 106, 205  
 free list, 180  
**free**

- library function, 48
- multiple calls of, 131

**freeall** list function, 48  
 French, Renée, xii  
**freq** program, 147, 161  
 Friedl, Jeffrey, 246  
 Frost, Robert, 85  
**fscanf** library function, 67  
 function, *see also* library function  
 function macros, *see also* macros  
 function
 

- addend list, 46
- addfront list, 46
- addname list, 42
- addop, 233, 244
- alloca, 180
- apply list, 47
- applyinorder tree, 53
- applypostorder tree, 54
- avg, 141
- C++ inline, 17, 19

**C++ sort**, 37  
**check**, 125  
**CSV advquoted**, 97–98  
**CSV csvfield**, 98  
**CSV csvnfield**, 98  
**CSV endofline**, 96  
**CSV main**, 89, 98, 103  
**CSV reset**, 96  
**CSV split**, 97  
**Csv::advplain**, 102  
**Csv::advquoted**, 102  
**Csv::endofline**, 101  
**Csv::getfield**, 102  
**Csv::getline**, 100  
**csvgetline**, 95  
**Csv::getnfield**, 102  
**Csv::split**, 101  
**delitem list**, 49  
**delname**, 43  
**divop**, 236  
**malloc**, 46, 110  
**emit**, 244  
**fprintf**, 49, 109  
**estrdup**, 110, 114  
**eval**, 233–234, 236  
**fdopen**, 134  
**freeall list**, 48  
**generate**, 235  
**getbits**, 183  
**grep**, 226  
**grep main**, 225  
**Icmp Integer comparison**, 38  
**icmp integer comparison**, 36  
**inccounter list**, 48  
**insert tree**, 51  
**isspm**, 167, 169, 177  
**leftmost longest matchstar**, 227  
**lookup binary search**, 31, 36  
**lookup hash table**, 56  
**lookup list**, 47  
**lookup tree**, 52  
**macro, isoctal**, 5  
**macros**, 17–19  
**Markov C add**, 68  
**Markov C addsuffix**, 68  
**Markov C build**, 67  
**Markov C++ build**, 77  
**Markov C generate**, 70  
**Markov C++ generate**, 78  
**Markov C hash**, 66  
**Markov C lookup**, 67  
**Markov C main**, 71  
**Markov C++ main**, 77  
**Markov Java Chain.add**, 73  
**Markov Java Chain.build**, 73  
**Markov Java Chain.generate**, 74  
**Markov Java main**, 72  
**Markov Java Prefix.equals**, 75  
**Markov Java Prefix.hashCode**, 74  
**match**, 224  
**matchhere**, 224  
**matchstar**, 225  
**memset**, 152  
**names**, 4  
**newitem list**, 45  
**nrlookup tree**, 53  
**nvcmp name-value comparison**, 37  
**pack**, 218  
**pack\_type1**, 217, 219  
**parameter, ... ellipsis**, 109, 218  
**pointer**, 34, 47, 122, 220–221, 233, 236, 244  
**printnv list**, 47  
**progname**, 110  
**prototype, C**, 191  
**pushop**, 236  
**quicksort**, 33  
**Quicksort.rand**, 39  
**Quicksort.sort**, 39  
**Quicksort.swap**, 39  
**receive**, 221  
**Scmp String comparison**, 38  
**scmp string comparison**, 35  
**setprogname**, 110  
**strupd**, 14, 110, 196  
**strings**, 132  
**strings main**, 133  
**strstr**, 167  
**swap**, 33  
**testmalloc**, 158  
**unpack**, 219  
**unpack\_type2**, 220  
**unquote**, 88  
**usage**, 114  
**virtual**, 221  
**wprintf**, 52, 109, 197  
**wrapper**, 111  
**fwrite library function**, 106, 205  
**Gamma, Erich**, 84  
**garbage collection**, 48, 75, 108  
  **reference count**, 108  
**gcc compiler**, 120  
**generate function**, 235  
  **Markov C**, 70  
  **Markov C++**, 78  
**generic class**, *see* container class  
**getbits function**, 183  
**getchar**  
  **idioms**, 13, 194  
  **library function**, 13, 194  
**getquotes.tcl** Tcl program, 87  
**gets**  
  **error**, 14, 156  
  **library function**, 14, 156  
**getur1.tcl** Tcl program, 230  
**GIF encoding**, 184

global variable, 3, 24, 104, 122  
 Gosling, James, 83, 212  
*got here* message, 124  
 graph of  
   hash table chains, 126  
   hash table size, 174  
**grep**  
   function, 226  
   implementation, 225–227  
   **main** function, 225  
   options, 228  
   program, 223–226  
 Grosse, Eric, xii  
 growing  
   arrays, 41–44, 58, 92, 95, 97, 158  
   hash table, 58

Hanson, David, 115, 245  
 Harbison, Sam, 212  
 hardware error, 130  
**hash**  
   function, 55–57  
   function, Java, 57  
   function multiplier, 56–57  
   table, 55–58, 78, 169  
   table chains, graph of, 126  
   table diagram, 55  
   table function, *lookup*, 56  
   table, growing, 58  
   table insertion, 56  
   table, prefix, 64  
   table size, 56–57, 65  
   table size, graph of, 174  
   value, 55  
**hash**  
   container, 76, 81  
   function, Markov C, 66  
 hashing, coordinate, 57–58  
**Hashtable** class, Java, 71  
 header  
   `<assert.h>`, 142  
   `"csv.h"`, 94  
   `<cctype.h>`, 18, 21, 129, 210  
   `"eprintf.h"`, 110  
   `<errno.h>`, 109  
   `"errors.h"`, 238  
   `<stdarg.h>`, 109, 218  
   `<stddef.h>`, 192  
   `<stdio.h>`, 104, 196  
   `<stdlib.h>`, 198  
   `<time.h>`, 171  
 header file  
   bug, 129  
   organization, 94  
 Helm, Richard, 84  
 Hemingway, Ernest, 63  
 Hennessy, John, 188

Herron, Andrew, xii  
 hexadecimal output, 125  
 histogram, 126  
 Hoare, C. A. R., 32, 37  
 holes in structure, 195  
 Holzmann, Gerard, xii, 57, 59  
 homoiousian vs. homoousian, 228  
 hot spot, 130, 172–174  
 HTML, 86, 157, 215, 230, 237  
   characters, 31  
 HTTP, 89, 204

**Icmp Integer** comparison function, 38  
**icmp integer** comparison function, 36  
 idioms, 10–17  
   **for** loop, 12, 194  
   **getchar**, 13, 194  
   infinite loop, 12  
   list traversal, 12  
   loop, 12–13, 140  
   **malloc**, 14  
   **memmove** array update, 43, 68  
   **new**, 14  
   **realloc**, 43, 95  
   side effects, 195  
   string copy, 14  
   string truncation, 26  
   **switch**, 16  
   idle loop, 177  
 IEEE floating-point arithmetic, 112, 181, 193  
**#if** preprocessor directive, 196  
**#ifdef**, *see also* conditional compilation  
**#ifndef** preprocessor directive, 25, 196, 198–201  
**if-else** ambiguity, 10  
**inccounter** list function, 48  
 increment operator, `++`, 9  
 incremental testing, 145  
 indentation style, 6, 10, 12, 15  
 independent implementations, testing by, 148  
**indexOf** Java library function, 30  
 Inferno operating system, 181, 210, 213  
 infinite loop idioms, 12  
 information hiding, 92, 99, 104, 202  
   in C, 94, 103  
 initialization, static, 99, 106  
 inline function, C++, 17, 19  
 in-order tree traversal, 53  
 input  
   mode, `rb`, 134, 207  
   stream, `cin`, 77  
   stream, `stdin`, 104  
**insert** tree function, 51  
 insertion, hash table, 56  
 instructions, stack machine, 235  
 integer  
   comparison function, `icmp`, 36  
   overflow, 36, 157

interface  
**Cmp**, 38  
**Comparable**, 37  
principles, 91, 103–106  
**Serializable**, 207  
**interface**, Java, 38  
interfaces, user, 113–115  
internationalization, 209–211  
interpreter, 231, 234  
intersection, portability by, 198  
I/O  
binary mode, 134, 207  
buffering, 180  
text mode, 134  
**IOException**, 113  
**iostream** library, C++, 77  
**isalpha** library function, 210  
ISO  
10646 encoding, 31, 210  
C++ standard, 76, 190, 212  
**isocial** function macro, 5  
**isprint** bug, 129, 136  
**isspm** function, 167, 169, 177  
**isupper** library function, 18, 21  
**isUpperCase** Java library function, 21

**Java**  
**Array**, 39  
**Array length field**, 22  
data type sizes, 193  
**Date class**, 172  
**DecimalFormat class**, 221  
hash function, 57  
**Hashtable class**, 71  
interface, 38  
library function, **Date.getTime**, 172  
library function, **indexOf**, 30  
library function, **isUpperCase**, 21  
library function, **Math.abs**, 39  
logical right shift operator, **>>>**, 194  
**Object**, 38, 40, 71  
quicksort, 37–40  
**Random class**, 39  
random library function, 24, 162  
**StreamTokenizer class**, 73  
synchronized declaration, 108  
**Vector class**, 71  
Virtual Machine, 237  
**JavaScript**, 215  
**JIT**, *see* just-in-time compiler  
Johnson, Ralph, 84  
Joy, Bill, 212  
just-in-time compiler, 81, 241, 243

Kernighan, Brian, 28, 212, 245  
Kernighan, Mark, xii  
key, search, 36, 55, 77

Knuth, Donald, 59, 159, 162, 172, 188, 245  
Koenig, Andy, xii, 239

Lakos, John, xii, 115  
language  
**eqn**, 229  
lawyer, 191  
mainstream, 191  
standard, 190  
languages  
scripting, 80, 82, 230  
testing, 150  
Latin-1 encoding, 210  
lazy evaluation, 92, 99  
leap year computation, 7, 11, 144  
leftmost longest  
match, 226  
**matchstar** function, 227  
**length** field, Java **Array**, 22  
library  
C++ **iostream**, 77  
design, 91–94  
sort, 34–37  
library function  
**abort**, 125  
**atexit**, 107  
**bsearch**, 36  
**clock**, 171  
**ctime**, 25, 144  
**Date.getTime** Java, 172  
**fflush**, 126  
**fgets**, 22, 88, 92, 140, 156  
**find**, 30  
**find\_first\_of**, 101–102  
**fread**, 106, 205  
**free**, 48  
**fscanf**, 67  
**fwrite**, 106, 205  
**getchar**, 13, 194  
**gets**, 14, 156  
**indexOf** Java, 30  
**isalpha**, 210  
**isupper**, 18, 21  
**isUpperCase** Java, 21  
Java random, 24, 162  
**longjmp**, 113  
**malloc**, 14, 120, 131, 157  
**Math.abs** Java, 39  
**memcmp**, 173  
**memcpy**, 43, 105  
**memmove**, 43, 68, 105  
**memset**, 182  
**new**, 14, 120  
**qsort**, 34  
**rand**, 33, 70  
**realloc**, 43, 95, 120  
**scanf**, 9, 156, 183  
**setbuf**, **setvbuf**, 126

**setjmp**, 113  
**setmode**, 134  
**sprintf**, 67  
**strchr**, 30, 167  
**strcmp**, 26  
**strcpy**, 14  
**strcspn**, 97, 101, 155  
**strerror**, 109, 112  
**strlen**, 14  
**strncmp**, 167  
**strstr**, 30, 167  
**strtok**, 88, 96, 105, 108, 155  
**vfprintf**, 109  
Linderman, John, xii  
Lindholm, Tim, 245  
line continuation character, \, 240  
linear  
  algorithm, 30, 41, 46–47  
  search, 30, 32  
list  
  bug, 128  
  diagram, 45  
  doubly-linked, 49, 81  
  function, addend, 46  
  function, addfront, 46  
  function, addname, 42  
  function, apply, 47  
  function, delitem, 49  
  function, freeall, 48  
  function, inccounter, 48  
  function, lookup, 47  
  function, newitem, 45  
  function, printnv, 47  
  representation, 45–46, 49  
  singly-linked, 45  
  traversal idioms, 12  
**list** container, 81  
lists, 44–50  
literate programming, 240  
little languages, 151, 216, 229  
little-endian, 204  
local variable, 3, 122  
  pointer to, 130  
Locanthy, Bart, 241, 246  
log file, 111, 125, 131  
logical  
  operator, &&, 6, 193  
  operator, ||, 6, 193  
  right shift operator, >> Java, 194  
  shift, 135, 194  
log *n* algorithm, 32, 41, 51–52, 76  
**longjmp** library function, 113  
lookup  
  binary search function, 31, 36  
  function, Markov C, 67  
  hash table function, 56  
  list function, 47  
  tree function, 52  
loop  
  **do-while**, 13, 133, 225  
  elimination, 179  
  idioms, 12–13, 140  
  inversion, 169  
**LOOP** macro, 240  
loop  
  unrolling, 179  
  variable declaration, 12  
machine  
  stack, 234  
  virtual, 203, 213, 232, 236  
machine-dependent code, 181  
macro, 17–19  
  argument, multiple evaluation of, 18, 129  
  **assert**, 142  
  code generation by, 240  
  **LOOP**, 240  
  **NELEMS**, 22, 31  
  **va\_arg**, **va\_list**, **va\_start**, **va\_end**, 109, 218  
magic numbers, 2, 19–22, 129  
Maguire, Steve, 28, 137  
**main** function  
  CSV, 89, 98, 103  
  **grep**, 225  
  Markov C, 71  
  Markov C++, 77  
  Markov Java, 72  
  **strings**, 133  
mainstream language, 191  
**malloc**  
  debugging, 131  
  idioms, 14  
  library function, 14, 120, 131, 157  
management  
  memory, 48  
  resource, 92, 106–109  
**map** container, 72, 76, 81  
Markov  
  Awk program, 79  
  C add function, 68  
  C addsuffix function, 68  
  C build function, 67  
  C++ build function, 77  
  C generate function, 70  
  C++ generate function, 78  
  C hash function, 66  
  C lookup function, 67  
  C main function, 71  
  C++ main function, 77  
  chain algorithm, 62–63  
  data structure diagram, 66  
  hash table diagram, 66  
  Java Chain class, 72  
  Java Chain.add function, 73  
  Java Chain.build function, 73  
  Java Chain.generate function, 74

**Java** `main` function, 72  
`Java Prefix` class, 72  
`Java Prefix` constructor, 74  
`Java Prefix.equals` function, 75  
`Java Prefix.hashCode` function, 74  
 Perl program, 80  
 program testing, 160–162  
 run-time table, 81  
 state, 64  
`test` program, 161  
`Markov` class, 72  
 Mars Pathfinder, 121  
 Marx Brothers, 253  
`match`, leftmost longest, 226  
`match` function, 224  
`matchhere` function, 224  
`matchstar` function, 225  
 leftmost longest, 227  
`Math.abs` Java library function, 39  
 McConnell, Steve, 28, 115, 137  
 McIlroy, Doug, xii, 59  
 McNamee, Paul, xii  
 mechanization, 86, 146, 149, 155, 237–240  
`memcmp` library function, 173  
`memcpy` library function, 43, 105  
 Memishian, Peter, xii  
`memmove`  
     array update idioms, 43, 68  
     library function, 43, 68, 105  
`memory allocator`, *see* `malloc`, `new`  
`memory`  
     allocation, 48, 67, 92  
     allocation error, 130  
     leak, 107, 129, 131  
     management, 48  
`memset`  
     function, 152  
     library function, 182  
     test, 152–153  
`mental model bug`, 127  
`message`, *see also* `eprintf`, `wprintf`  
`message`  
     *can't get here*, 124  
     *can't happen*, 15, 142, 155  
`format`, error, 114  
     *got here*, 124  
`metacharacter`  
     . any character, 223  
     [] character class, 223, 228  
     \$ end of string, 222  
     + one or more, 223, 228  
     | OR, 223  
     \ quote, 223, 228  
     ^ start of string, 222  
     \* zero or more, 223, 225, 227  
     ? zero or one, 223, 228  
`metacharacters`  
     Perl, 231  
     regular expression, 222  
`MIME encoding`, 203  
 Minnie, A., 253  
`misleading error message`, 134  
 Mitchell, Don P., 82  
`Modula-3`, 237  
 Mullender, Sape, xii  
 Mullet, Kevin, 115  
`multiple`  
     assignment, 9  
     calls of `free`, 131  
     evaluation, 18–19, 22  
     evaluation of macro argument, 18, 129  
`multiplier`, hash function, 56–57  
`multi-threading`, 90, 108, 118  
`multi-way decisions`, 14  
`names`  
     descriptive, 3  
     function, 4  
     variable, 3–4, 155  
`Nameval structure`, 31, 42, 45, 50, 55  
`name-value structure`, *see* `Nameval structure`  
`name-value comparison function`, `nvcmp`, 37  
`naming convention`, 3–5, 104  
     \_\_, 104  
`NaN` not a number, 112  
`near pointer`, 192  
`negated expressions`, 6, 8, 25  
`NELEMS` macro, 22, 31  
 Nelson, Peter, xii  
 Nemeth, Evi, xii  
`new`  
     idioms, 14  
     library function, 14, 120  
`newitem` list function, 45  
`n log n` algorithm, 34, 41  
`non-printing characters`, 132  
`non-reproducible bug`, 130–131  
`NONWORD` value, 69  
`not a number`, `NaN`, 112  
`notation`  
     for zero, 21  
     `printf`-like, 87, 99, 217  
`nrlookup` tree function, 53  
`null byte`, '\0', 21  
`NULL` pointer, 21  
`null` reference, 21, 73  
`numbers`, magic, 2, 19–22, 129  
`numeric patterns of error`, 124  
`numerology`, 124  
`nvcmp` name-value comparison function, 37  
`NVtab` structure, 42  
`object copy`, 67, 73, 107–108, 161

**Object**, Java, 38, 40, 71  
 off-by-one error, 13, 124, 141  
 one or more metacharacter, +, 223, 228  
*O*-notation, *see also* algorithm  
*O*-notation, 40–41  
 table, 41  
 on-the-fly compiler, *see* just-in-time compiler  
 opaque type, 104  
 operating system  
   Inferno, 181, 210, 213  
   Plan 9, 206, 210, 213, 238  
   virtual, 202, 213  
 operator  
   & bitwise, 7, 127  
   && logical, 6, 193  
   ++ increment, 9  
   = assignment, 9, 13  
   >> right shift, 8, 135, 194  
   >>= assignment, 8  
   >>> Java logical right shift, 194  
   ?: conditional, 8, 193  
   | bitwise, 7, 127  
   || logical, 6, 193  
   bitblt, 241  
   function table, optab, 234  
   overloading, 100, 183  
   precedence, 6–7, 127  
   relational, 6, 127  
   sizeof, 22, 192, 195  
 optab operator function table, 234  
 optimization, compiler, 176, 186  
 options, grep, 228  
 OR metacharacter, |, 223  
 order of evaluation error, 9, 193  
 organization, header file, 94  
 out of bounds error, 153  
 output  
   debugging, 123  
   format, 89  
   hexadecimal, 125  
   stream, `stdout`, 104  
 overflow, integer, 36, 157  
 overloading, operator, 100, 183

pack function, 218  
 pack\_type1 function, 217, 219  
 packet format diagram, 216  
 pack, unpack, 216–221  
 pair container, 112  
 parameter, . . . ellipsis function, 109, 218  
 parameters, default, 100  
 parentheses, redundant, 6  
 parenthesization, 18  
   and ambiguity, 6  
 parse tree, 54, 232  
   diagram, 54, 232

parser generator, *see* compiler-compiler  
 pattern matching, *see* regular expression  
 patterns, error, 120  
 Patterson, David, 188  
 Pentium floating-point error, 130  
 performance  
   bug, 18, 82, 175  
   cost model, 184  
   estimation, 184–187  
   expected, 40  
   graph, 126, 174  
   test suite, 168  
   worst-case, 40

Perl  
   metacharacters, 231  
   program, enum.p1, 239  
   program, Markov, 80  
   program, unhtml.p1, 230  
   regular expression, 230  
   test suite, 162  
 picture, *see* diagram  
 Pike, Rob, 213, 245–246  
 pipeline, CPU, 179, 244  
 pivot element, quicksort, 32–34  
 Plan 9 operating system, 206, 210, 213, 238  
 Plauger, P. J., 28  
 pointer  
   dangling, 130  
   far, 192  
   function, 34, 47, 122, 220–221, 233, 236, 244  
   near, 192  
   NULL, 21  
   to local variable, 130  
   void\*, 21, 43, 47  
 portability, 189  
   by intersection, 198  
   by union, 198  
 position of {} braces, 10  
 POSIX standard, 198, 212  
 post-condition, 141  
 post-order tree traversal, 54, 232  
 PostScript, 203, 215, 237, 239  
 PPM encoding, 184  
*Practice of Programming* web page, xi  
 precedence, operator, 6–7, 127  
 pre-condition, 141  
**Prefix**  
   class, Markov Java, 72  
   constructor, Markov Java, 74  
 prefix hash table, 64  
**Prefix.equals** function, Markov Java, 75  
**Prefix.hashCode** function, Markov Java, 74  
 pre-order tree traversal, 54  
 preprocessor directive  
   `#define`, 2, 20, 240  
   `#elif`, 199  
   `#endif`, 199

**#if**, 196  
**#ifdef**, 25, 196, 198–201  
 Presotto, David, 213  
 principles, interface, 91, 103–106  
**printf**  
     conversion error, 120  
     extensions, 216  
     format, dynamic, 68  
     format string, 216  
     `%.*s` format, 133  
**printf-like notation**, 87, 99, 217  
**printv** list function, 47  
 production code, 83, 99  
**profile**  
     Awk, 174  
     spam filter, 173–174  
**profiling**, 167, 172–174  
**progrname** function, 110  
**program**  
     **byteorder**, 205  
     counter, 236, 243  
     **enum.p1** Perl, 239  
     **fmt** Awk, 229  
     **freq**, 147, 161  
     **getquotes.tcl** Tcl, 87  
     **geturl.tcl** Tcl, 230  
     **grep**, 223–226  
     inverse, 147  
     Markov Awk, 79  
     Markov Perl, 80  
     Markov test, 161  
     **sizeof**, 192  
     **split.awk** Awk, 229  
     **strings**, 131–134  
     **unhtml.p1** Perl, 230  
     **vis**, 134  
**programmable tools**, 228–231  
**programming, defensive**, 114, 142  
**protocol checker**, Supertrace, 57  
**prototype**  
     code, 83, 87  
     CSV, 87–91  
     **csvgetline**, 88  
**pushop** function, 236  
  
**qsort**  
     argument error, 122  
     library function, 34  
**quadratic algorithm**, 40, 43, 176  
**questionable code notation**, ?, 2, 88  
**quicksort**  
     algorithm, 32  
     analysis, 34  
     diagram, 33  
     Java, 37–40  
     pivot element, 32–34  
**quicksort** function, 33  
  
**Quicksort**.**rand** function, 39  
**Quicksort**.**sort** function, 39  
**Quicksort**.**swap** function, 39  
**quote metacharacter**, \, 223, 228  
**quotes, stock**, 86  
  
     \r carriage return, 89, 96, 203–204  
     Rabinowitz, Marty, xii  
     **rand** library function, 33, 70  
     **Random** class, Java, 39  
     random selection, *l/k*, 70  
     **random** library function, Java, 24, 162  
     rb input mode, 134, 207  
     readability of expressions, 6  
     **realloc**  
         idioms, 43, 95  
         library function, 43, 95, 120  
     **receive** function, 221  
     recent change error, 120  
     records, test, 151  
     recovery, error, 92, 109–113  
     reduction in strength, 178  
     redundant parentheses, 6  
     reentrant code, 108  
     reference  
         argument, 111, 220  
         **null**, 21, 73  
     reference count garbage collection, 108  
     regression testing, 149  
     regular expression, 99, 222–225, 239, 242  
         examples, 223, 230, 239  
         metacharacters, 222  
         Perl, 230  
         Tcl, 230  
     Reiser, John, 246  
     relational operator, 6, 127  
     representation  
         list, 45–46, 49  
         sparse matrix, 183  
         tree, 50  
         two's complement, 194  
     reproducible error, 123  
     **reset** function, CSV, 96  
     resource management, 92, 106–109  
     return, *see* carriage return  
     right shift  
         operator, **>>**, 8, 135, 194  
         operator, **>>>** Java logical, 194  
     Ritchie, Dennis, xii, 212–213  
  
     Sam text editor, 202, 213  
     Sano, Darrell, 115  
     **scanf**  
         error, 120  
         library function, 9, 156, 183  
     Schwartz, Randal, 83  
     **ScmpString** comparison function, 38

**scmp** string comparison function, 35  
**script**  
  configuration, 201  
  test, 149, 160  
**scripting languages**, 80, 82, 230  
**search**  
  algorithm, sequential, 30  
  key, 36, 55, 77  
**searching**, 30–32  
**Sedgewick, Robert**, 59  
**selection**,  $1/k$  random, 70  
**self-checking code**, 125  
**self-contained test**, 150  
**semantic comments**, 239  
**sentinel**, 30, 69–71  
**sequential search algorithm**, 30  
**Serializable interface**, 207  
**setbuf, setvbuf** library function, 126  
**setjmp** library function, 113  
**setmode** library function, 134  
**setprogname** function, 110  
**Shakespeare, William**, 165  
**Shaney, Mark V.**, xii, 84  
**shell**, *see* command interpreter  
**Shneiderman, Ben**, 115  
**side effects**, 8–9, 18, 193  
  idioms, 195  
**signals**, 197  
**single point of truth**, 238  
**singly-linked list**, 45  
**size, hash table**, 56–57, 65  
**size\_t type**, 192, 199  
**sizeof**  
  operator, 22, 192, 195  
  program, 192  
**sizes**  
  C/C++ data type, 192, 216  
  Java data type, 193  
**sort**  
  algorithm, tree, 53  
  library, 34–37  
**sort function, C++**, 37  
**sorting strings**, 35  
**source code control**, 121, 127  
**space efficiency**, 182–184  
**spam filter**, 166–170  
  data structure diagram, 170  
  profile, 173–174  
**sparse matrix representation**, 183  
**special-case tuning**, 181  
**special-purpose allocator**, 180, 182  
**specification**, 87, 93  
  CSV, 93  
**split function, CSV**, 97  
**split.awk** Awk program, 229  
**spreadsheet format**, *see also* comma-separated values  
**spreadsheet**, 139, 221  
  Excel, 97  
**sprintf** library function, 67  
**stack**  
  machine, 234  
  machine instructions, 235  
  trace, 118–119, 122  
**standard**  
  ANSI/ISO C, 190, 212  
  ISO C++, 76, 190, 212  
  language, 190  
  POSIX, 198, 212  
**Standard Template Library**, *see* STL  
**start of string metacharacter, ^**, 222  
**state, Markov**, 64  
**State structure**, 65  
**static initialization**, 99, 106  
**static**  
  array, 131  
  declaration, 94  
**statistical test**, 161  
**status return**  
  command, 109, 225  
  error, 109  
**<stdarg.h>** header, 109, 218  
**<stddef.h>** header, 192  
**stderr** error stream, 104, 126  
**stdin** input stream, 104  
**<stdio.h>** header, 104, 196  
**<stdlib.h>** header, 198  
**stdout** output stream, 104  
**Steele, Guy**, 212  
**Stevens, Rich**, xii, 212  
**STL**, 49, 76, 104, 155, 192  
**stock quotes**, 86  
**Strachey, Giles Lytton**, 215  
**strchr** library function, 30, 167  
**strcmp** library function, 26  
**strcpy** library function, 14  
**strcspn** library function, 97, 101, 155  
**strupr** function, 14, 110, 196  
**StreamTokenizer class, Java**, 73  
**strerror** library function, 109, 112  
**stress testing**, 155–159, 227  
**string copy idioms**, *see also* **strupr**  
**string**  
  comparison function, **scmp**, 35  
  copy idioms, 14  
  truncation idioms, 26  
**string class, C++**, 100  
**strings**  
  function, 132  
  main function, 133  
  program, 131–134  
**strlen** library function, 14  
**strncmp** library function, 167  
**Stroustrup, Bjarne**, xii, 83

**strstr**  
     function, 167  
     implementation, 167–168  
     library function, 30, 167  
**strtok** library function, 88, 96, 105, 108, 155  
**structure**  
     Code, 234  
     holes in, 195  
     member alignment, 195  
     Nameval, 31, 42, 45, 50, 55  
     NVtab, 42  
     State, 65  
     Suffix, 66  
     Symbol, 232  
     Tree, 233  
 Strunk, William, 1, 28  
**style**  
     expression, 6–8  
     indentation, 6, 10, 12, 15  
 subscript out of range error, 14, 140, 157  
 suffix, 62  
 Suffix structure, 66  
 sum command, 208  
 Supertrace protocol checker, 57  
 swap function, 33  
 Swift, Jonathan, 213  
**switch**  
     fall-through, 16  
     idioms, 16  
 Symbol structure, 232  
 symbol table, 55, 58  
 synchronized declaration, Java, 108  
 syntax tree, *see* parse tree  
 System.err error stream, 126  
 Szymanski, Tom, xii

**table**  
     Markov run-time, 81  
     O-notation, 41  
     optab operator function, 234  
 tail recursion, 53  
 Taylor, Ian Lance, xii  
 Tcl  
     program, getquotes.tcl, 87  
     program, geturl.tcl, 230  
     regular expression, 230  
     teddy bear, 123, 137  
 test  
     Awk, 150  
     beta release, 160  
     coverage, 148  
     data files, 157  
     memset, 152–153  
     program bug, 129  
     records, 151  
     scaffold, 89, 98, 146, 149, 151–155  
     script, 149, 160

self-contained, 150  
 statistical, 161  
 suite, performance, 168  
 suite, Perl, 162  
**test** program, Markov, 161  
**testing**  
     binary search, 146  
     black box, 159  
     boundary condition, 140–141, 152, 159–160  
     by independent implementations, 148  
     compiler, 147, 239  
     conservation properties, 147, 161  
     exhaustive, 154  
     incremental, 145  
     languages, 150  
     Markov program, 160–162  
     regression, 149  
     stress, 155–159, 227  
     tools, 147, 149  
     white box, 159  
**testmalloc** function, 158  
 text mode I/O, 134  
 Thimbleby, Harold, 115  
 Thompson, Ken, xii, 188, 213, 242, 246  
 threaded code, 234  
**time** command, 171  
<time.h> header, 171  
 timer resolution, CLOCKS\_PER\_SEC, 172  
**tools**  
     programmable, 228–231  
     testing, 147, 149  
 Toyama, Kentaro, xii  
**tradeoffs**, design, 90  
 Traveling Salesman Problem, 41  
**tree**, 50–54, 231–237  
     balanced, 52, 76  
     binary search, 50  
     function, applyinorder, 53  
     function, applypostorder, 54  
     function, insert, 51  
     function, lookup, 52  
     function, nrlookup, 53  
     parse, 54, 232  
     representation, 50  
     sort algorithm, 53  
**Tree** structure, 233  
**tree traversal**  
     in-order, 53  
     post-order, 54, 232  
     pre-order, 54  
 Trickey, Howard, xii, 213  
**trie** data structure, 171  
 TRIP test for TEX, 159, 162  
**try** block, 113  
**tuning**  
     code, 176, 178–182  
     special-case, 181

- tuple, 112
- two's complement representation, 194
- type
  - derived, 38
  - opaque, 104
  - `size_t`, 192, 199
- `typedef` declaration, 76, 217
- typographical bug, 128
  
- `unhtml.p1` Perl program, 230
- Unicode encoding, 31, 210, 228
- uninitialized variables, 120, 159
- union, portability by, 198
- unpack function, 219
- unpack\_type2 function, 220
- unquote function, 88
- unsigned characters, 57, 152, 193
- usage function, 114
- user interfaces, 113–115
- USS Yorktown*, 142
- UTF-8 encoding, 211, 213, 228
- `uencode`, 203
  
- `va_arg`, `va_list`, `va_start`, `va_end` macro, 109, 218
- values, error return, 91, 111, 141, 143
- van der Linden, Peter, 28
- Van Wyk, Chris, xii
- variable
  - `errno`, 112, 193
  - global, 3, 24, 104, 122
  - local, 3, 122
  - names, 3–4, 155
- variables
  - `csvgetline`, 94
  - uninitialized, 120, 159
- Vector class, Java, 71
- vector container, 76, 100
- Venturi, Robert, 189
- `vfprintf` library function, 109
  
- virtual
  - function, 221
  - machine, 203, 213, 232, 236
  - operating system, 202, 213
- `vis` program, 134
- Visual Basic, 215, 237
- Vlissides, John, 84
- `void*` pointer, 21, 43, 47
  
- Wadler, Phil, xii
- Wait, John W., xii
- Wall, Larry, 83
- Wang, Daniel C., xii
- warning message, *see weprintf*
- web
  - browser, 86, 231
  - page, *Practice of Programming*, xi
- Weinberger, Peter, xii
- `weprintf` function, 52, 109, 197
- white box testing, 159
- White, E. B., 1, 28
- wide characters, 211
- Wiener, Norbert, 139
- wildcards, \*, 106, 222
- Winterbottom, Philip, 213
- worst-case performance, 40
- wrapper function, 111
- Wright, Margaret, xii
  
- X Window system, 202, 206
  
- yacc compiler-compiler, 232, 245
- Year 2000 problem, 144, 182
- Yellin, Frank, 245
- Yorktown*, 142
- Young, Cliff, xii
  
- zero, 21
  - division by, 141–142, 236, 241
  - notation for, 21
- zero or more metacharacter, \*, 223, 225, 227
- zero or one metacharacter, ?, 223, 228