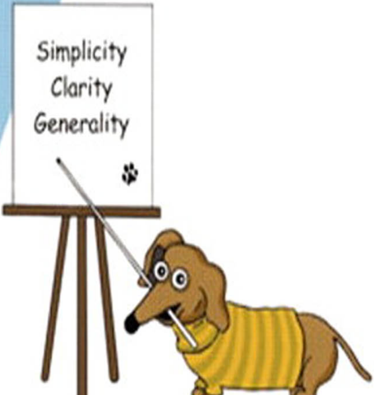
The Practice of Programming

Brian W. Kernighan
Rob Pike



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

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

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

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

Visit AW on the Web: www.awprofessional.com

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

Library of Congress Cataloging-in-Publication Data

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

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

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  afqnzgxl  lhidlwcd rjdjuvpydrlwnjy
```

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:	*Suffix words that follow:*
`Show your`	`flowcharts tables`
`your flowcharts`	`and will`
`flowcharts and`	`conceal`
`flowcharts will`	`be`
`your tables`	`and and`
`will be`	`mystified. obvious.`
`be mystified.`	`Show`
`be obvious.`	(*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.

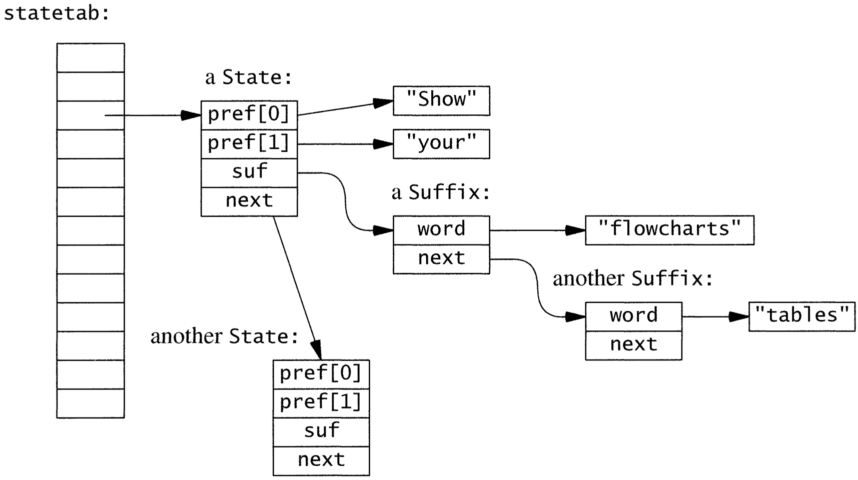
The prefix can be stored as an array of words. The elements of the hash table will be represented as a `State` data type, associating the `Suffix` list with the prefix:

```
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:



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 *) emalloc(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 `NONWORD`s 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 `NONWORD`s 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` `NONWORD`s. 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 R10000	400MHz Pentium II	Lines of 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).

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*