A SOFTWARE TOOLS SAMPLER

Prentice-Hall International Editions

WEBB MILLER

A SOFTWARE TOOLS SAMPLER

WEBB MILLER
Department of Computer Science
The Pennsylvania State University
University Park, Pennsylvania

Prentice-Hall International, Inc.

Library of Congress Cataloging-in-Publication Data

MILLER, WEBB
A software tools sampler.

Bibliography: p. 340
Includes index.
1. Computer software. 2. C (Computer program language) 3. UNIX (Computer operating system)
I. Title.
QA76.754.M55 1987 005.36'9 86-30234
ISBN 0-13-821984-2

Editorial/production supervision
and interior design: *Theresa A. Soler*
Manufacturing buyer: *Ed O'Dougherty*

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-821984-2 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall, Inc., *Englewood Cliffs, New Jersey*

A SOFTWARE TOOLS SAMPLER

PRENTICE-HALL SOFTWARE SERIES

Brian W. Kernighan, Advisor

PREFACE

This book contains a small ensemble of useful and interesting *software tools*—programs that help you prepare documents and programs on a computer. Each tool's capability and construction are discussed in detail and enhancements are outlined. After reading Chapter 1 (at most an hour or two if you already know the C programming language) you can go directly to any chapter of interest.

You should get copies of the programs, experiment with them, and change them to suit your needs. All programs listed in this book are available for a nominal charge. For information write:

James F. Fegen, Jr.
Executive Editor
Technical+Reference Division
Prentice-Hall, Inc.
Englewood Cliffs, N.J. 07632

The book's prerequisites are

- Programming experience and a familiarity with systematic methods for program development, such as *top-down design*.
- Experience with data structures equivalent to an undergraduate course on the subject. The terms *pointer, hashing, binary search*, and *dynamic storage allocation* should be completely familiar to you.
- Knowledge of, or willingness to learn the C programming language.

While the tools provide capabilities available from the UNIX† operating system, the code is new and UNIX is mentioned only superficially.

The book is a text on software tools. Initial versions were written at the University of Arizona, where the tools course is the first of three upper-division undergraduate classes covering computer system software. The other two classes treat programming systems (compilers, linkers, and debuggers) and operating systems. One of the purposes of this book is to teach about a major category of system software.

The tools course has an additional distinctive goal. It provides many students with their main exposure to complete and realistic software. Earlier courses exhibit only programs that can be built in a day (or an hour) and later ones often construct only toys for programming projects.

Besides use for a software tools class, this book might serve as a building block for a software engineering course. A text such as *Principles of Software Engineering and Design* by Marvin Zelkowitz, Alan Shaw, and John Gannon (Prentice-Hall, 1979) could introduce general software engineering principles, with examples and programming projects drawn from this book.

A third use is for self-study by a well-prepared and dedicated reader. Such a reader might want to turn a non-UNIX system into a more pleasant and productive place to work or might just be curious to see how these software tools can be built.

I have followed in the footsteps of the book *Software Tools* by Brian Kernighan and P. J. Plauger (Addison-Wesley, 1976), which was used for years in the tools class at the University of Arizona. Progress in computer science and improved preparation of the entering students led me to cover substantially more complex tools and to use a different programming language. The resulting class notes became this book.

My sincerest thanks go to Dave Hanson, Gene Myers, and Titus Purdin for reading drafts of this book and offering countless suggestions.

Webb Miller

†UNIX is a trademark of Bell Laboratories.

CONTENTS

1

INTRODUCTION

Some basic material should be mastered before studying programs in later chapters. Accomplished C programmers can extract the necessary information from this chapter in one or two hours. Others should allot substantially more time and be prepared to consult other sources.

Reading the later programs requires a knowledge of C. This book does not provide a complete language description; for that, you need a book on C. *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978) is an authoritative introduction, and *A C Reference Manual* by Samuel P. Harbison and Guy L. Steele, Jr. (Prentice-Hall, 1984) is an excellent resource for experienced C programmers.

Section 1.1 is essentially an ''entrance examination'' on C: when you understand the programs given there, read the rest of the book. Of course, newcomers to C will become fluent in the language only after completing several programming assignments from Chapters 2 to 5. Even C experts should look at Section 1.1, since it outlines the book's basic assumptions about the C programming environment.

The remainder of this chapter covers two C programs that lie midway, in terms of size and complexity, between the trivial programs of Section 1.1 and the programs in Chapters 2 to 5. The main goal is insight into the large-scale structure of the later programs. Readers unfamiliar with C may benefit from studying the code in detail.

The programs of Sections 1.2 and 1.3 illustrate the importance of the ''decision hiding principle'': *a program's structure should confine the effects of each implementation decision to a small, easily identified section of code.* Software conforming to this principle is easy to comprehend (and, hence, comparatively easy to get working) because relatively few implementation decisions must be grasped to understand a given module. Moreover, such software is easy to modify, since revising an implementation decision invalidates a minimal amount of code. Indeed, the implementation decisions that seem most likely to be changed later should be hidden with particular care.

Section 1.2 introduces *abstract data types*, an especially useful application of the decision hiding principle. The general idea is to keep the bulk of the program from directly manipulating an important data structure; instead, data access is restricted to a few tightly-specified "access functions". Details of the specific data structures implementing the access functions are hidden from the rest of the program, and the data structures are easily changed.

Abstract data types are the key to understanding much of the large-scale structure of later programs. Programs are often divided into manageable pieces by encapsulating each main data structure in a distinct module, then treating those modules as abstract data types. Typically, the remainder of the code can be modularized according to relationship to the data modules. For example, the code that moves data from an input file to data module *A* becomes module *X*, the procedures that access data module *A* and build data module *B* constitute module *Y*, and so on.

Section 1.3 discusses decision hiding for system dependencies in programs. Not all programs in this book are portable; some must be changed before they will run under another operating system or on a different machine. To minimize the work required to move a program, the nonportable code has been isolated.

1.1 GETTING STARTED

The short programs of this section provide a natural introduction to software tools. The first group of procedures is used throughout the book. The remaining programs are complete and useful software tools.

1.1.1 Basic UNIX Command Syntax

In this section, and at isolated points in the remainder of the book, use of a software tool is illustrated with the UNIX command syntax. Other command languages would have worked as well; the only purpose is to give a concrete idea of what it feels like to use the tools. The few properties of UNIX needed for these examples are summarized below. The paper "The UNIX programming environment" by Brian Kernighan and John Mashey (*IEEE Computer* magazine, April 1981, pp. 12–24) is a good source for learning more.

Under UNIX, the user can organize files into arbitrary groupings called *directories*. For example, the source files, object files, and executable file for a program are often grouped into their own directory.

UNIX programs are run by typing a line that contains the program name, perhaps followed by a list of arguments that are separated by blanks. Arguments often consist of file names or ''flags'' that select options. By convention, a leading minus (−) character distinguishes a flag from a file name. For example, the command

```
cc -O thud.c
```

applies the C compiler *cc* to the C source file *thud.c*, with the −*O* flag requesting optimized object code. Another UNIX convention is that files containing C source code have names ending with the two characters ''*.c*''.

The UNIX command interpreter, called the *shell*, provides a shorthand notation for specifying lists of file names. Specifically, in a command like

```
cc *.c
```

the string ''**.c*'' is replaced by the list of file names in the current directory that end in ''*.c*'', i.e., all C source files. Thus, if the current directory consists of the files *foo.c*, *thud.c*, and *prog.docum*, the command is equivalent to

```
cc foo.c thud.c
```

A second useful service of the UNIX shell is connecting the output of one command to the input of another. For example, *ls* is the command that lists the names of files in the current directory, and *lc* (pp. 12–14) counts its input lines. The UNIX command

```
ls | lc
```

connects the output of *ls* to the input of *lc*, creating a command that counts the number of files in the current directory (assuming that *ls* lists files one per line). Two commands can be connected this way if the first writes standard output and the second reads standard input. (The terms *standard output* and *standard input* are discussed below.) A *pipeline* is a chain of simpler commands linked in this manner by the shell.

1.1.2 Required Functions and Macros

Four classes of functions are assumed available. They are listed here for quick reference, then discussed more thoroughly when first used. The Appendix contains complete details.

Standard I/O Library. C statements for input or output are provided by a ''standard I/O library.'' Any source file using this library of functions should have the line

```
#include <stdio.h>
```

(or an *#include* line naming a file containing that line) near the beginning. The library provides the following functions and macros, which the book's programs use for input and output. (The only exception is the *fastfind* program of Section 4.1, which uses system-specific input procedures.)

fopen(), fclose(), fflush()	open, close, or flush an I/O stream
getc(), getchar()	get an input character
gets(), fgets()	get a string of input characters
printf(), fprintf(), sprintf()	formatted output conversion
putc(), putchar()	output a character
puts(), fputs()	output a string of characters
rewind()	return to the beginning of a file

As part of the I/O facilities, the following macros are defined by the *stdio.h* header file.

EOF	an integer returned upon end of file
FILE	the ''type'' associated with a file
NULL	the null pointer (can point to a character or a FILE)
stderr	FILE pointer for standard error file
stdin	FILE pointer for standard input file
stdout	FILE pointer for standard output file

All six macros are predefined constants; don't try to assign values to them.

Standard String Functions. The following functions manipulate character strings that are terminated by a null character ('*\0*'). *Strcat()* and *strcpy()*, the two that create a string, terminate the new string with a null character, but do not check for overflow of the new string. In some C implementations, *index()* is called *strchr()*.

index(s, c)	return the first location of the character *c* in *s*
strcat(s, t)	append a copy of *t* to the end of *s*
strcmp(s, t)	return 0 if and only if *s* equals *t*
strcpy(s, t)	copy *t* to *s* and return *s*
strlen(s)	return the length of *s*

Character-Classification Macros. Files containing the line

```
#include <ctype.h>
```

can thereafter employ character-testing macros from the list:

isalnum(c)	*c* is one of 'a'–'z', 'A'–'Z', or '0'–'9'
isalpha(c)	*c* is one of 'a'–'z' or 'A'–'Z'
isdigit(c)	*c* is one of '0'–'9'
islower(c)	*c* is one of 'a'–'z'
isprint(c)	*c* is a printing character (not a control character)
isspace(c)	*c* is a space, tab, or newline character
isupper(c)	*c* is one of 'A'–'Z'

For example, the condition

```
if (isspace(c)) ...
```

tests whether *c* is a "whitespace" character.

System-Specific Functions. The following machine-dependent functions are used; others are introduced in Chapters 2 and 5, as needed.

exit(n)	terminate execution, signaling *n* to the parent process
free(p)	free the memory allocated when *malloc()* returned *p*
malloc(n)	allocate *n* bytes and return the address (*NULL* signals failure)

1.1.3 Lib.c—A Library of C Procedures

Let's begin our quick tour of C programs with seven procedures that are used throughout the book.

Savename(). When commands can be combined in pipelines, it is desirable to know which of the constituent commands produced an error message. For example, if the pipeline

```
ls | find | lc
```

produces the message

```
Missing argument.
```

it is unclear which of the commands *ls*, *find*, or *lc* was incorrectly specified; the response

```
find: Missing argument.
```

is more informative.

Most of the programs in this book begin execution with a call like:

```
savename("find");
```

Any subsequent fatal error message will be preceded by the program's name, a colon (:), and a space.

```
#define MAX_NAME 50   /* maximum length of program or file name */
static char prog_name[MAX_NAME+1];   /* used in error messages */
/* savename - record a program name for error messages */
savename(name)
char *name;
{
	char *strcpy();

	if (strlen(name) <= MAX_NAME)
		strcpy(prog_name, name);
}
```

Savename() invokes the standard string function *strlen()* to count the characters in *name*. Another standard string function, *strcpy()*, copies *name* to the array *prog_name[]*, where it can be accessed by procedures in *lib.c* (the file containing *savename()*). *Name* is not copied if it is too long; *prog_name[]* can hold a 50-character name plus the ''null character'' that C uses to mark the end of a string.

Strcpy() is declared to be a function returning a character pointer, even though the returned value is unused; some C compilers demand that the declaration be present. C rules imply that *strlen()* returns an *int* since no declaration states otherwise.

Fatal(). *Fatal()* is used to terminate execution because of an error condition.

```
/* fatal - print message and die */
fatal(msg)
char *msg;
{
	if (prog_name[0] != '\0')
		fprintf(stderr, "%s: ", prog_name);
	fprintf(stderr, "%s\n", msg);
	exit(1);
}
```

Fatal() appends a newline character to the message it is given, then calls the system-specific function *exit()* to terminate execution and make its argument (1 to signal an error) available to the outside world. If *savename()* has deposited the program's name in *prog_name[]*, then the name, a colon, and a space are printed before the message. On the other hand, if *savename()* has not been called, then *progname[0]* is '\0' (because global character arrays are automatically initialized with null characters), so no program name is printed.

Fatalf(). *Fatalf()* works like *fatal()* except that the *msg* string can contain a conversion specification, like *%s*.

```
/* fatalf - format message, print it, and die */
fatalf(msg, val)
char *msg, *val;
{
	if (prog_name[0] != '\0')
		fprintf(stderr, "%s: ", prog_name);
	fprintf(stderr, msg, val);
	putc('\n', stderr);
	exit(1);
}
```

A typical use of *fatalf()* is:

```
char *name;
...
fatalf("Cannot open %s.", name);
```

which prints a final message of the form

```
Cannot open thud.c.
```

Although it violates programming etiquette and draws warnings from interprocedural analyzers like the UNIX *lint* program, programs in this book occasionally make calls such as

```
int k;
...
fatalf("Improper line number: %d.", k);
```

I don't know of any systems where the inconsistently typed second argument causes *fatalf()* to perform improperly. Of course, it is essential that the conversion specification match the second argument; for example,

```
int k;
...
fatalf("Improper line number: %s.", k);
```

won't work.

Ckopen(). Sometimes there is no graceful way to recover from an unsuccessful attempt to open a file. When the best contingency plan is to print a diagnostic message and terminate execution, programs can call *ckopen()*.

```
/* ckopen - open file; check for success */
FILE *ckopen(name, mode)
char *name, *mode;
{
	FILE *fopen(), *fp;

	if ((fp = fopen(name, mode)) == NULL)
		fatalf("Cannot open %s.", name);
	return(fp);
}
```

Ckopen() needs both the name of the file and a *mode* telling the intended use of the file. For example, setting *mode* to "*w*" (the string, not a single character '*w*') informs the operating system that you want to write to the file. *Ckopen()* employs the local *FILE* pointer variable *fp* and invokes the standard I/O function *fopen()*, which returns a *FILE* pointer. The test

```
if ((fp = fopen(name, mode)) == NULL)
```

calls *fopen()* with *ckopen()*'s arguments, assigns the returned *FILE* pointer to *fp*, then compares it with the *NULL* pointer. Unless *fopen()* signals failure by returning *NULL*, the *FILE* pointer is returned to the calling procedure. If *fopen()* fails, then *ckopen()* calls *fatalf()* with a diagnostic message.

Ckalloc(). A program can ask the operating system for a specified number of bytes of storage by calling *ckalloc()*.

```
/* ckalloc - allocate space; check for success */
char *ckalloc(amount)
int amount;
{
	char *malloc(), *p;

	if ((p = malloc( (unsigned) amount)) == NULL)
		fatal("Ran out of memory.");
	return(p);
}
```

Ckalloc() calls the system-specific function *malloc()* to provide the storage. If *malloc()* indicates failure by returning *NULL*, then *ckalloc()* calls *fatal()* to terminate execution. Otherwise, *malloc()* returns a pointer to a free block of memory, and *ckalloc()* hands that pointer back to the calling procedure.