

SECOND EDITION



LINUX

Application Development by Example

The Fundamental APIs



ARNOLD ROBBINS

FREE SAMPLE CHAPTER |



Praise for the Second Edition

“This may be the most clearly written narrative-style coverage of the topic that I have encountered and one of the most complete that is not solely a reference work.

“[The book] has forced me to think differently about some of these topics and how they build on one another. It’s almost like when I studied LISP way back when and it turned all my thinking about programming on its head. Thank you! It is a privilege to review this.

“This is one of the better-crafted tech books I’ve reviewed across two decades. The exercises consistently help the reader focus on specific things in the text and gently struggle and learn concepts that might otherwise slip through the cracks. I failed to find any exercise that I would delete and can’t think of any I would add.”

— *Matthew Helmke, Linux author and consultant*

Praise for the First Edition

“This is an excellent introduction to Linux programming. The topics are well chosen and lucidly presented. I learned things myself, especially about internationalization, and I’ve been at this for quite a while.”

— *Chet Ramey, coauthor and maintainer of the Bash shell*

“This is a good introduction to Linux programming. Arnold’s technique of showing how experienced programmers use the Linux programming interfaces is a nice touch, much more useful than the canned programming examples found in most books.”

— *Ulrich Drepper, project lead, GNU C library*

“A gentle yet thorough introduction to the art of UNIX system programming, *Linux Programming by Example* uses code from a wide range of familiar programs to illustrate each concept it teaches. Readers will enjoy an interesting mix of in-depth API descriptions and portability guidelines, and will come away well prepared to begin reading and writing systems applications. *Heartily recommended.*”

— *Jim Meyering, coauthor and maintainer of the GNU Core Utility Programs*

This page intentionally left blank

Linux Application Development by Example

The Fundamental APIs

Second Edition

Arnold Robbins

◆ Addison-Wesley
Hoboken, New Jersey

Cover image: JIN KANSA/stock.adobe.com

Figure 17.1: Ubuntu

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 the publisher was aware of a trademark claim, the designations have been printed with 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 programs and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Portions of Chapter 1, Copyright © 1994 Arnold David Robbins, first appeared in an article in Issue 16 of *Linux Journal*, reprinted by permission.

Portions of the documentation for Valgrind, Copyright © 2003 Julian Seward, reprinted by permission.

The GNU programs in this book are Copyright © 1985-2024, Free Software Foundation, Inc.. The full list of files and copyright dates is provided in the Preface. Each program is “free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.” Appendix C of this book provides the text of the GNU General Public License.

All V7 Unix code and documentation are Copyright © Caldera International Inc. 2001-2002. All rights reserved. They are reprinted here under the terms of the Caldera Ancient UNIX License, which is reproduced in full in Appendix B. Code from the One True Awk is Copyright © Lucent Technologies, 1997. The license for it is reproduced in Appendix D. Code from 4.4 BSD is Copyright © the Regents of the University of California, 1985, 1989, 1991, 1993. The license for it is reproduced in Appendix E.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, MS, and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation in the United States and other countries. Linux is a registered trademark of Linux Torvalds.

The example code written by Arnold Robbins for this book is covered by the following BSD 2 clause License:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Please contact us with concerns about any potential bias at www.pearson.com/en-us/report-bias.html.

Visit us on the Web: informit.com

Library of Congress Control Number: 2025944250

Copyright © 2004, 2026 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/en-us/global-permission-granting.html.

ISBN-13: 978-0-13-532552-0

ISBN-10: 0-13-532552-8

\$PrintCode

Contents

Foreword	xv
Preface	xvii
PART I Files and Users	1
Chapter 1 Introduction	3
1.1 The Linux/Unix File Model	3
1.1.1 Files and Permissions	3
1.1.2 Directories and File Names	5
1.1.3 Executable Files	6
1.1.4 Devices	7
1.2 The Linux/Unix Process Model	7
1.2.1 Pipes: Hooking Processes Together	9
1.3 Standard C versus Original C	9
1.4 Why GNU Programs Are Better	13
1.4.1 Program Design	13
1.4.2 Program Behavior	14
1.4.3 C Code Programming	14
1.4.4 Things That Make a GNU Program Better	15
1.4.5 Parting Thoughts about the <i>GNU Coding Standards</i>	17
1.5 Portability Revisited	17
1.5.1 Why Be Portable?	17
1.5.2 Portability Recommendations	17
1.6 Some Words about Coding Style	18
1.7 Artificial Intelligence Isn't Intelligent	19
1.8 Suggested Reading	19
1.9 Summary	20
Exercises	21
Chapter 2 Arguments, Options, and the Environment	23
2.1 Option and Argument Conventions	23
2.1.1 POSIX Conventions	24
2.1.2 GNU Long Options	26

2.2	Basic Command-Line Processing	26
2.2.1	The V7 echo Program	27
2.3	Option Parsing: <code>getopt()</code> and <code>getopt_long()</code>	28
2.3.1	Single-Letter Options	29
2.3.2	GNU <code>getopt()</code> and Option Ordering	31
2.3.3	Long Options	32
2.4	The Environment	37
2.4.1	Environment Management Functions	38
2.4.2	The Entire Environment: <code>environ</code>	39
2.4.3	GNU <code>env</code>	40
2.5	Summary	49
	Exercises	49
Chapter 3 User-Level Memory Management		51
3.1	Linux/Unix Address Space	51
3.2	Memory Allocation	55
3.2.1	Library Calls: <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code>	55
3.2.2	String Copying: <code>strdup()</code>	71
3.2.3	System Calls: <code>brk()</code> and <code>sbrk()</code>	72
3.2.4	Lazy Programmer Calls: <code>alloca()</code>	73
3.2.5	Address Space Examination	74
3.3	Summary	76
	Exercises	77
Chapter 4 Files and File I/O		79
4.1	Introducing the Linux/Unix I/O Model	79
4.2	Presenting a Basic Program Structure	79
4.3	Determining What Went Wrong	81
4.3.1	Values for <code>errno</code>	82
4.3.2	Error Message Style	86
4.4	Doing Input and Output	87
4.4.1	Understanding File Descriptors	87
4.4.2	Opening and Closing Files	88
4.4.3	Reading and Writing	91
4.4.4	Example: Unix <code>cat</code>	94
4.5	Random Access: Moving Around within a File	96
4.6	Creating Files	101
4.6.1	Specifying Initial File Permissions	101
4.6.2	Controlling Default Permissions with <code>umask()</code>	102
4.6.3	Creating Files with <code>creat()</code>	103
4.6.4	Revisiting <code>open()</code>	104
4.7	Forcing Data to Disk	106
4.8	Setting File Length	107

4.9	Summary	108
	Exercises	109
Chapter 5 Directories and File Metadata		111
5.1	Considering Directory Contents	111
	5.1.1 Definitions	111
	5.1.2 Directory Contents	113
	5.1.3 Hard Links	114
	5.1.4 File Renaming	117
	5.1.5 File Removal	118
	5.1.6 Symbolic Links	119
5.2	Creating and Removing Directories	121
5.3	Reading Directories	123
	5.3.1 Basic Directory Reading	124
	5.3.2 BSD Directory Positioning Functions	129
5.4	Obtaining Information about Files	130
	5.4.1 Linux File Types	130
	5.4.2 Retrieving File Information	131
	5.4.3 Linux Only: Specifying Higher-Precision File Times	135
	5.4.4 Determining File Type	135
	5.4.5 Working with Symbolic Links	141
5.5	Avoiding Race Conditions: <code>openat()</code> and Friends	144
5.6	Changing Ownership, Permission, and Modification Times	145
	5.6.1 Changing File Ownership: <code>chown()</code> , <code>fchown()</code> , and <code>lchown()</code>	146
	5.6.2 Changing Permissions: <code>chmod()</code> and <code>fchmod()</code>	147
	5.6.3 Using <code>fchown()</code> and <code>fchmod()</code> for Security	147
	5.6.4 Changing Timestamps: <code>utime()</code> and Successors	148
5.7	Summary	151
	Exercises	153
Chapter 6 General Library Interfaces—Part 1		155
6.1	Times and Dates	155
	6.1.1 Retrieving the Current Time: <code>time()</code> and <code>difftime()</code>	156
	6.1.2 Breaking Down Times: <code>gmtime()</code> and <code>localtime()</code>	157
	6.1.3 Formatting Dates and Times	159
	6.1.4 Converting a Broken-Down Time to a <code>time_t</code>	164
	6.1.5 Parsing a Date and Time into a <code>struct tm</code>	166
	6.1.6 Getting Time-Zone Information	168
6.2	Sorting and Searching Functions	171
	6.2.1 Sorting: <code>qsort()</code>	171
	6.2.2 Binary Searching: <code>bsearch()</code>	180
6.3	User and Group Names	190
	6.3.1 User Database	190
	6.3.2 Group Database	193

6.4	Terminals: <code>isatty()</code>	196
6.5	Suggested Reading	196
6.6	Summary	197
	Exercises	198
Chapter 7 Putting It All Together: <code>ls</code>		201
7.1	V7 <code>ls</code> Options	201
7.2	V7 <code>ls</code> Code	202
7.3	Summary	218
	Exercises	218
Chapter 8 Filesystems and Directory Walks		221
8.1	Mounting and Unmounting Filesystems	221
	8.1.1 Reviewing the Background	221
	8.1.2 Looking at Different Filesystem Types	224
	8.1.3 Mounting Filesystems: <code>mount</code>	225
	8.1.4 Unmounting Filesystems: <code>umount</code>	228
8.2	Files for Filesystem Administration	228
	8.2.1 Using Mount Options	230
	8.2.2 Working with Mounted Filesystems: <code>getmntent()</code>	231
8.3	Retrieving Per-Filesystem Information	234
	8.3.1 POSIX Style: <code>statvfs()</code> and <code>fstatvfs()</code>	235
	8.3.2 Linux Style: <code>statfs()</code> and <code>fstatfs()</code>	242
8.4	Moving Around in the File Hierarchy	247
	8.4.1 Changing Directory: <code>chdir()</code> and <code>chdir()</code>	247
	8.4.2 Getting the Current Directory: <code>getcwd()</code>	248
	8.4.3 Processing a File Hierarchy: <code>fts_open()</code> and Friends	250
8.5	Processing a File Hierarchy: GNU <code>du</code>	261
8.6	Changing the Root Directory: <code>chroot()</code>	269
8.7	Summary	270
	Exercises	271
PART II Processes, Networking, and Internationalization		273
Chapter 9 Process Management and Pipes		275
9.1	Process Creation and Management	275
	9.1.1 Creating a Process: <code>fork()</code>	275
	9.1.2 Identifying a Process: <code>getpid()</code> and <code>getppid()</code>	279
	9.1.3 Setting Process Priority: <code>nice()</code>	282
	9.1.4 Starting New Programs: The <code>exec()</code> Family	283
	9.1.5 Terminating a Process	289
	9.1.6 Recovering a Child's Exit Status	293

9.2	Process Groups	300
9.2.1	Job Control Overview	300
9.2.2	Process Group Identification: <code>getpgrp()</code> and <code>getpgid()</code>	301
9.2.3	Process Group Setting: <code>setpgid()</code> and <code>setpgrp()</code>	302
9.3	Basic Interprocess Communication: Pipes and FIFOs	302
9.3.1	Pipes	303
9.3.2	FIFOs	306
9.4	File Descriptor Management	307
9.4.1	Duplicating Open Files: <code>dup()</code> and <code>dup2()</code>	307
9.4.2	Creating Nonlinear Pipelines: <code>/dev/fd/XX</code>	314
9.4.3	Managing File Attributes: <code>fcntl()</code>	315
9.5	Example: Two-Way Pipes in <code>gawk</code>	323
9.6	Suggested Reading	327
9.7	Summary	328
	Exercises	330
Chapter 10 Signals		333
10.1	Introduction	333
10.2	Signal Actions	333
10.3	Standard C Signals: <code>signal()</code> and <code>raise()</code>	334
10.3.1	The <code>signal()</code> Function	334
10.3.2	Sending Signals Programmatically: <code>raise()</code>	337
10.4	Signal Handlers in Action	337
10.4.1	Traditional Systems	337
10.4.2	BSD and GNU/Linux	340
10.4.3	Ignoring Signals	340
10.4.4	Restartable System Calls	341
10.4.5	Race Conditions and <code>sig_atomic_t</code> (ISO C)	344
10.4.6	Additional Caveats	346
10.4.7	Our Story So Far, Episode I	346
10.5	The System V Release 3 Signal APIs: <code>sigset()</code> et al.	349
10.6	POSIX Signals	350
10.6.1	Uncovering the Problem	351
10.6.2	Signal Sets: <code>sigset_t</code> and Related Functions	351
10.6.3	Managing the Signal Mask: <code>sigprocmask()</code> et al.	352
10.6.4	Catching Signals: <code>sigaction()</code>	353
10.6.5	Retrieving Pending Signals: <code>sigpending()</code>	357
10.6.6	Making Functions Interruptible: <code>siginterrupt()</code>	357
10.6.7	Sending Signals: <code>kill()</code> and <code>killpg()</code>	358
10.6.8	Our Story So Far, Episode II	360
10.7	Signals for Interprocess Communication	360
10.8	Important Special-Purpose Signals	363
10.8.1	Alarm Clocks: <code>sleep()</code> , <code>alarm()</code> , and <code>SIGALRM</code>	363
10.8.2	Job Control Signals	365
10.8.3	Parental Supervision: Three Different Strategies	366

10.9	Signals across <code>fork()</code> and <code>exec()</code>	378
10.10	Summary	379
	Exercises	381
Chapter 11	Permissions and User and Group ID Numbers	383
11.1	Checking Permissions	383
	11.1.1 Real and Effective IDs	383
	11.1.2 Setuid and Setgid Bits	384
11.2	Retrieving User and Group IDs	385
11.3	Checking as the Real User: <code>access()</code>	388
11.4	Setting Extra Permission Bits for Directories	391
	11.4.1 Default Group for New Files and Directories	391
	11.4.2 Directories and the Sticky Bit	392
11.5	Setting Real and Effective IDs	393
	11.5.1 Changing the Group Set	393
	11.5.2 Changing the Real and Effective IDs	394
	11.5.3 Using the Setuid and Setgid Bits	396
11.6	Working with All Three IDs: <code>getresuid()</code> and <code>setresuid()</code> (Linux)	398
11.7	Crossing a Security Minefield: Setuid root	399
11.8	Suggested Reading	400
11.9	Summary	401
	Exercises	403
Chapter 12	Resource Limits	405
12.1	Introduction	405
12.2	System Limits: <code>sysconf()</code> , <code>pathconf()</code> , and <code>fpathconf()</code>	405
	12.2.1 How It Works	405
	12.2.2 System Configuration Constants: <code>sysconf()</code>	406
	12.2.3 Filesystem Limitations: <code>pathconf()</code> and <code>fpathconf()</code>	409
12.3	Getting Configuration String Variables: <code>confstr()</code>	411
12.4	Basic Process Limits: <code>ulimit()</code>	413
12.5	Hard and Soft Limits: <code>getrlimit()</code> and <code>setrlimit()</code>	414
12.6	Summary	417
	Exercises	418
Chapter 13	General Library Interfaces—Part 2	419
13.1	Assertion Statements: <code>assert()</code>	419
13.2	Low-Level Memory: The <code>memXXX()</code> Functions	423
	13.2.1 Setting Memory: <code>memset()</code>	423
	13.2.2 Copying Memory: <code>memcpy()</code> , <code>memmove()</code> , and <code>memccpy()</code>	423
	13.2.3 Comparing Memory Blocks: <code>memcmp()</code>	425
	13.2.4 Searching for a Byte Value: <code>memchr()</code>	426
13.3	Temporary Files	426
	13.3.1 Generating Temporary File Names (Bad)	426

13.3.2	Creating and Opening Temporary Files (Good)	430
13.3.3	Using the TMPDIR Environment Variable	433
13.4	Committing Suicide: <code>abort()</code>	434
13.5	Nonlocal Gotos	435
13.5.1	Using Standard Functions: <code>setjmp()</code> and <code>longjmp()</code>	435
13.5.2	Handling Signal Masks: <code>sigsetjmp()</code> and <code>siglongjmp()</code>	437
13.5.3	Observing Important Caveats	438
13.6	Pseudorandom Numbers	442
13.6.1	Standard C: <code>rand()</code> and <code>srand()</code>	443
13.6.2	POSIX Functions: <code>random()</code> and <code>srandom()</code>	445
13.6.3	The <code>/dev/random</code> and <code>/dev/urandom</code> Special Files	447
13.6.4	Using <code>getrandom()</code> Instead of <code>/dev/urandom</code>	449
13.6.5	Cryptographically Secure Random Numbers	450
13.7	Metacharacter Expansions	450
13.7.1	Simple Pattern Matching: <code>fnmatch()</code>	451
13.7.2	File Name Expansion: <code>glob()</code> and <code>globfree()</code>	453
13.7.3	Shell Word Expansion: <code>wordexp()</code> and <code>wordfree()</code>	458
13.8	Regular Expressions	459
13.9	Suggested Reading	467
13.10	Summary	468
	Exercises	469
Chapter 14	Sockets and Basic Networking	473
14.1	Introduction, with a Little Bit of History	473
14.2	Networking Technologies	474
14.3	Internet Building Blocks	474
14.3.1	IPv4 Addresses	475
14.3.2	IPv6 Addresses	475
14.3.3	Addresses and Interfaces	476
14.3.4	Network Byte Order	476
14.4	Networking and Client/Server	477
14.5	Basic Structure of a Server Program	478
14.5.1	Creating a Socket: <code>socket()</code>	478
14.5.2	Associating the Socket with an Address: <code>bind()</code>	481
14.5.3	Waiting for a Connection: <code>listen()</code>	482
14.5.4	Starting a Conversation: <code>accept()</code>	483
14.5.5	Running the Application: <code>read()/write()</code>	484
14.5.6	Cleaning Up: <code>close()</code> and <code>shutdown()</code>	485
14.5.7	Identifying the Ends of a Connection	485
14.5.8	Example Server Code: <code>ftpd</code>	486
14.6	Basic Structure of a Client Program	488
14.6.1	Creating a Socket: <code>socket()</code>	489
14.6.2	Making the Call: <code>connect()</code>	489
14.6.3	Example Client Code: <code>ftp</code>	489

14.7	Specialized Send and Receive Functions	492
14.7.1	Using <code>send()</code> and <code>recv()</code>	493
14.7.2	Connectionless Communication: UDP Sockets	494
14.8	Handling Multiple Open Connections: <code>select()</code>	494
14.8.1	Example Code: 4.4 BSD <code>inetd</code>	496
14.9	<code>pselect()</code> : A Smarter Version of <code>select()</code>	501
14.10	Unix-Domain Sockets	502
14.11	Suggested Reading	502
14.12	Summary	503
	Exercises	504
Chapter 15	Internationalization and Localization	507
15.1	Introduction	507
15.2	Locales and the C Library	508
15.2.1	Locale Categories and Environment Variables	508
15.2.2	Setting the Locale: <code>setlocale()</code>	510
15.2.3	String Collation: <code>strcoll()</code> and <code>strxfrm()</code>	512
15.2.4	Low-Level Numeric and Monetary Formatting: <code>localeconv()</code>	515
15.2.5	High-Level Numeric and Monetary Formatting: <code>strfmon()</code> and <code>printf()</code>	519
15.2.6	Example: Formatting Numeric Values in <code>gawk</code>	521
15.2.7	Formatting Date and Time Values: <code>ctime()</code> and <code>strftime()</code>	523
15.2.8	Other Locale Information: <code>n1_langinfo()</code>	524
15.3	Dynamic Translation of Program Messages	526
15.3.1	Setting the Text Domain: <code>textdomain()</code>	527
15.3.2	Translating Messages: <code>gettext()</code>	527
15.3.3	Working with Plurals: <code>ngettext()</code>	528
15.3.4	Making <code>gettext()</code> Easy to Use	529
15.3.5	Rearranging Word Order with <code>printf()</code>	533
15.3.6	Testing Translations in a Private Directory	534
15.3.7	Setting the Output Codeset	534
15.3.8	Preparing Internationalized Programs	535
15.3.9	Creating Translations	536
15.4	Can You Spell That for Me, Please?	540
15.4.1	Wide Characters	541
15.4.2	Multibyte Character Encodings	545
15.4.3	Converting Bytes to Wide Characters	545
15.4.4	Converting Wide Characters to Bytes	550
15.4.5	Languages	552
15.4.6	Conclusion	553
15.5	Suggested Reading	553
15.6	Summary	553
	Exercises	555

Chapter 16	Extended Interfaces	557
16.1	Allocating Aligned Memory: <code>posix_memalign()</code> and <code>memalign()</code>	557
16.2	Locking Files	558
16.2.1	File Locking Concepts	558
16.2.2	POSIX Locking: <code>fcntl()</code> and <code>lockf()</code>	559
16.2.3	BSD Locking: <code>flock()</code>	565
16.2.4	Mandatory Locking	566
16.3	More Precise Times	567
16.3.1	Microsecond Times: <code>gettimeofday()</code>	567
16.3.2	Nanosecond Times: <code>clock_gettime()</code>	569
16.3.3	Interval Timers: <code>setitimer()</code> and <code>getitimer()</code>	570
16.3.4	More Exact Pauses: <code>nanosleep()</code>	573
16.4	Advanced Searching with Binary Trees	575
16.4.1	Introduction to Binary Trees	575
16.4.2	Tree Management Functions	577
16.4.3	Tree Insertion: <code>tsearch()</code>	578
16.4.4	Tree Lookup and Use of a Returned Pointer: <code>tfind()</code> and <code>tsearch()</code>	579
16.4.5	Tree Traversal: <code>twalk()</code>	581
16.4.6	Tree Node Removal and Tree Deletion: <code>tdelete()</code> and <code>tdestroy()</code>	585
16.5	Summary	586
	Exercises	587
PART III	Debugging and Final Project	589
Chapter 17	Debugging	591
17.1	First Things First	591
17.2	Compilation for Debugging	592
17.3	GDB Basics	593
17.3.1	Getting a core File	594
17.3.2	Running GDB	597
17.3.3	Setting Breakpoints, Single-Stepping, and Setting Watchpoints	599
17.3.4	Escaping the Line-at-a-Time Jail	603
17.3.5	Repeatable, Reversible Debugging with <code>rr</code>	604
17.3.6	Honorable Mention: <code>lldb</code>	605
17.4	Programming for Debugging	606
17.4.1	Compile-Time Debugging Code	606
17.4.2	Runtime Debugging Code	622
17.5	Debugging Tools I: A Modern <code>lint</code>	632
17.6	Debugging Tools II: Memory Allocation Debuggers	633
17.6.1	Valgrind: A Versatile Tool	634
17.6.2	Address Sanitizer	644
17.7	Asking for Help	650
17.8	Software Testing	651

17.9 Debugging Rules	653
17.10 Suggested Reading	655
17.11 Summary	656
Exercises	657
Chapter 18 A Project That Ties Everything Together	659
18.1 Project Description	659
18.2 Suggested Reading	661
PART IV Appendices	663
Appendix A Teach Yourself Programming in Ten Years	665
Appendix B Caldera Ancient UNIX License	671
Appendix C GNU General Public License	673
Appendix D License for the One True Awk	685
Appendix E License for 4.4 BSD Code	687
Index	689

Foreword

Everyone remembers when they started programming. You could have started in elementary school, high school or college (like me), or before. You could have been a hobbyist, finding interesting things on the Internet and wanting to tinker. Maybe you started as part of a job.

Either way, it wasn't long before you found C, attracted by its spare elegance, its economy of expression, and the power it afforded to do surprisingly complex things. Whether you started with the "Hello, World" program—something of a koan in the C universe—from *The C Programming Language*, or a different book, or no book at all, you quickly discovered that the path to good programming practices was to understand the power of the programming environment and learn from well-written programs that used it. But where to find them?

This is the book for you.

This book fills the gap between learning C and its syntax and being able to use it to write useful applications in the GNU/Linux environment. You'll find its explanations of the APIs clear and coherent, and its use of existing programs to show how everything fits together illustrative.

Arnold Robbins has distilled years of programming practice into this volume, and shared tidbits of his own experiences, mistakes, and solutions. After reading, I'm confident that you'll appreciate the GNU/Linux environment and what you can do with it.

It's something I wish I had had way back when I first started to learn and program in C. I think it will become an essential part of your programming journey.

— Chet Ramey
Novelty, Ohio
USA

Preface

One of the best ways to learn about programming is to read well-written programs. This book teaches the fundamental Linux system APIs—those that form the core of any significant program—by presenting code from production programs that you use every day.

By looking at concrete programs, you not only can see how to use the Linux APIs but also can examine the real-world issues (performance, portability, robustness) that arise in writing software.

While the book’s title is *Linux Application Development by Example*, everything we cover, unless otherwise noted, applies to other Unix-derived systems as well.¹ In particular, we focus on GNU/Linux systems. In general we use “Linux” to mean the Linux kernel, and “GNU/Linux” to mean the total system (kernel, libraries, tools). Also, we often say “Linux” when we mean all of Linux, GNU/Linux, and Unix; if something is specific to one system or the other, we mention it explicitly.

Audience

This book is intended for the person who understands programming and is familiar with the basics of C, at least on the level of *The C Programming Language* by Brian Kernighan and Dennis Ritchie. (Java, Python, and Go programmers wishing to read this book should understand C pointers, since C code makes heavy use of them.) The examples use both the 1999 version of Standard C and Original C.²

In particular, you should be familiar with all C operators, control-flow structures, variable and pointer declarations and use, the string management functions, the use of `exit()`, and the `<stdio.h>` suite of functions for file input/output.

You should understand the basic concepts of *standard input*, *standard output*, and *standard error*, and the fact that all C programs receive an array of character strings representing invocation options and arguments. You should also be familiar with the fundamental command-line tools, such as `cd`, `cp`, `date`, `ln`, `ls`, `man` (and `info` if you have it), `rmdir`, and `rm`; the use of long and short command-line options; environment variables; and I/O redirection, including pipes.

¹Systems derived from the original Unix source code, such as Solaris, HP-UX, and AIX, can today be classified as legacy systems. However, there are a number of BSD-derived systems in active development, such as NetBSD, FreeBSD, and OpenBSD, along with many others.

²Although there are newer C standards, we don’t need any of the features they offer.

We assume that you want to write programs that work not just under GNU/Linux but across the range of Unix-like systems. To that end, we mark each interface as to its availability (GLIBC systems only, or defined by POSIX, and so on), and portability advice forms an integral part of the text.

The programming taught here may be at a lower level than you're used to; that's OK. The system calls are the fundamental building blocks for higher operations and are thus low level by nature. This in turn dictates our use of C: the APIs were designed for use from C, and code that interfaces them to higher-level languages, such as C++, Java, Python, or Go, will necessarily be lower level in nature, and most likely, written in C. It may help to remember that "low level" doesn't mean "bad," it just means "more challenging."

What You Will Learn

Fundamentals may get old, but they usually don't become wrong.
— Grant Taylor

This book focuses on the basic APIs that form the core of Linux programming, the *fundamental* things you need to know to write software that makes good use of the GNU/Linux and POSIX APIs:

- Arguments, options, and the environment
- Memory management
- File input/output
- File metadata
- Processes and signals
- Interprocess communication
- Users and groups
- Resource limits
- Programming support (sorting, argument parsing, and so on)
- Basic networking
- Internationalization
- Debugging

We have purposely kept the list of topics short. We believe that it is intimidating to try to learn "all there is to know" from a single book. Most readers prefer smaller, more focused books, and the best Unix books are all written that way.³

³Although this book isn't exactly small, we've seen several POSIX programming books that are easily twice as big.

We have also made an effort to avoid too much information (TMI), giving you exactly what you need to do your work: no less, but also no more.

The APIs we cover include both system calls and library functions. Indeed, at the C level, both appear as simple function calls. A *system call* is a direct request for system services, such as reading or writing a file or creating a process. A *library function*, on the other hand, runs at the user level, possibly never requesting any services from the operating system. System calls are documented in section 2 of the online reference manual and library functions are documented in section 3. (As with the online manual pages for commands, you access the manual pages for system calls and library functions with the `man` command.)

Our goal is to teach you the use of the Linux APIs by example, in particular through the use, wherever possible, of both original Unix source code and the GNU utilities. Unfortunately, there aren't as many self-contained examples as we thought there'd be. Thus, we have written numerous small demonstration programs as well. We stress programming principles, especially those aspects of GNU programming such as “no arbitrary limits,” that make the GNU utilities into exceptional programs.

The choice of everyday programs to study is deliberate. If you've been using GNU/Linux for any length of time, you already understand what programs such as `ls` and `cp` do; it then becomes easy to dive straight into *how* the programs work, without having to spend a lot of time learning *what* they do.

Occasionally, we present both higher-level and lower-level ways of doing things. Usually the higher-level standard interface is implemented in terms of the lower-level interface or construct. We hope that such views of what's “under the hood” will help you understand how things work; for all the code you write, you should always use the higher-level, standard interface.

Similarly, we sometimes introduce functions that provide certain functionality and then recommend (with a stated reason) that these functions be avoided! The primary reason for this approach is so that you'll be able to recognize these functions when you see them and thus understand the code using them. A well-rounded knowledge of a topic requires understanding not just what you can do, but what you should and should not do.

Finally, each chapter concludes with suggested readings and exercises. Some of the exercises involve modifying or writing code. Others are more in the category of “thought experiments” or “Why do you think ...?” We recommend that you do all of them—they will help cement your understanding of the material.

Small Is Beautiful: Unix Programs

Hoare's law:
Inside every large program is a small program struggling to get out.
— C. A. R. Hoare

Initially, we planned to teach the Linux API by using the code from the GNU utilities. However, the modern versions of even simple command-line programs (like `mv` and `cp`) are large and many-featured. This is particularly true of the GNU variants of the standard utilities, which allow long and short options, do everything required by POSIX, and often have additional, seemingly unrelated options as well (like output highlighting).

It then becomes reasonable to ask, “Given such a large and confusing forest, how can we focus on the one or two important trees?” In other words, if we present the current full-featured program, will it be possible to see the underlying core operation of the program?

That is when *Hoare's law*⁴ inspired us to look to the original Unix programs for example code. The original V7 Unix utilities are small and straightforward, making it easy to see what's going on and to understand how the system calls are used. (V7 was released around 1979; it is the common ancestor of all modern Unix systems, including GNU/Linux and the BSD systems.)

For many years, Unix source code was protected by copyrights and trade secret license agreements, making it difficult to use for study and impossible to publish. This is still true of all commercial Unix source code. However, in 2002, Caldera (currently operating as SCO) made the original Unix code (through V7 and 32V Unix) available under an Open Source-style license (see Appendix B, “Caldera Ancient UNIX License,” page 671). This makes it possible for us to include the code from the early Unix system in this book.

Standards

Throughout the book we refer to several different formal standards. A *standard* is a document describing how something works. Formal standards exist for many things; for example, the shape, placement, and meaning of the holes in the electrical outlet in your wall are defined by a formal standard so that all the power cords in your country work in all the outlets.

So, too, do formal standards for computing systems define how they are supposed to work; this enables developers and users to know what to expect from their software and enables them to complain to their vendor when software doesn't work.

Of interest to us here are:

⁴This famous statement was made at *The International Workshop on Efficient Production of Large Programs* in Jablonna, Poland, August 10–14, 1970.

1. *ISO/IEC International Standard 9899:1999 Programming Languages — C, second edition, 1999.* The second formal standard for the C programming language.
2. *ISO/IEC International Standard 14882:2024 Programming Languages — C++, 2024.* The current formal standard for the C++ programming language.
3. *The Open Group Base Specifications Issue 8, IEEE Std 1003.1TM–2024 edition.*⁵ The current version of the POSIX standard; it describes the behavior expected of Unix and Unix-like systems. This edition covers both the system call and library interface, as seen by the C/C++ programmer, and the shell and utilities interface, seen by the user. It consists of several volumes:
 - *Base Definitions.* The definitions of terms, facilities, and header files.
 - *System Interfaces.* The system calls and library functions. POSIX terms them all “functions.”
 - *Shell and Utilities.* The shell language and utilities available for use with shell programs and interactively.
 - *Rationale.* Explanations and rationales for the choice of facilities that are or are not included in the standard.

Although language standards aren’t exciting reading, you may wish to consider purchasing a copy of the C standard, as it provides the final definition of the language. Copies can be purchased from ANSI⁶ and from ISO.⁷ (The PDF version of the C standard is quite affordable.)

The POSIX standard is available online. The standard is intended for implementation on both Unix and Unix-like systems, as well as on non-Unix systems. Thus, the base functionality it provides is a subset of what Unix systems have. However, the POSIX standard also defines optional *extensions*—additional functionality, for example, for threads or real-time support. Of most importance to us is the *X/Open System Interface* (XSI) extension, which describes facilities from historical Unix systems.

Throughout the book, we mark each API as to its availability: ISO C, POSIX, XSI, GLIBC only, or nonstandard but commonly available.

Features and Power: GNU Programs

Restricting ourselves to just the original Unix code would have made an interesting history book, but it would not have been very useful a quarter of the way into the twenty-first century. Modern programs do not have the same constraints (memory, CPU power, disk space and speed) that the early Unix systems did. Furthermore, they need to operate in a multilingual world—ASCII and American English aren’t enough.

⁵<https://pubs.opengroup.org/onlinepubs/9799919799/>

⁶<http://www.ansi.org>

⁷<http://www.iso.ch>

More importantly, one of the primary freedoms expressly promoted by the Free Software Foundation (FSF) and the GNU Project⁸ is the “freedom to study.” GNU programs are intended to provide a large corpus of well-written programs that journeyman programmers can use as a source from which to learn.

By using GNU programs, we want to meet both goals: show you well-written, modern code from which you will learn how to write good code and how to use the APIs well.

We believe that GNU software is better because it is free (in the sense of “freedom,” not “free beer”). But it’s also recognized that GNU software is often *technically* better than the corresponding Unix counterparts, and we devote space in Section 1.4, “Why GNU Programs Are Better,” page 13, to explaining why.

A number of the GNU code examples come from *gawk* (GNU *awk*). The main reason is that it’s a program with which we’re very familiar, and therefore it was easy to pick examples from it. We don’t otherwise make any special claims about it.

Summary of Chapters

Driving a car is a holistic process that involves multiple simultaneous tasks. In many ways, Linux programming is similar, requiring that you understand multiple aspects of the API, such as file I/O, file metadata, directories, storage of time information, and so on.

The first part of the book looks at enough of these individual items to enable studying the first significant program, the V7 1s. Then we complete the discussion of files and users by looking at file hierarchies and the way filesystems work and are used.

Chapter 1, “Introduction,” page 3,

describes the Unix and Linux file and process models, looks at the differences between Original C and 1999 Standard C, and provides an overview of the principles that make GNU programs generally better than standard Unix programs. The chapter also includes advice on programming style, and some thoughts on the use of AI for software development.

Chapter 2, “Arguments, Options, and the Environment,” page 23,

describes how a C program accesses and processes command-line arguments and options and explains how to work with the environment.

Chapter 3, “User-Level Memory Management,” page 51,

provides an overview of the different kinds of memory in use and available in a running process. User-level memory management is central to every nontrivial application, so it’s important to understand it early on.

Chapter 4, “Files and File I/O,” page 79,

discusses basic file I/O, showing how to create and use files. This understanding is important for everything else that follows.

⁸<http://www.gnu.org>

- Chapter 5, “Directories and File Metadata,” page 111,*
describes how directories, hard links, and symbolic links work. It then describes file metadata, such as owners, permissions, and so on, as well as covering how to work with directories.
- Chapter 6, “General Library Interfaces—Part 1,” page 155,*
looks at the first set of general programming interfaces that we need so that we can make effective use of a file’s metadata.
- Chapter 7, “Putting It All Together: 1s,” page 201,*
ties together everything seen so far by looking at the V7 1s program.
- Chapter 8, “Filesystems and Directory Walks,” page 221,*
describes how filesystems are mounted and unmounted and how a program can tell what is mounted on the system. It also describes how a program can easily “walk” an entire file hierarchy, taking appropriate action for each object it encounters.
- The second part of the book deals with process creation and management, interprocess communication with pipes and signals, user and group IDs, resource limits, additional general programming interfaces, and basic networking. Next, the book describes internationalization with GNU `gettext` and then several advanced APIs.
- Chapter 9, “Process Management and Pipes,” page 275,*
looks at process creation, program execution, IPC with pipes, and file descriptor management, including nonblocking I/O.
- Chapter 10, “Signals,” page 333,*
discusses signals, a simplistic form of interprocess communication. Signals also play an important role in a parent process’s management of its children.
- Chapter 11, “Permissions and User and Group ID Numbers,” page 383,*
looks at how processes and files are identified, how permission checking works, and how the `setuid` and `setgid` mechanisms work.
- Chapter 12, “Resource Limits,” page 405,*
describes the ways in which processes may use system resources, and in particular how they may be constrained from using too many resources.
- Chapter 13, “General Library Interfaces—Part 2,” page 419,*
looks at the rest of the general APIs; many of these are more specialized than the first general set of APIs.
- Chapter 14, “Sockets and Basic Networking,” page 473,*
introduces the basic facilities for communication between processes on different computers.
- Chapter 15, “Internationalization and Localization,” page 507,*
explains how to enable your programs to work in multiple languages, with almost no pain.

Chapter 16, “Extended Interfaces,” page 557,

describes several extended versions of interfaces covered in previous chapters, as well as covering file locking in full detail.

The third part rounds off the book with a chapter on debugging, since (almost) no one gets things right the first time, and we suggest a final project to cement your knowledge of the APIs covered in this book.

Chapter 17, “Debugging,” page 591,

describes the basics of the GDB debugger, transmits as much of our programming experience in this area as possible, and looks at several useful tools for doing different kinds of debugging.

Chapter 18, “A Project That Ties Everything Together,” page 659,

presents a significant programming project that makes use of just about everything covered in the book.

Several appendices cover topics of interest, including the licenses for the source code used in this book.

Appendix A, “Teach Yourself Programming in Ten Years,” page 665,

invokes the famous saying “Rome wasn’t built in a day.” Similarly, Linux/Unix expertise and understanding come only with time and practice. With that in mind, we have included this essay by Peter Norvig, which we highly recommend.

Appendix B, “Caldera Ancient UNIX License,” page 671,

covers the Unix source code used in this book.

Appendix C, “GNU General Public License,” page 673,

covers the GNU source code used in this book.

Appendix D, “License for the One True Awk,” page 685,

covers the source code from the One True Awk that is used in this book.

Appendix E, “License for 4.4 BSD Code,” page 687,

covers the source code from the 4.4 BSD distribution that is used in this book.

Typographical Conventions

Like all books on computer-related topics, we use certain typographical conventions to convey information. *Definitions* or first uses of terms appear in italics, like the word “Definitions” at the beginning of this sentence. Italics are also used for *emphasis*, for citations of other works, and for commentary in examples. Variable items such as arguments or file names, appear *like this*. Occasionally, we use a bold font when a point needs to be made **strongly**.

Things that exist on a computer are in a constant-width font, such as file names (`foo.c`) and command names (`ls`, `grep`). Short snippets that you type are additionally enclosed in single quotes: `'ls -l *.c'`.

`$` and `>` are the default Bourne shell primary and secondary prompts and are used to display interactive examples. *User input* appears in a different font from regular computer output in examples. Examples look like this:

```
$ ls -l *.texi          Look at files. Option is letter l, not numeral "one"
-rw-rw-r-- 1 arnold arnold 33165 Apr 24 18:03 00-preface.texi
-rw-rw-r-- 1 arnold arnold 48622 Apr 24 18:05 01-intro.texi
-rw-rw-r-- 1 arnold arnold 65930 Apr 23 22:43 02-cmdline.texi
...
```

We prefer the Bourne shell and its variants (`ksh93`, `Bash`) over the C shell; thus, all our examples show only the Bourne shell. Be aware that quoting and line-continuation rules are different in the C shell; if you use it, you're on your own!⁹

When referring to functions in programs, we append an empty pair of parentheses to the function's name: `printf()`, `strcpy()`. When referring to a manual page (accessible with the `man` command), we follow the standard Unix convention of writing the command or function name in italics and the section in parentheses after it, in regular type: *awk*(1), *printf*(3).

Where to Get Unix and GNU Source Code

You may wish to have copies of the programs we use in this book for your own experimentation and review. All the source code is available over the Internet, and your GNU/Linux distribution contains the source code for the GNU utilities.

Unix Code

Archives of various “ancient” versions of Unix are maintained by the UNIX Heritage Society (TUHS).¹⁰

Of most interest is that it is possible to browse the archive of old Unix source code on the Web. Start with <http://www.tuhs.org/UnixTree/>. Almost all the example code in this book is from the Seventh Edition Research UNIX System, also known as “V7.”

The TUHS site is physically located in Australia, although there are mirrors of the archive around the world—see http://www.tuhs.org/archive_sites.html. This page also indicates that the archive is available for mirroring with `rsync`. (See <http://rsync.samba.org/> if you don't have `rsync`; it's standard on GNU/Linux systems.)

As of the time of this writing, you will need about 6.5–7 gigabytes of disk to copy the entire archive. To copy the archive, create an empty directory, and in it, run the following commands:

⁹See the *csh*(1) and *tcsh*(1) manpages and the book *Using csh & tcsh*, by Paul DuBois (O'Reilly & Associates, 1995), ISBN-13: 978-1-56592-132-0.

¹⁰<http://www.tuhs.org>

```
mkdir Applications Distributions Documentation Tools
```

```
rsync -avz minnie.tuhs.org::UA_Root      .
rsync -avz minnie.tuhs.org::UA_Applications Applications
rsync -avz minnie.tuhs.org::UA_Distributions Distributions
rsync -avz minnie.tuhs.org::UA_Documentation Documentation
rsync -avz minnie.tuhs.org::UA_Tools     Tools
```

(For your convenience, these commands are available in a script named `get-tuhs-files.sh` in the `ch-01-intro` directory of the book's GitHub repository.)

It's interesting to note that V7 code does not contain any copyright or permission notices in it. The authors wrote the code primarily for themselves and their research, leaving the permission issues to AT&T's corporate licensing department.

GNU Code

If you're using GNU/Linux, then your distribution will have come with source code, presumably in whatever packaging format it uses (Red Hat RPM files, Debian DEB files, Slackware `.tar.gz` files, etc.). The examples in the book are from the GNU Coreutils, version 9.4. You can search for the appropriate source package for your distribution, if you like. Or follow the instructions in the next few paragraphs to retrieve the code.

If you prefer to retrieve the files yourself from the GNU FTP site, you will find them at <https://ftp.gnu.org/gnu/coreutils/coreutils-9.4.tar.gz>.¹¹

You can use either `wget` or `curl` to retrieve the file:

```
$ wget https://ftp.gnu.org/gnu/coreutils/coreutils-9.4.tar.gz
... lots of output here as file is retrieved ...
```

Alternatively, you can use good old-fashioned `ftp`¹² to retrieve the file:

```
$ ftp ftp.gnu.org           Connect to the GNU ftp site
Trying 209.51.188.20:21 ...
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:arnold): anonymous           Use anonymous ftp
...                                             Site notice deleted
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /gnu/coreutils           Change to the Coreutils directory
250 Directory successfully changed.
ftp> bin
200 Switching to Binary mode.
ftp> get coreutils-9.4.tar.gz     Retrieve the file
```

¹¹The tar file is also available compressed with xz; this file is somewhat smaller than the one compressed with gzip.

¹²Note that the FSF prefers you use HTTPS instead of FTP; their server indicates that they will eventually disable FTP access.

```

local: coreutils-9.4.tar.gz remote: coreutils-9.4.tar.gz
229 Entering Extended Passive Mode (|||25110|)
150 Opening BINARY mode data connection for coreutils-9.4.tar.gz (14714577 bytes).
100% |*****| 14369 KiB 792.25 KiB/s 00:00 ETA
226 Transfer complete.
14714577 bytes received in 00:18 (785.75 KiB/s)
ftp> quit Log off
221 Goodbye.

```

Once you have the file, extract it as follows:

```

$ gzip -dc < coreutils-9.4.tar.gz | tar -xvpf - Extract files
... lots of output here as files are extracted ...

```

Systems using GNU tar may use this incantation:

```

$ tar -xvpzf coreutils-9.4.tar.gz Extract files
... lots of output here as files are extracted ...

```

In compliance with the GNU General Public License, here is the copyright information for all GNU programs quoted in this book: all the programs are “free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.” See Appendix C, “GNU General Public License,” page 673, for the text of the GNU General Public License.

Coreutils 9.4 file
lib/safe-read.c
lib/safe-write.c
src/du.c
src/env.c
src/install.c
src/link.c
src/ls.c
src/sort.c
src/stdbuf.c
src/system.h

Copyright dates
Copyright © 1993–1994, 1998, 2002–2006, 2009–2023
Copyright © 2002, 2009–2023
Copyright © 1988–2023
Copyright © 1986–2023
Copyright © 1989–2023
Copyright © 2001–2023
Copyright © 1985–2023
Copyright © 1988–2023
Copyright © 2009–2023
Copyright © 1989–2023

Gawk 3.0.0 file
posix/gawkmisc.c

Copyright dates
Copyright © 1986, 1988, 1989, 1991–1995

Gawk 3.0.6 file
eval.c

Copyright dates
Copyright © 1986, 1988, 1989, 1991–2000

Gawk 3.1.8 file
eval.c
io.c

Copyright dates
Copyright © 1986, 1988, 1989, 1991–2010
Copyright © 1986, 1988, 1989, 1991–2010

Gawk 5.3.0 file	Copyright dates
awk.h	Copyright © 1986, 1988, 1989, 1991–2023
builtin.c	Copyright © 1986, 1988, 1989, 1991–2023
io.c	Copyright © 1986, 1988, 1989, 1991–2023
main.c	Copyright © 1986, 1988, 1989, 1991–2023
node.c	Copyright © 1986, 1988, 1989, 1991–2001, 2003–2015, 2017–2019, 2021–2023
posix/gawkmisc.c	Copyright © 1986, 1988, 1989, 1991–1998, 2001–2004, 2011, 2021–2023

Gawk 5.3.1 file	Copyright dates
printf.c	Copyright © 1986, 1988, 1989, 1991–2024

Gettext 0.22.5 file	Copyright dates
gettext.h	Copyright © 1995–1998, 2000–2002, 2004–2006, 2009–2020

GLIBC 2.35 file	Copyright dates
/usr/include/locale.h	Copyright © 1991–2002

Make 4.4.1 file	Copyright dates
read.c	Copyright © 1988–2023

We use a few examples from the One True AWK, published by Brian Kernighan. Appendix D, “License for the One True Awk,” page 685, presents its license. Here is the copyright information for it:

One True Awk file	Copyright dates
awk.h	Copyright © 1997
b.c	Copyright © 1997

Example code from 4.2 BSD used in Section 14.5.8, “Example Server Code: ftpd,” page 486, is covered by the Caldera Open Source license.

Finally, we use some example code from 4.4 BSD. The license for this code is in Appendix E, “License for 4.4 BSD Code,” page 687. Here is the relevant copyright information.

4.4 BSD file	Copyright dates
ftp.c	Copyright © 1985, 1989, 1993
inetd.c	Copyright © 1983, 1991

Where to Get the Example Programs

The example programs are available on GitHub. You can get them from <https://github.com/arnoldrobbins/LinuxByExample-2e>. Errata for the book will be available there as well.

Acknowledgments for the Second Edition

Thanks to Debra Williams and Mark Taub for endorsing a new edition of this book.

Thanks to Chet Ramey for answering questions about macOS, POSIX, and other things. I thank him also for his kind words in the Foreword.

Thanks to Nelson H. F. Beebe for answering questions related to Solaris and other Unix systems. Thanks to Geoff Clare for help with questions related to POSIX.

Gavin Smith and Patrice Dumas currently maintain and develop the Texinfo markup language and its toolset, which I used to write the book. I thank them.

Thanks to Matthew Helmke, Professor Brian Kernighan, Chet Ramey, and Miriam Robbins for their technical reviews of this edition. Their comments materially improved the entire text. As for the previous edition, any remaining errors are mine.

I thank Arthur Johnson for his amazing copyediting, and Chuti Prasertsith for the fantastic cover design. Thanks to the production team headed by Julie Nahil and Sumitra Boopalan.

Once again, my deepest gratitude and love to my wife, Miriam, for her support and encouragement during the book's writing.

— Nof Ayalon, ISRAEL
September 2025

Acknowledgments for the First Edition

Writing a book is lots of work, and doing it well requires help from many people. Dr. Brian W. Kernighan, Dr. Doug McIlroy, Peter Memishian, and Peter van der Linden reviewed the initial book proposal. David J. Agans, Fred Fish, Don Marti, Jim Meyering, Peter Norvig, and Julian Seward provided reprint permission for various items quoted throughout the book. Thanks to Geoff Collyer, Ulrich Drepper, Yosef Gold, Dr. C. A. R. (Tony) Hoare, Dr. Manny Lehman, Jim Meyering, Dr. Dennis M. Ritchie, Julian Seward, Henry Spencer, and Dr. Wladyslaw M. Turski, who provided much useful general information. Thanks also to the other members of the GNITS gang: Karl Berry, Akim DeMaille, Ulrich Drepper, Greg McGary, Jim Meyering, François Pinard, and Tom Tromeu, who all provided helpful feedback about good programming practice. Karl Berry, Alper Ersoy, and Dr. Nelson H. F. Beebe provided valuable technical help with the Texinfo and DocBook/XML toolchains.

Good technical reviewers not only make sure that an author gets his facts right, but they also ensure that he thinks carefully about his presentation. Dr. Nelson H. F. Beebe, Geoff Collyer, Russ Cox, Ulrich Drepper, Dr. Brian W. Kernighan, Randy Lechlitner, Peter Memishian, Jim Meyering, Chet Ramey, and Louis Taber acted as technical reviewers for the entire book. Dr. Michael Brennan provided helpful comments on Chapter 15.

[This is Chapter 17 in the current edition.] Both the prose and many of the example programs benefited from their reviews. I hereby thank all of them. As most authors usually say here, “Any remaining errors are mine.”

I would especially like to thank Mark Taub of Pearson Education for initiating this project, for his enthusiasm for the series, and for his help and advice as the book moved through its various stages. Anthony Gemmellaro did a phenomenal job of realizing my concept for the cover, and Gail Cocker’s interior design is beautiful. Fay Gemmellaro made the production process enjoyable, instead of a chore. Dmitry and Alina Kirsanov did the figures, page layout, and indexing; they were a pleasure to work with.

Finally, my deepest gratitude and love to my wife, Miriam, for her support and encouragement during the book’s writing.

— Nof Ayalon, ISRAEL
April 2004

Register your copy of *Linux Application Development by Example, Second Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135325520) and click Submit. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Chapter 6

General Library Interfaces—Part 1

We saw in Chapter 5, “Directories and File Metadata,” page 111, that directly reading a directory returns file names in the order in which they’re kept in the directory. We also saw that the `struct stat` contains all the information about a file, except its name. However, some components of that structure are not directly usable; they’re just numeric values.

This chapter presents the rest of the APIs needed to make full use of the `struct stat` component values. In order, we cover the following topics: `time_t` values for representing times and the time formatting functions; sorting and searching functions (for sorting file names, or any other data); the `uid_t` and `gid_t` types for representing users and groups and the functions that map them to and from the corresponding user and group names; and finally, a function to test whether a file descriptor represents a terminal.

POSIX systems have many more general-purpose APIs than just the ones in this chapter. We cover more of them later, in Chapter 13, “General Library Interfaces—Part 2,” page 419, and in Chapter 16, “Extended Interfaces,” page 557.

6.1 Times and Dates

Time values are kept in the type known as `time_t`. The ISO C standard guarantees that this is a numeric type but does not otherwise specify what it is (integer or floating-point), or the range or the precision of the values stored therein.

On GNU/Linux and Unix systems, `time_t` values represent “seconds since the Epoch.” The *Epoch* is the beginning of recorded time, which is midnight, January 1, 1970, UTC. On most systems, a `time_t` is a C long int. For 32-bit systems, this means that the `time_t` “overflows” sometime on January 19, 2038. By then, we hope, everyone will be using 64-bit systems (or bigger!), where the `time_t` type is at least 64 bits big.

Various functions exist to retrieve the current time, compute the difference between two `time_t` values, convert `time_t` values into a more usable representation, and format both representations as character strings. Additionally, a date and time representation can be converted back into a `time_t`, and limited time-zone information is available.

A separate set of functions provides access to the current time with a higher resolution than one second. The functions work by providing two discrete values: the time as seconds since the Epoch, and the number of microseconds or nanoseconds within the current second. These functions are described later in the book, in Section 16.3.1 “Microsecond Times: `gettimeofday()`,” page 567, and in Section 16.3.2, “Nanosecond Times: `clock_gettime()`,” page 569.

The data structures were introduced briefly in Section 5.6.4, “Changing Timestamps: `utime()` and Successors,” page 148.

6.1.1 Retrieving the Current Time: `time()` and `difftime()`

The `time()` system call retrieves the current date and time; `difftime()` computes the difference between two `time_t` values:

```
#include <time.h> ISO C
```

```
time_t time(time_t *t);
double difftime(time_t time1, time_t time0);
```

`time()` returns the current time. If the `t` parameter is not `NULL`, then the value pointed to by `t` is also filled in with the current time. It returns (`time_t`) `-1` if there was an error, and `errno` is set.

Although ISO C doesn’t specify what’s in a `time_t` value, POSIX does indicate that it represents time in seconds. Thus, it’s both common and portable to make this assumption. For example, to see if a time value represents something that is six months or more in the past, one might use code like this:

```
/* Error checking omitted for brevity */
time_t now, then, some_time;

time(& now); Get current time
then = now - (6L * 31 * 24 * 60 * 60); Approximately six months ago

... set some_time, for example, via stat() ...
if (some_time < then)
    /* more than 6 months in the past */
else
    /* less than 6 months in the past */
```

However, since strictly portable code may need to run on non-POSIX systems, the `difftime()` function exists to produce the difference between two times. The same test, using `difftime()`, would be written this way:

```
time_t now, some_time;
const double six_months = 6.0 * 31 * 24 * 60 * 60;

time(& now); Get current time
... set some_time, for example, via stat() ...

if (difftime(now, some_time) >= six_months)
    /* more than 6 months in the past */
else
    /* less than 6 months in the past */
```

The return type of `difftime()` is a `double` because a `time_t` could possibly represent fractions of a second as well. On POSIX systems, it always represents whole seconds.

In each of the preceding examples, note the use of a typed constant to force the computation to be done with the right type of math: `6L` in the first instance for long integers; `6.0` in the second, for floating point.

6.1.2 Breaking Down Times: `gmtime()` and `localtime()`

In practice, the “seconds since the Epoch” form of a date and time isn’t very useful except for simple comparisons. Computing the components of a time yourself, such as the month, day, year, and so on, is error prone, since the local time zone (possibly with daylight saving time) must be taken into account, leap years must be computed correctly, and so forth. Fortunately, two standard routines do this job for you:

```
#include <time.h> ISO C

struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
```

`gmtime()` returns a pointer to a `struct tm` that represents UTC time. `localtime()` returns a pointer to a `struct tm` representing the local time; that is, it takes the current time zone and daylight saving time into account. In effect, this is “wall-clock time,” the date and time as it would be displayed on a wall clock or on a wristwatch. (How this works is discussed later; see Section 6.1.6, “Getting Time-Zone Information,” page 168.)

Both functions return a pointer to a `struct tm`, which looks like this:

```
struct tm {
    int    tm_sec;        /* seconds */
    int    tm_min;        /* minutes */
    int    tm_hour;       /* hours */
    int    tm_mday;       /* day of the month */
    int    tm_mon;        /* month */
    int    tm_year;       /* year */
    int    tm_wday;       /* day of the week */
    int    tm_yday;       /* day in the year */
    int    tm_isdst;     /* daylight saving time */
};
```

The `struct tm` is referred to as a *broken-down time*, since the `time_t` value is “broken down” into its component parts. The component parts, their ranges, and their meanings are shown in Table 6.1.

The ISO C standard presents most of these values as “x since y.” For example, `tm_sec` is “seconds since the minute,” `tm_mon` is “months since January,” `tm_wday` is “days since Sunday,” and so on. This helps to understand why all the values start at 0. (The single exception, logically enough, is `tm_mday`, the day of the month, which ranges from 1–31.) Of course,

Table 6.1: Fields in the struct `tm`

Member	Range	Meaning
<code>tm_sec</code>	0–60	Second within a minute. Second 60 allows for leap seconds. (C90 had the range as 0–61.)
<code>tm_min</code>	0–59	Minute within an hour.
<code>tm_hour</code>	0–23	Hour within the day.
<code>tm_mday</code>	1–31	Day of the month.
<code>tm_mon</code>	0–11	Month of the year.
<code>tm_year</code>	0–N	Year, in years since 1900.
<code>tm_wday</code>	0–6	Day of week, Sunday = 0.
<code>tm_yday</code>	0–365	Day of year, January 1 = 0.
<code>tm_isdst</code>	< 0, 0, > 0	Daylight saving time flag.

having them start at zero is also practical; since C arrays are zero-based, it makes using these values as indices trivial:

```
static const char *const days[] = {                                Array of day names
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday",
};
time_t now;
struct tm *curtime;

time(& now);                                                    Get current time
curtime = gmtime(& now);                                        Break it down
printf("Day of the week: %s\n", days[curtime->tm_wday]);        Index and print
```

Both `gmtime()` and `localtime()` return a pointer to a struct `tm`. The pointer points to a static struct `tm` maintained by each routine, and it is likely that these struct `tm` structures are overwritten each time the routines are called. Thus, it's a good idea to make a *copy* of the returned struct. Reusing the previous example:

```
static const char *const days[] = { /* As before */ };
time_t now;
struct tm curtime;                                            Structure, not pointer

time(& now);                                                    Get current time
curtime = *gmtime(& now);                                       Break it down and copy data
printf("Day of the week: %s\n", days[curtime.tm_wday]);        Index and print, use . not ->
```

The `tm_isdst` field indicates whether or not daylight saving time (DST) is currently in effect. A value of 0 means DST is not in effect, a positive value means it is, and a negative value means that no DST information is available. (The C standard is purposely vague, indicating only zero, positive, or negative; this gives implementors the most freedom.)

6.1.3 Formatting Dates and Times

The examples in the previous section showed how the fields in a `struct tm` could be used to index arrays of character strings for printing informative date and time values. While you could write your own code to use such arrays for formatting dates and times, standard routines alleviate the work.

6.1.3.1 Simple Time Formatting: `asctime()` and `ctime()`

The first two standard routines, listed below, produce output in a fixed format:

```
#include <time.h> ISO C

char *asctime(const struct tm *tm);
char *ctime(const time_t *timep);
```

As with `gmtime()` and `localtime()`, `asctime()` and `ctime()` return pointers to static buffers that are likely to be overwritten upon each call. Furthermore, these two routines return strings in the same format. They differ only in the kind of argument they accept. `asctime()` and `ctime()` should be used when all you need is simple date and time information:

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

int
main(void)
{
    time_t now;

    time(& now);
    printf("%s", ctime(& now));
}
```

When run, this program produces output of the form: ‘Sun May 25 16:32:42 2025’. The terminating newline *is* included in the result. To be more precise, the return value points to an array of 26 characters, as shown in Figure 6.1.

Much older Unix code relies on the fact that the values have a fixed position in the returned string. When using these routines, remember that they include a trailing newline. Thus, the

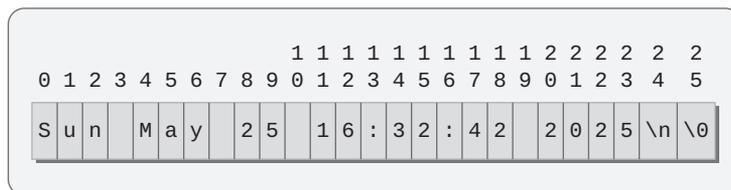


Figure 6.1: Return string from `ctime()` and `asctime()`

small example program uses a simple "%s" format string for `printf()`, and not "%s\n", as might be expected.

`ctime()` saves you the step of calling `localtime()`; it's essentially equivalent to

```
time_t now;
char *curtime;

time(& now);
curtime = asctime(localtime(& now));
```

6.1.3.2 Complex Time Formatting: `strftime()`

While `asctime()` and `ctime()` are often adequate, they are also limited:

- The output format is fixed. There's no way to rearrange the order of the elements.
- The output does not include time-zone information.
- The output uses abbreviated month and day names.
- The output assumes English names for the months and days.

For these reasons, C90 introduced the `strftime()` standard library routine:

```
#include <time.h> ISO C

size_t strftime(char *s, size_t max, const char *format,
                const struct tm *tm);
```

`strftime()` is similar to `sprintf()`. The arguments are as follows:

`char *s`

A buffer to hold the formatted string.

`size_t max`

The size of the buffer.

`const char *format`

The format string.

`const struct tm *tm`

A `struct tm` pointer representing the broken-down time to be formatted.

The format string contains literal characters, intermixed with conversion specifiers that indicate what is to be placed into the string, such as the full weekday name, the hour according to a 24-hour or 12-hour clock, a.m. or p.m. designations, and so on. (Examples are coming shortly.)

If the entire string can be formatted within `max` characters, the return value is the number of characters placed in `s`, *not* including the terminating zero byte. Otherwise, the return value

is 0. In the latter case, the contents of `s` are “indeterminate.” The following simple example gives the flavor of how `strftime()` is used:

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

int
main(void)
{
    char buf[100];
    time_t now;
    struct tm *curtime;

    time(& now);
    curtime = localtime(& now);
    (void) strftime(buf, sizeof buf,
        "It is now %A, %B %d, %Y, %I:%M %p", curtime);

    printf("%s\n", buf);
    exit(0);
}
```

When run, this program prints something like:

```
It is now Sunday, May 25, 2025, 04:34 PM
```

Table 6.2 provides the full list of conversion specifiers, their possible alternative representations, and their meanings.

A *locale* is a way of describing the current location, taking into account such things as language, character set, and defaults for formatting dates, times, and monetary amounts. We deal with them in Chapter 15, “Internationalization and Localization,” page 507. For now, it’s enough to understand that the results from `strftime()` for the same format string can vary according to the current locale.

The versions starting with `%E` and `%O` are for “alternative representations.” Some locales have multiple ways of representing the same thing; these specifiers provide access to the additional representations. If a particular locale does not support alternative representations, then `strftime()` uses the regular version.

Many Unix versions of `date` allow you to provide, on the command line, a format string that begins with a `+` character. `date` then formats the current date and time and prints it according to the format string:

```
$ date +'It is now %A, %B %d, %Y, %I:%M %p'
It is now Sunday, May 25, 2025, 04:35 PM
```

The output depends on the current locale (where in the world the system thinks you are; locales and the environment variables that control them are discussed later, in Section 15.2, “Locales

Table 6.2: `strftime()` conversion format specifiers

Specifier(s)	Meaning
%a	The locale’s abbreviated weekday name.
%A	The locale’s full weekday name.
%b, %Ob	The locale’s abbreviated month name.
%B, %OB	The locale’s full month name.
%c, %Ec	The locale’s “appropriate” date and time representation.
%C, %EC	The century (00–99).
%d, %Od	The day of the month (01–31).
%D	Same as %m/%d/%y.
%e, %Oe	The day of the month. A single digit is preceded with a space (1–31).
%F	Same as %Y-%m-%d (ISO 8601 date format).
%g	The last two digits of week-based year (00–99).
%G	The ISO 8601 week-based year.
%h	Same as %b.
%H, %OH	The hour in a 24-hour clock (00–23).
%I, %OI	The hour in a 12-hour clock (01–12).
%j	The day of the year (001–366).
%m, %Om	The month as a number (01–12).
%M, %OM	The minute as a number (00–59).
%n	A newline character ('\n').
%p	The locale’s AM/PM designation.
%r	The locale’s 12-hour clock time.
%R	Same as %H:%M.
%S, %OS	The second as a number (00–60).
%t	A tab character ('\t').
%T	Same as %H:%M:%S (ISO 8601 time format).
%u, %Ou	ISO 8601 weekday number, Monday = 1 (1–7).
%U, %OU	Week number, first Sunday is first day of week 1 (00–53).
%V, %OV	ISO 8601 week number (01–53).
%w, %Ow	The weekday as a number, Sunday = 0 (0–6).
%W, %OW	Week number, first Monday is first day of week 1 (00–53).
%x, %Ex	The locale’s “appropriate” date representation.
%X, %EX	The locale’s “appropriate” time representation.
%y, %Ey, %Oy	The last two digits of the year (00–99).
%Y, %EY	The year as a number.
%z	Time zone offset from UTC, in the form +h:mm or -h:mm.
%Z	The locale’s time zone, or no characters if no time-zone information is available.
%%	A single %.

and the C Library,” page 508.) For example, by setting the environment variable `LC_ALL` to `it_IT.utf8` (Italy), we might get something like this:

```
$ LC_ALL=it_IT.utf8 date +'It is now %A, %B %d, %Y, %I:%M %p'
It is now domenica, maggio 25, 2025, 04:36
```

Table 6.3: "C" locale values for certain `strftime()` formats

Specifier	Meaning
%a	The first three characters of %A.
%A	One of Sunday, Monday, ..., Saturday.
%b	The first three characters of %B.
%B	One of January, February, ..., December.
%c	Same as %a %b %e %T %Y.
%p	One of AM or PM.
%r	Same as %I:%M:%S %p.
%x	Same as %m/%d/%y.
%X	Same as %T.
%Z	Implementation-defined.

Note that despite the %p in the format string, there is no a.m./p.m. indication. By setting `LC_ALL` to `C`, we can get the traditional Unix output:

```
$ LC_ALL=C date +'It is now %A, %B %d, %Y, %I:%M %p'
It is now Sunday, May 25, 2025, 04:35 PM
```

Many of `strftime()`'s specifiers come from such existing Unix date implementations. The %n and %t formats are not strictly necessary in C, since the tab and newline characters can be directly embedded in the string. However, in the context of a date format string on the command line, they make more sense. Thus they're included in the specification for `strftime()` as well.¹

The ISO 8601 standard defines (among other things) how weeks are numbered within a year. According to this standard, weeks run Monday through Sunday, and Monday is day 1 of the week, not day 0. If the week in which January 1 comes out contains at least four days in the new year, then it is considered to be week 1. Otherwise, that week is the last week of the previous year, numbered 52 or 53. These rules are used for the computation of the %g, %G, and %V format specifiers. (While parochial Americans such as the author may find these rules strange, they are commonly used throughout Europe.)

Many of the format specifiers produce results that are specific to the current locale. In addition, several indicate that they produce the "appropriate" representation for the locale (for example, %x). The C standard defines the values for the "C" locale. These values are listed in Table 6.3.

In addition to the `strftime()` features mandated by the C standard, POSIX extends the format specifiers in a manner similar to what `printf()` allows. Between the % and the format letter, you may also provide:

- An optional flag. There are two possible values:
 - 0 Specify that the values should be padded with '0' characters.
 - + Print a leading + for positive numeric values, or a - for negative ones.

¹Modern shells provide a special type of string constant, `$'...'`, wherein certain escape sequences are interpolated into them by the shell, including newline and tab. This makes %n and %t even less useful.

In our testing, GLIBC does not support either of these flags, but things may change in the future.

- A positive numeric field width. If the formatted value is smaller than the field's width, it is padded with the default padding character. On GLIBC systems, this is '0'.
- The optional E and O modifiers shown earlier.

It should be obvious that `strftime()` provides considerable flexibility and control over date- and time-related output, in much the same way as `printf()` and `sprintf()` do. Furthermore, `strftime()` cannot overflow its buffer, since it checks against the passed-in size parameter, making it a safer routine than is `sprintf()`.

As a simple example, consider the creation of program log files, when a new file is created every hour. The file name should embed the date and time of its creation in its name:

```
/* Error checking omitted for brevity */
char fname[PATH_MAX];      /* PATH_MAX is in <limits.h> */
time_t now;
struct tm *tm;
int fd;

time(& now);
tm = localtime(& now);
strftime(fname, sizeof fname, "/var/log/myapp.%Y-%m-%d-%H:%M", tm);
fd = creat(name, 0600);
...
```

The year-month-day-hour-minute format causes the file names to sort in the order they were created.

NOTE

Some time formats are more useful than others. For example, 12-hour times are ambiguous, as are any purely numeric date formats. (What does '9/11' mean? It depends on where you live.) Similarly, two-digit years are also a bad idea. Use `strftime()` judiciously.

6.1.4 Converting a Broken-Down Time to a `time_t`

Obtaining seconds-since-the-EPOCH values from the system is easy; that's how date and times are stored in inodes and returned from `time()` and `stat()`. These values are also easy to compare for equality or by `<` and `>` for simple earlier-than/later-than tests.

However, dates entered by humans are not so easy to work with. For example, many versions of the `touch` command allow you to provide a date and time to which `touch` should set a file's modification or access time (with `utimensat()`, as described in Section 5.6.4, "Changing Timestamps: `utime()` and Successors," page 148).

Converting a date as entered by a person into a `time_t` value is difficult: leap years must be taken into account, time zones must be compensated for, and so on. Therefore, the C90 standard introduced the `mktime()` function:

```
#include <time.h> ISO C

time_t mktime(struct tm *tm);
```

To use `mktime()`, fill in a `struct tm` with appropriate values: year, month, day, and so on. If you know whether daylight saving time was in effect for the given date, set the `tm_isdst` field appropriately: 0 for “no,” and positive for “yes.” Otherwise, use a negative value for “don’t know.” The `tm_wday` and `tm_yday` fields are ignored.

`mktime()` assumes that the `struct tm` represents a local time, not UTC. It returns a `time_t` value representing the passed-in date and time, or it returns (`time_t`) -1 if the given date/time cannot be represented correctly. Upon a successful return, all the values in the `struct tm` are adjusted to be within the correct ranges, and `tm_wday` and `tm_yday` are set correctly as well. Here is a simple example:

```
1 /* ch-general1-echodate.c --- demonstrate mktime(). */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <time.h>
6
7 int
8 main(void)
9 {
10     struct tm tm;
11     time_t then;
12
13     printf("Enter a Date/time as YYYY/MM/DD HH:MM:SS : ");
14     scanf("%d/%d/%d %d:%d:%d",
15         & tm.tm_year, & tm.tm_mon, & tm.tm_mday,
16         & tm.tm_hour, & tm.tm_min, & tm.tm_sec);
17
18     /* Error checking on values omitted for brevity. */
19     tm.tm_year -= 1900;
20     tm.tm_mon--;
21
22     tm.tm_isdst = -1;      /* Don't know about DST */
23
24     then = mktime(& tm);
25
26     printf("Got: %s", ctime(& then));
27     exit(EXIT_SUCCESS);
28 }
```

Line 13 prompts for a date and time, and lines 14–16 read it in. (Production code should check the return value from `scanf()`. `mktime()` checks that the values it receives are within sane ranges.) Lines 19 and 20 compensate for the different basing of years and months, respectively. Line 24 indicates that we don't know whether the given date and time represent daylight saving time. Line 24 calls `mktime()`, and line 26 prints the result of the conversion. When compiled and run, we see that it works:

```
$ ch-general11-echodate
Enter a Date/time as YYYY/MM/DD HH:MM:SS : 2025/07/10 12:44:55
Got: Thu Jul 10 12:44:55 2025
```

Negative Times

In general, the `time_t` is a signed integer, making it possible to have “negative” times, or values representing times that occurred before the Epoch of January 1, 1970.

This presents a problem for `mktime()`, which can return `-1` upon an error. The C24 standard indicates how to distinguish the two cases:

[on error] the function returns the value `(time_t)(-1)` and does not change the value of the `tm_wday` component of the structure.

By first setting `tm_wday` to a value that is out of range (less than zero or greater than six), and then checking if it changed, you can tell if the call to `mktime()` succeeded.² The `mktime(3)` manpage provides similar advice.

6.1.5 Parsing a Date and Time into a struct `tm`

With `strftime()`, we turn a struct `tm` into a human-readable string. POSIX provides an additional routine to go in the opposite direction: taking a string and turning it back into a struct `tm`. This lets you avoid the pain of manually parsing a date to get a struct `tm`. The routine is `strptime()`:

```
#define _XOPEN_SOURCE          GLIBC
#include <time.h>              POSIX XSI

char *strptime(const char *buf, const char *format, struct tm *tm);
```

The parameters are:

```
const char *buf
```

The text of the date string to be parsed.

```
const char *format
```

A format string describing the contents of `buf`.

²Thanks to Geoff Clare in `comp.lang.awk` for this tip.

```
struct tm *tm
```

A pointer to a struct `tm` in which to place the parsed values.

Upon error the return value is `NULL`. Otherwise, the return value points to the first character that was not parsed. Here is a simple example program:

```
/* ch-general1-parsetime.c --- Adapted from POSIX strptime() description. */

#define _XOPEN_SOURCE          /* Needed on GLIBC, must be before all includes! */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <time.h>

int
main(int argc, char **argv)
{
    struct tm tm;
    time_t t;

    if (strptime("15 Dec 2025 18:07:53", "%d %b %Y %H:%M:%S", &tm) == NULL) {
        fprintf(stderr, "%s: strptime failed: %s\n", argv[0], strerror(errno));
        exit(EXIT_FAILURE);
    }

    printf("year: %d; month: %d; day: %d;\n",
           tm.tm_year + 1900, tm.tm_mon, tm.tm_mday);
    printf("hour: %d; minute: %d; second: %d\n",
           tm.tm_hour, tm.tm_min, tm.tm_sec);
    printf("week day: %d; year day: %d\n", tm.tm_wday, tm.tm_yday);

    /* Tell mktime() to determine if daylight saving time is in effect */
    tm.tm_isdst = -1;
    t = mktime(&tm);
    if (t == -1) {
        fprintf(stderr, "%s: mktime failed: %s\n", argv[0], strerror(errno));
        exit(EXIT_FAILURE);
    }
    printf("Seconds since the Epoch: %ld\n", (long) t);
    return EXIT_SUCCESS;
}
```

When run, it produces this output:

```
$ ch-general1-parsetime
year: 2025; month: 11; day: 15;
```

```
hour: 18; minute: 7; second: 53
week day: 1; year day: 348
Seconds since the Epoch: 1765814873
```

On GLIBC systems, in order to access the declaration of `strptime()`, you must define `_XOPEN_SOURCE` *before* including *any* standard header files.

The conversion specifiers available for use in `format` are purposely the same as those for `strftime()`; see Table 6.2. GLIBC provides an additional value, `%s`, which scans a seconds-since-the-Epoch value.

Additionally, `%a` and `%A` are equivalent to each other, as are `%b` and `%B`; `%n` and `%t` match any whitespace characters.

POSIX allows optional flags of `0` or `+` that are to be ignored, and an optional field width, neither of which are mentioned by the Linux `strptime(3)` manpage. The optional `E` and `O` modifiers are allowed.

6.1.6 Getting Time-Zone Information

Early Unix systems embedded time-zone information into the kernel when it was compiled. The rules for daylight saving time conversions were generally hard-coded, which was painful for users outside the United States or in places within the United States that didn't observe DST.

Modern systems have abstracted that information into binary files read by the C library when time-related functions are invoked. This technique avoids the need to recompile libraries and system executables when the rules in a particular location change and makes it much easier to update the rules.

The C language interface to time-zone information evolved across different Unix versions, both System V and Berkeley, until finally it was standardized by POSIX as follows:

```
#include <time.h>                                POSIX

extern char *tzname[2];
extern long timezone;                             POSIX XSI
extern int daylight;                              POSIX XSI

void tzset(void);
```

The `tzset()` function examines the `TZ` environment variable to find time-zone and daylight saving time information.³ If that variable isn't set, then `tzset()` uses an “implementation-defined default time zone,” which is most likely the time zone of the machine you're running on.

After `tzset()` has been called, the local time-zone information is available in several variables:

³Although POSIX standardizes `TZ`'s format, it isn't all that interesting, so we haven't bothered to document it here. After all, it is `tzset()` that has to understand the format, not user-level code. Implementations can, and do, use formats that extend POSIX.

```
extern char *tzname[2]
```

The standard and daylight saving time names for the time zone. For example, for U.S. locations in the Eastern time zone, the time-zone names are "EST" (Eastern Standard Time) and "EDT" (Eastern Daylight Time).

```
extern long timezone
```

The difference, in seconds, between the current time zone and UTC. The standard does not explain how this difference works. In practice, negative values represent time zones *east* of (ahead of, or later than) UTC, and positive values represent time zones *west* of (behind, or earlier than) UTC. If you look at this value as “how much to change the local time to make it be the same as UTC,” then the sign of the value makes sense.

```
extern int daylight
```

This variable is zero if daylight saving time conversions should never be applied in the current time zone, and nonzero otherwise.

NOTE

The `daylight` variable does *not* indicate whether daylight saving time is currently in effect! Instead, it merely states whether the current time zone can even have daylight saving time.

The POSIX standard indicates that `ctime()`, `localtime()`, `mktime()`, and `strftime()` all act “as if” they call `tzset()`. This means that they need not actually call `tzset()`, but they must behave as if it had been called. (The wording is intended to provide a certain amount of flexibility for implementors while guaranteeing correct behavior for user-level code.)

In practice, this means that you will almost never have to call `tzset()` yourself. However, it’s there if you need it.

Local Time: How Does It Know?

GNU/Linux systems store time-zone information in files and directories underneath `/usr/share/zoneinfo`:

```
$ cd /usr/share/zoneinfo
```

```
$ ls -FC
```

Africa/	America/	Antarctica/	Arctic/
Asia/	Atlantic/	Australia/	Brazil/
CET	CST6CDT	Canada/	Chile/
Cuba@	EET	EST	EST5EDT
Egypt@	Eire@	Etc/	Europe/
Factory	GB-Eire@	GB@	GMT+0@
GMT-0@	GMT0@	GMT@	Greenwich@
HST	Hongkong@	Iceland@	Indian/
Iran@	Israel@	Jamaica@	Japan@
Kwajalein@	Libya@	MET	MST

MST7MDT	Mexico/	NZ-CHAT@	NZ@
Navajo@	PRC@	PST8PDT	Pacific/
Poland@	Portugal@	ROC@	ROK@
Singapore@	Turkey@	UCT@	US/
UTC@	Universal@	W-SU@	WET
Zulu@	iso3166.tab	leap-seconds.list	leapseconds
localtime@	posix/	posixrules@	right/
tzdata.zi	zone.tab	zone1970.tab	

Part of the process of installing a system is to choose the time zone. The correct time-zone data file is then placed in `/etc/localtime`:

```
$ file /etc/localtime
/etc/localtime: symbolic link to /usr/share/zoneinfo/Asia/Jerusalem
```

On our system, this is a symbolic link to a file in `/usr/share/zoneinfo`. On other systems, it may be a stand-alone copy of the time-zone file for the time zone. The advantage of using a separate copy is that everything still works if `/usr` isn't mounted.

The `TZ` environment variable, if set, overrides the default time zone:

```
$ date                               Date and time in default time zone
Sun May 25 09:47:59 AM EDT 2025

$ export TZ=PST8PDT                  Change time zone to U.S. West Coast
$ date                               Print date and time
Sun May 25 06:48:14 AM PDT 2025
```

6.1.6.1 BSD Systems Gotcha: `timezone()`, Not `timezone`

Instead of the POSIX `timezone` variable, several systems derived from 4.4 BSD provide a `timezone()` function:

```
#include <time.h>                                BSD

char *timezone(int zone, int dst);
```

The `zone` argument is the number of *minutes* west of GMT, and `dst` is true if daylight saving time is in effect. The return value is a string giving the name of the indicated zone, or a value expressed relative to GMT. This function provides compatibility with the V7 function of the same name and behavior.

This function's existence makes portable use of the POSIX `timezone` variable difficult. Fortunately, we don't see a huge need for it: `strftime()` should be sufficient for all but the most unusual needs.

6.2 Sorting and Searching Functions

Sorting and searching are two fundamental operations for which a need arises continually in many applications. The C library provides a number of standard interfaces for performing these tasks. They are general purpose in nature, suitable for working with any kind of data, as opposed to being tuned for particular applications.

All the routines share a common theme: data is managed through `void *` pointers, and user-provided functions supply ordering. Note also that these APIs apply to *in-memory* data. Sorting and searching structures in files is considerably more involved and beyond the scope of an introductory text such as this one. (However, the `sort` command works well for text files; see the `sort(1)` manpage. Sorting binary files requires that a special-purpose program be written.)

Because no one algorithm works well for all applications, there are several different sets of library routines for maintaining searchable collections of data. This chapter covers only one simple interface for searching. A more advanced interface is described in Section 16.4, “Advanced Searching with Binary Trees,” page 575. Furthermore, we purposely don’t explain the underlying algorithms, since this is a book on APIs, not algorithms and data structures. What’s important to understand is that you can treat the APIs as “black boxes” that do a particular job without needing to understand the details of *how* they do the job.

6.2.1 Sorting: `qsort()`

Sorting is accomplished with `qsort()`:

```
#include <stdlib.h> ISO C  
  
void qsort(void *base, size_t nmem, size_t size,  
           int (*compare)(const void *, const void *));
```

The name `qsort()` comes from C. A. R. Hoare’s Quicksort algorithm, which was used in the initial Unix implementation. (Nothing in the POSIX standard dictates the use of this algorithm for `qsort()`. The GLIBC implementation uses a highly optimized combination of Quicksort and Insertion Sort.)

`qsort()` sorts arrays of arbitrary objects. It works by shuffling opaque chunks of memory from one spot within an array to another and relies on you, the programmer, to provide a comparison function that allows it to determine the ordering of one array element relative to another. The arguments are as follows:

`void *base`
The address of the beginning of the array.

`size_t nmem`
The total number of elements in the array.

`size_t size`

The size of each element in the array. The best way to obtain this value is with the C `sizeof` operator.

`int (*compare)(const void *, const void *)`

A possibly scary declaration for a *function pointer*. It says that “compare points to a function that takes two ‘const void *’ parameters, and returns an int.”

Most of the work is in writing a proper comparison function. The return value should mimic that of `strcmp()`: less than zero if the first value is “less than” the second, zero if they are equal, and greater than zero if the first value is “greater than” the second. It is the comparison function that defines the meaning of “less than” and “greater than” for whatever it is you’re sorting. For example, to compare two `double` values, we could use this function:

```
int
dcomp(const void *d1p, const void *d2p)
{
    const double *d1, *d2;

    d1 = (const double *) d1p;           Cast pointers to right type
    d2 = (const double *) d2p;

    if (*d1 < *d2)                       Compare and return right value
        return -1;
    else if (*d1 > *d2)
        return 1;
    else if (*d1 == *d2)
        return 0
    else
        return -1;    /* NaN sorts before real numbers */
}
```

This shows the general boilerplate for a comparison function: convert the arguments from `void *` to pointers to the type being compared and then return a comparison value.

For floating-point values, a simple subtraction such as ‘`return *d1 - *d2`’ doesn’t work, particularly if one value is very small or if one or both values are special “not a number” or “infinity” values. Thus we have to do the comparison manually, including taking into account the not-a-number value (which doesn’t even compare equal to itself!).

6.2.1.1 Example: Sorting Employees

For more complicated structures, a more involved function is necessary. For example, consider the following (rather trivial) `struct employee`:

```
struct employee {
    char lastname[30];
    char firstname[30];
};
```

```

    long emp_id;
    time_t start_date;
};

```

We might write a function to sort employees by last name, first name, and ID number:

```

int
emp_name_id_compare(const void *e1p, const void *e2p)
{
    const struct employee *e1, *e2;
    int last, first;

    e1 = (const struct employee *) e1p;           Convert pointers
    e2 = (const struct employee *) e2p;

    if ((last = strcmp(e1->lastname, e2->lastname)) != 0)   Compare last names
        return last;                                       Last names differ

    /* same last name, check first name */
    if ((first = strcmp(e1->firstname, e2->firstname)) != 0) Compare first names
        return first;                                     First names differ

    /* same first name, check ID numbers */
    if (e1->emp_id < e2->emp_id)                             Compare employee ID
        return -1;
    else if (e1->emp_id == e2->emp_id)
        return 0;
    else
        return 1;
}

```

The logic here is straightforward, initially comparing on last names, then first names, and then using the employee ID number if the two names are the same. By using `strcmp()` on strings, we automatically get the right kind of negative/zero/positive value to return.

The employee ID comparison can't just use subtraction: suppose `long` is 64 bits and `int` is 32 bits, and the two values differ only in the upper 32 bits (say the lower 32 bits are zero). In such a case, the subtraction result would automatically be cast to `int`, throwing away the upper 32 bits and returning an incorrect value.

NOTE

We could have stopped with the comparison on first names, in which case all employees with the same last and first names would be grouped, but *without any other ordering*.

This point is important: `qsort()` does not guarantee a *stable* sort. A stable sort is one in which, if two elements compare equal based on some key value(s), they will maintain their original ordering, relative to each other, in the final sorted array. For example, consider three employees with the same first and last names and with employee numbers 17, 42, and 81.

Their order in the original array might have been 42, 81, and 17 (meaning that employee 42 is at a lower index than employee 81, who, in turn, is at a lower index than employee 17). After sorting, the order might be 81, 42, and 17. If this is an issue, then the comparison routine must take *all* important key values into consideration. (Ours does.)

Simply by using a different function, we can sort employees by seniority:

```
int
emp_seniority_compare(const void *e1p, const void *e2p)
{
    const struct employee *e1, *e2;
    double diff;

    e1 = (const struct employee *) e1p;           Cast pointers to correct type
    e2 = (const struct employee *) e2p;

    diff = difftime(e1->start_date, e2->start_date); Compare times
    if (diff < 0)
        return -1;
    else if (diff > 0)
        return 1;
    else
        return 0;
}
```

For maximum portability we have used `difftime()`, which returns the difference in seconds between two `time_t` values. For this specific case, a cast such as

```
return (int) difftime(e1->start_date, e2->start_date);
```

should do the trick, since `time_t` values are within reasonable ranges. Nevertheless, we instead use a full three-way `if` statement, just to be safe.

Here is a sample data file, listing nine U.S. presidents:

```
$ cat presdata.txt
Trump Donald 47 1737396000           Last name, First name, President number, Inauguration
Biden Joseph 46 1611165600
Trump Donald 45 1484935200
Obama Barack 44 1232474400
Bush George 43 980013600
Clinton William 42 727552800
Bush George 41 601322400
Reagan Ronald 40 348861600
Carter James 39 222631200
```

`ch-general1-sortemp.c` shows a simple program that reads this file into a struct employee array and then sorts it, using the two different comparison functions just presented:

```
1 /* ch-general1-sortemp.c --- Demonstrate qsort() with two comparison functions. */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <time.h>
7
8 struct employee {
9     char lastname[30];
10    char firstname[30];
11    long emp_id;
12    time_t start_date;
13 };
14
15 /* emp_name_id_compare --- compare by name, then by ID */
16
17 int
18 emp_name_id_compare(const void *e1p, const void *e2p)
19 {
20     ... as shown previously, omitted to save space ...
21 }
22
23 /* emp_seniority_compare --- compare by seniority */
24
25 int
26 emp_seniority_compare(const void *e1p, const void *e2p)
27 {
28     ... as shown previously, omitted to save space ...
29 }
30
31
32
33 /* main --- demonstrate sorting */
34
35 int
36 main(void)
37 {
38     #define NPRES 20
39     struct employee presidents[NPRES];
40     int i, npres;
41     char buf[BUFSIZ];
42
43     /* Very simple code to read data: */
44     for (npres = 0; npres < NPRES && fgets(buf, BUFSIZ, stdin) != NULL;
45         npres++) {
46         sscanf(buf, "%s %s %ld %ld\n",
47             presidents[npres].lastname,
```

```

78         presidents[npres].firstname,
79         & presidents[npres].emp_id,
80         & presidents[npres].start_date);
81     }
82
83     /* npres is now number of actual lines read. */
84
85     /* First, sort by name */
86     qsort(presidents, npres, sizeof(struct employee), emp_name_id_compare);
87
88     /* Print output */
89     printf("Sorted by name:\n");
90     for (i = 0; i < npres; i++)
91         printf("\t%s %s\t%ld\t%s",
92             presidents[i].lastname,
93             presidents[i].firstname,
94             presidents[i].emp_id,
95             ctime(& presidents[i].start_date));
96
97     /* Now, sort by seniority */
98     qsort(presidents, npres, sizeof(struct employee), emp_seniority_compare);
99
100    /* And print again */
101    printf("Sorted by seniority:\n");
102    for (i = 0; i < npres; i++)
103        printf("\t%s %s\t%ld\t%s",
104            presidents[i].lastname,
105            presidents[i].firstname,
106            presidents[i].emp_id,
107            ctime(& presidents[i].start_date));
108 }

```

Lines 74–81 read in the data. Note that *any* use of `scanf()` requires “well behaved” input data. If, for example, any name is more than 29 characters, there’s a problem. In this case, we’re safe, but production code must be considerably more careful.

Line 86 sorts the data by name and employee ID, and then lines 89–95 print the sorted data. Similarly, line 98 re-sorts the data, this time by seniority, with lines 101–107 printing the results. When compiled and run, the program produces the following results:

```
$ ch-general1-sorttemp < presdata.txt
```

```
Sorted by name:
```

```

    Biden Joseph      46      Wed Jan 20 13:00:00 2021
    Bush George       41      Fri Jan 20 13:00:00 1989
    Bush George       43      Sat Jan 20 13:00:00 2001
    Carter James      39      Thu Jan 20 13:00:00 1977
    Clinton William   42      Wed Jan 20 13:00:00 1993

```

```

Obama Barack 44 Tue Jan 20 13:00:00 2009
Reagan Ronald 40 Tue Jan 20 13:00:00 1981
Trump Donald 45 Fri Jan 20 13:00:00 2017
Trump Donald 47 Mon Jan 20 13:00:00 2025

```

Sorted by seniority:

```

Carter James 39 Thu Jan 20 13:00:00 1977
Reagan Ronald 40 Tue Jan 20 13:00:00 1981
Bush George 41 Fri Jan 20 13:00:00 1989
Clinton William 42 Wed Jan 20 13:00:00 1993
Bush George 43 Sat Jan 20 13:00:00 2001
Obama Barack 44 Tue Jan 20 13:00:00 2009
Trump Donald 45 Fri Jan 20 13:00:00 2017
Biden Joseph 46 Wed Jan 20 13:00:00 2021
Trump Donald 47 Mon Jan 20 13:00:00 2025

```

(We’ve used 1:00 p.m. as an approximation for the time when each president started working.)⁴

One point is worth mentioning: `qsort()` rearranges the data in the array. If each array element is a large structure, *a lot* of data will be copied back and forth as the array is sorted. It may pay instead to set up *a separate array of pointers*, each of which points at one element of the array, and then use `qsort()` to sort the pointer array, accessing the *unsorted* data through the *sorted* pointers.

The price paid is the extra memory to hold the pointers and modification of the comparison function to use an extra pointer indirection when comparing the structures. The benefit returned can be a considerable speedup, since only a four- or eight-byte pointer is moved around at each step, instead of a large structure. (Our `struct employee` is 68 bytes in size on a 32-bit system and 80 bytes in size on a 64-bit one. Swapping four-byte pointers moves 17 times less data than does swapping structures. Even swapping eight-byte pointers moves 10 times less data than does swapping structures.) For millions of in-memory structures, the difference can be significant, and this may be even more important if you’re developing for an embedded system with very limited memory.

NOTE

If you’re a C++ programmer, beware! `qsort()` may be dangerous to use with arrays of objects! `qsort()` does raw memory moves, copying bytes. It’s completely unaware of C++ constructs such as copy constructors or `operator=()` functions. Instead, use one of the STL sorting functions, or use the separate-array-of-pointers technique.

6.2.1.2 Example: Sorting Directory Contents

In Section 5.3, “Reading Directories,” page 123, we demonstrated that directory entries are returned in physical directory order. Most of the time, it’s much more useful to have directory

⁴The output shown here is for U.S. Eastern Standard Time. You will get different results for the same program and data if you use a different time zone.

contents sorted in some fashion, such as by name or by modification time. Several routines make it easy to do this, using `qsort()` as the underlying sorting agent:

```
#include <dirent.h>                                POSIX

int scandir(const char *dir, struct dirent ***namelist,
            int (*select)(const struct dirent *),
            int (*compare)(const struct dirent **, const struct dirent **));
int alphasort(const struct dirent **a, const struct dirent **b);

int versionsort(const struct dirent **a, const struct dirent **b);    GLIBC
```

The `scandir()` and `alphasort()` functions were first made available in 4.2 BSD. They have always been widely supported, and are now standardized by POSIX. `versionsort()` is a GNU extension.

`scandir()` reads the directory named by `dir`, creates an array of `struct dirent` pointers by using `malloc()`, and sets `*namelist` to point to the beginning of that array. Both the array of pointers and the pointed-to `struct dirent` structures are allocated with `malloc()`; it is up to the calling code to use `free()` to avoid memory leaks.

Use the `select` function pointer to choose entries of interest. When this value is `NULL`, all valid directory entries are included in the final array. Otherwise, `(*select)()` is called for each entry, and those entries for which it returns nonzero (true) are included in the array.

The `compare` function pointer compares two directory entries. It is passed to `qsort()` for use in sorting.

`alphasort()` compares file names lexicographically. It uses the `strcoll()` function for comparison. `strcoll()` is similar to `strcmp()` but takes locale-related sorting rules into consideration (see Section 15.2.3, “String Collation: `strcoll()` and `strxfrm()`,” page 512).

`versionsort()` is a GNU extension, that uses the GNU `strverscmp()` function to compare file names (see `strverscmp(3)`). To make a long story short, this function understands common file name versioning conventions and compares appropriately.

`ch-general11-sortdir.c` shows a program similar to `ch-fileinfo-catdir.c`. However, it uses `scandir()` and `alphasort()` to do the work:

```
1 /* ch-general11-sortdir.c --- Demonstrate scandir(), alphasort(). */
2
3 #include <stdio.h>                                /* for printf() etc. */
4 #include <errno.h>                                /* for errno */
5 #include <stdlib.h>                               /* for free() */
6 #include <string.h>                               /* for strerror() */
7 #include <sys/types.h>                            /* for system types */
8 #include <dirent.h>                              /* for directory functions */
9
10 char *myname;
11 int process(const char *dir);
12
13 /* main --- loop over directory arguments */
```

```
14
15 int
16 main(int argc, char **argv)
17 {
18     int i;
19     int errs = 0;
20
21     myname = argv[0];
22
23     if (argc == 1)
24         errs = process(".");    /* default to current directory */
25     else
26         for (i = 1; i < argc; i++)
27             errs += process(argv[i]);
28
29     return (errs != 0);
30 }
31
32 /* nodots --- ignore dot files, for use by scandir() */
33
34 int
35 nodots(const struct dirent *dp)
36 {
37     return (dp->d_name[0] != '.');
38 }
39
40 /*
41  * process --- do something with the directory, in this case,
42  *              print inode/name pairs on standard output.
43  *              Return 0 if all OK, 1 otherwise.
44  */
45
46 int
47 process(const char *dir)
48 {
49     DIR *dp;
50     struct dirent **entries;
51     int nents, i;
52
53     nents = scandir(dir, & entries, nodots, alphasort);
54     if (nents < 0) {
55         fprintf(stderr, "%s: scandir failed: %s\n", myname,
56                 strerror(errno));
57         return 1;
58     }
```

```

59
60     for (i = 0; i < nents; i++) {
61         printf("%8ld %s\n", entries[i]->d_ino, entries[i]->d_name);
62         free(entries[i]);
63     }
64
65     free(entries);
66
67     return 0;
68 }

```

The `main()` program (lines 1–30) follows the standard boilerplate we’ve used before. The `nodots()` function (lines 34–38) acts as the `select` parameter, choosing only file names that don’t begin with a period.

The `process()` function (lines 46–68) is quite simple, with `scandir()` doing most of the work. Note how each element is released separately with `free()` (line 62) and how the entire array is also released (line 65).

When run, the directory contents do indeed come out in sorted order, without `.` and `..`:

```

$ ch-general1-sortdir Default action displays current directory
9207429 00-preface.texi
9208690 01-intro.texi
9205291 02-cmdline.texi
9207424 03-memory.texi
9207611 04-fileio.texi
9207616 05-fileinfo.texi
9208272 06-general1.texi
...

```

6.2.2 Binary Searching: `bsearch()`

A *linear search* is pretty much what it sounds like: you start at the beginning, and check each element of the array being searched until you find what you need. For something simple like finding integers, this usually takes the form of a `for` loop. Consider this function:

```

/* ifind --- linear search, return index if found or -1 if not */

int
ifind(int x, const int array[], size_t nelems)
{
    size_t i;

    for (i = 0; i < nelems; i++)
        if (array[i] == x) /* found it */
            return i;

    return -1;
}

```

The advantage to linear searching is that it's simple; it's easy to write the code correctly the first time. Furthermore, it always works. Even if elements are added to the end of the array or removed from the array, there's no need to sort the array.

The disadvantage to linear searching is that it's slow. On average, for an array containing `nelems` elements, a linear search for a random element does '`nelems / 2`' comparisons before finding the desired element. This becomes prohibitively expensive, even on modern high-performance systems, as `nelems` becomes large. Thus, you should only use linear searching on small arrays.

Unlike a linear search, binary searching requires that the input array already be sorted. The disadvantage here is that if elements are added, the array must be re-sorted before it can be searched. (When elements are removed, the rest of the array contents must still be shuffled down. This is not as expensive as re-sorting, but it can still involve a lot of data motion.)

The advantage to binary searching—and it's a significant one—is that binary searching is blindingly fast, requiring at most $\lfloor \log_2(N) \rfloor + 1$ comparisons, where N is the number of elements in the array. The `bsearch()` function is declared as follows:

```
#include <stdlib.h> ISO C

void *bsearch(const void *key, const void *base, size_t nmemb,
              size_t size, int (*compare)(const void *, const void *));
```

The parameters and their purposes are similar to those of `qsort()`:

```
const void *key
    The object being searched for in the array.

const void *base
    The start of the array.

size_t nmemb
    The number of elements in the array.

size_t size
    The size of each element, obtained with sizeof.

int (*compare)(const void *, const void *)
    The comparison function. It must work the same way as the qsort() comparison function, returning negative/zero/positive according to whether the first parameter is less than/equal to/greater than the second one.
```

`bsearch()` returns `NULL` if the object is not found. Otherwise, it returns a pointer to the found object. If more than one array element matches `key`, it is unspecified which one is returned. Thus, as with `qsort()`, make sure that the comparison function accounts for all relevant parts of the searched data structure.

`ch-general1-searchemp.c` shows `bsearch()` in practice, extending the `struct employee` example used previously:

```

1  /* ch-general11-searchemp.c --- Demonstrate bsearch(). */
2
3  #include <stdio.h>
4  #include <errno.h>
5  #include <string.h>
6  #include <stdlib.h>
7  #include <time.h>
8
9  struct employee {
10     char lastname[30];
11     char firstname[30];
12     long emp_id;
13     time_t start_date;
14 };
15
16 /* emp_id_compare --- compare by ID */
17
18 int
19 emp_id_compare(const void *e1p, const void *e2p)
20 {
21     const struct employee *e1, *e2;
22
23     e1 = (const struct employee *) e1p;
24     e2 = (const struct employee *) e2p;
25
26     if (e1->emp_id < e2->emp_id)
27         return -1;
28     else if (e1->emp_id == e2->emp_id)
29         return 0;
30     else
31         return 1;
32 }
33
34 /* print_employee --- print an employee structure */
35
36 void
37 print_employee(const struct employee *emp)
38 {
39     printf("%s %s\t%d\t%s", emp->lastname, emp->firstname,
40           emp->emp_id, ctime(& emp->start_date));
41 }

```

Lines 9–14 define the struct `employee`; it's the same as before. Lines 18–32 serve as the comparison function, for both `qsort()` and `bsearch()`. It compares on employee ID number only. Lines 36–41 define `print_employee()`, which is a convenience function for printing the structure, since this is done from multiple places. The code continues:

```
43 /* main --- demonstrate sorting */
44
45 int
46 main(int argc, char **argv)
47 {
48     #define NPRES 20
49     struct employee presidents[NPRES];
50     int i, npres;
51     char buf[BUFSIZ];
52     struct employee *the_pres;
53     struct employee key;
54     int id;
55     FILE *fp;
56
57     if (argc != 2) {
58         fprintf(stderr, "usage: %s datafile\n", argv[0]);
59         exit(1);
60     }
61
62     if ((fp = fopen(argv[1], "r")) == NULL) {
63         fprintf(stderr, "%s: %s: could not open: %s\n", argv[0],
64                 argv[1], strerror(errno));
65         exit(1);
66     }
67
68     /* Very simple code to read data: */
69     for (npres = 0; npres < NPRES && fgets(buf, BUFSIZ, fp) != NULL;
70         npres++) {
71         sscanf(buf, "%s %s %ld %ld",
72                presidents[npres].lastname,
73                presidents[npres].firstname,
74                & presidents[npres].emp_id,
75                & presidents[npres].start_date);
76     }
77     fclose(fp);
78
79     /* npres is now number of actual lines read. */
80
81     /* First, sort by id */
82     qsort(presidents, npres, sizeof(struct employee), emp_id_compare);
83
84     /* Print output */
85     printf("Sorted by ID:\n");
86     for (i = 0; i < npres; i++) {
87         putchar('\t');
```

```

88     print_employee(& presidents[i]);
89 }
90
91 for (;;) {
92     printf("Enter ID number: ");
93     if (fgets(buf, BUFSIZ, stdin) == NULL)
94         break;
95
96     sscanf(buf, "%d\n", & id);
97     key.emp_id = id;
98     the_pres = (struct employee *) bsearch(& key, presidents, npres,
99         sizeof(struct employee), emp_id_compare);
100
101     if (the_pres != NULL) {
102         printf("Found: ");
103         print_employee(the_pres);
104     } else
105         printf("Employee with ID %d not found!\n", id);
106 }
107
108 putchar('\n'); /* Print a newline on EOF. */
109
110 exit(0);
111 }

```

The `main()` function starts with argument checking (lines 57–60). It then reads the data from the named file (lines 68–77). Standard input cannot be used for the employee data, since that is reserved for prompting the user for the employee ID to search for.

Lines 82–89 sort and print the employees. The program then goes into a loop, starting on line 91. It prompts for an employee ID number, exiting the loop upon end-of-file. To search the array, we use the `struct employee` named `key`. It's enough to set just its `emp_id` field to the entered ID number; none of the other fields are used in the comparison (line 97).

If an entry is found with the matching key, `bsearch()` returns a pointer to it. Otherwise it returns `NULL`. The return is tested on line 101, and appropriate action is then taken. Finally, line 108 prints a newline character so that the system prompt will come out on a fresh line. Here's a transcript of what happens when the program is compiled and run:

```
$ ch-general1-searchemp presdata.txt
```

Run the program

```
Sorted by ID:
```

Carter James	39	Thu Jan 20 13:00:00 1977
Reagan Ronald	40	Tue Jan 20 13:00:00 1981
Bush George	41	Fri Jan 20 13:00:00 1989
Clinton William	42	Wed Jan 20 13:00:00 1993
Bush George	43	Sat Jan 20 13:00:00 2001
Obama Barack	44	Tue Jan 20 13:00:00 2009
Trump Donald	45	Fri Jan 20 13:00:00 2017

Biden Joseph	46	Wed Jan 20 13:00:00 2021	
Trump Donald	47	Mon Jan 20 13:00:00 2025	
Enter ID number: 45			<i>Enter a valid number</i>
Found: Trump Donald	45	Fri Jan 20 13:00:00 2017	<i>It's found</i>
Enter ID number: 29			<i>Enter an invalid number</i>
Employee with ID 29 not found!			<i>It's not found</i>
Enter ID number: 40			<i>Try another good one</i>
Found: Reagan Ronald	40	Tue Jan 20 13:00:00 1981	<i>This one is found too</i>
Enter ID number: ^D			<i>CTRL-D entered for EOF</i>
\$			<i>Ready for next command</i>

Additional, more advanced APIs for searching data collections are described in Section 16.4, “Advanced Searching with Binary Trees,” page 575.

6.2.2.1 Example: Sorting and Searching Together

Sorting data and searching it go together. We next present an example from real-world code, drawn this time from the current version of the original Unix `awk` program.⁵ The code comes from that portion of `awk` that matches regular expressions.

By way of introduction, regular expression matching in `awk` uses a technique based on automata theory. In particular, it uses a Deterministic Finite Automaton, or DFA. The DFA matcher starts out in an initial state. Based on the input characters, it moves from one state to the next until it arrives at the terminating state or determines that it cannot do so. In the former case, the regular expression it represents has matched the input text; in the latter, it has not. Figure 6.2 shows the transitions among states for the regular expression `'(a|b)c'`.

State 1 is the initial state, and State 4 is the final state. The figure is very simplistic. From any given state, it may be possible to move to multiple following states, based on the current input character. This is termed a *state transition*.

In the early days of computing, when characters were always a single byte in size, the representation of a state transition was very simple. One could use a two-dimensional table,

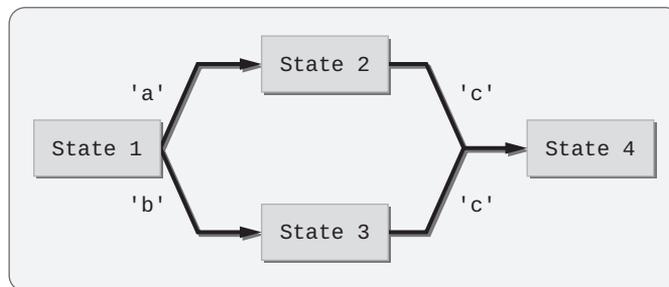


Figure 6.2: A very simple DFA

⁵See the code on GitHub (<https://github.com/onetrueawk/awk>) if you're interested.

where the row is indexed by the current state and the column is indexed by the current input character. The value stored in the table would indicate the next state to move to:

```
next_state = transition[current_state][current_char];
```

In such a representation, each row took up a maximum of 256 entries. And indeed, `awk` used such a representation:

```
...
if ((ns = f->gototab[s][*p]) != 0)
    s = ns;
...
```

Here, `f` is the DFA, `ns` is the next state, `s` is the current state, and `*p` is the current input character. `gototab` is short for “goto table”: a table indicating which state to go to next.

The problem is that in today’s world, characters are no longer just a single byte in size. In particular, the Unicode character set has code points (character values) that are normally represented with at least four bytes, and are encoded in files using one to four bytes. (There’s more on this later in the book; see Section 15.4.2, “Multibyte Character Encodings,” page 545.) There are well over a million valid Unicode code points—1,114,112 to be exact—thus it’s not practical for each row in a transition table to have over a million entries.

To solve this problem, Brian Kernighan moved the goto table into a different structure, accessed via functions. The data structure for a `gototab` consisted of a “goto table entry”:

```
typedef struct gtt { /* gototab entry */
    unsigned int ch;           The character of interest
    unsigned int state;       The state to move to
} gtt;
```

Initially, he left the size of each row at (approximately) 256 entries, and he used a linear search within the row to find the current input character, and from there return the next state. The function is named `get_gototab()`:

```
static int get_gototab(fa *f, int state, int ch) /* hide gototab implementation */
{
    int i;
    for (i = 0; i < f->gototab_len; i++) {
        if (f->gototab[state][i].ch == 0)
            break;
        if (f->gototab[state][i].ch == ch)
            return f->gototab[state][i].state;
    }
    return 0;
}
```

During the matching, if input characters are not found in the `gototab`, they and their next state are added to it.⁶ This happens in `set_gototab()`:

⁶We don’t understand why this is the case; the code is *very* opaque. We choose here to explain what the code does, without getting into why it works the way it does.

```

static int set_gototab(fa *f, int state, int ch, int val) /* hide gototab implementation */
{
    int i;
    for (i = 0; i < f->gototab_len; i++) {
        if (f->gototab[state][i].ch == 0 || f->gototab[state][i].ch == ch) {
            f->gototab[state][i].ch = ch;
            f->gototab[state][i].state = val;
            return val;
        }
    }
    overflo(__func__);
    return val; /* not used anywhere at the moment */
}

```

The new implementation worked, but if an input file contained many Unicode characters, `awk`'s matching became unacceptably slow, and it was possible to exceed the fixed limit on the number of entries.

Your author decided to see if these problems could be fixed without too much disruption in the code. This involved three changes:

- Dynamically growing the size of the `gototab` as needed.
- Looking up the entries in the `gototab` with a binary search.
- Keeping the entries sorted so that binary search would work.

First off, the structures got rearranged a little, in order to accommodate the dynamic sizing:

```

typedef struct gtte { /* gototab entry */
    unsigned int ch;
    unsigned int state;
} gtte;

typedef struct gtt { /* gototab */
    size_t allocated;
    size_t inuse;
    gtte *entries;
} gtt;

```

For both sorting and searching, we need a comparison function:

```

static int entry_cmp(const void *l, const void *r)
{
    const gtte *left, *right;

    left = (const gtte *) l;
    right = (const gtte *) r;

    return left->ch - right->ch;
}

```

Finding an entry uses a classic call to `bsearch()`:

```
static int get_gototab(fa *f, int state, int ch) /* hide gototab implementation */
{
    gtte key;
    gtte *item;

    key.ch = ch;
    key.state = 0; /* irrelevant */
    item = bsearch(& key, f->gototab[state].entries,
                  f->gototab[state].inuse, sizeof(gtte),
                  entry_cmp);

    if (item == NULL)
        return 0;
    else
        return item->state;
}
```

Adding a new character and state is more complicated, because the list must be kept sorted.⁷ In pseudocode, though, it's rather straightforward:

```
if the array is empty:
    add the character and state
    increment the count of entries in use
    return
else if the current character is greater than the last one in the array:
    // since it's greater, the array stays sorted
    grow the array if necessary
    add the character and state
    increment the count of entries in use
    return
else:
    binary search the list to see if we already have the current character
    if so, update the state and return

// At this point, the current character isn't in the list, so it must be added:
grow the array if necessary
add the character and state
increment the count of entries in use
sort the array
return
```

⁷There's potentially a lot of sorting happening here. Some versions of `qsort()` don't perform well on data that is almost sorted, as can happen in this case. The version in GLIBC doesn't have that problem.

With that introduction, here is the new version of `set_gototab()`:

```
static int set_gototab(fa *f, int state, int ch, int val) /* hide gototab implementation */
{
    if (f->gototab[state].inuse == 0) {
        f->gototab[state].entries[0].ch = ch;
        f->gototab[state].entries[0].state = val;
        f->gototab[state].inuse++;
        return val;
    } else if ((unsigned)ch > f->gototab[state].entries[f->gototab[state].inuse-1].ch) {
        // not seen yet, insert and return
        gtt *tab = & f->gototab[state];
        if (tab->inuse + 1 >= tab->allocated)
            resize_gototab(f, state);

        f->gototab[state].entries[f->gototab[state].inuse].ch = ch;
        f->gototab[state].entries[f->gototab[state].inuse].state = val;
        f->gototab[state].inuse++;
        return val;
    } else {
        // maybe we have it, maybe we don't
        gtte key;
        gtte *item;

        key.ch = ch;
        key.state = 0; /* irrelevant */
        item = (gtte *) bsearch(& key, f->gototab[state].entries,
                               f->gototab[state].inuse, sizeof(gtte),
                               entry_cmp);

        if (item != NULL) {
            // we have it, update state and return
            item->state = val;
            return item->state;
        }
        // otherwise, fall through to insert and reallocate.
    }

    gtt *tab = & f->gototab[state];
    if (tab->inuse + 1 >= tab->allocated)
        resize_gototab(f, state);
    f->gototab[state].entries[tab->inuse].ch = ch;
    f->gototab[state].entries[tab->inuse].state = val;
    ++tab->inuse;
}
```

```

qsort(f->gototab[state].entries,
      f->gototab[state].inuse, sizeof(gtte), entry_cmp);

return val; /* not used anywhere at the moment */
}

```

The end result is that `awk`'s regular expression matching is now both free of the previous fixed limit, and acceptably performant. Binary search saves the day!

We return to this suite of functions in Section 17.6.1.1, “Valgrind Example: `gototab` in the One True Awk,” page 642.

6.3 User and Group Names

While the operating system works with user and group ID numbers for storage of file ownership and for permission checking, humans prefer to work with user and group *names*.

Early Unix systems kept the information that mapped names to ID numbers in simple text files, `/etc/passwd` and `/etc/group`. These files still exist on modern systems, and their format is unchanged from that of V7 Unix. However, they no longer tell the complete story. Large installations with many networked hosts keep the information in *network databases*: ways of storing the information on a small number of servers that are then accessed over the network.⁸ However, this usage is *transparent* to most applications since access to the information is done through the same API as was used for retrieving the information from the text files. It is for this reason that POSIX standardizes only the APIs; the `/etc/passwd` and `/etc/group` files need not exist as such for a system to be POSIX compliant.

The APIs to the two databases are similar; most of our discussion focuses on the user database.

6.3.1 User Database

The traditional `/etc/passwd` format maintains one line per user. Each line has seven fields, each of which is separated from the next by a colon character:

```

$ grep arnold /etc/passwd
arnold:x:1000:1000:Arnold D. Robbins,,,:/home/arnold:/bin/bash

```

In order, the fields are as follows:

The user name

This is what the user types to log in. It is also what shows up for `'ls -l'` and in any other context that displays users.

⁸Common network databases include Sun Microsystems' Network Information Service (NIS) and NIS+, Kerberos (Hesiod), macOS DirectoryServices, and LDAP, the Lightweight Directory Access Protocol. BSD systems keep user information in on-disk databases and generate the `/etc/passwd` and `/etc/group` files automatically.

The password field

Once upon a time, this was the user’s encrypted password. Today, this field is likely to be an x (as shown), meaning that the password information is held in a different file that isn’t world-readable. This separation is a security measure; if the encrypted password isn’t available to nonprivileged users, it is much harder to “crack.”

The user ID number

This should be unique—one number per user.

The group ID number

This is the user’s initial group ID number. As is discussed later, processes have multiple groups associated with them.

The user’s real name

This is at least a first and last name. Some systems allow for comma-separated fields, for office location, phone number, and so on (again, as shown), but this is not standardized.

The login directory

This directory becomes the home directory for users when they log in (\$HOME—the default for the cd command).

The login program

The program to run when the user logs in. This is usually a shell, but it need not be. If this field is left empty, the default is /bin/sh.

Access to the user database is through the routines declared in `<pwd.h>`:

```
#include <pwd.h>                                POSIX

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);

struct passwd *getpwent(void);                  POSIX XSI
void setpwent(void);
void endpwent(void);
```

The fields in the `struct passwd` used by the various API routines correspond directly to the fields in the password file:

```
struct passwd {
    char    *pw_name;        /* user name */
    char    *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user id */
    gid_t   pw_gid;        /* group id */
    char    *pw_gecos;     /* real name */
    char    *pw_dir;       /* home directory */
    char    *pw_shell;     /* shell program */
};
```

(The name `pw_gecos` is historical; when the early Unix systems were being developed, this field held the corresponding information for the user’s account on the Bell Labs Honeywell systems running the GECOS operating system.)

The purpose of each routine is described in the following list:

`struct passwd *getpwent(void)`

Return a pointer to an internal static `struct passwd` structure containing the “current” user’s information. This routine reads through the entire password database one record at a time, returning a pointer to a structure for each user. The same pointer is returned each time; that is, the internal `struct passwd` is overwritten for each user’s entry. When `getpwent()` reaches the end of the password database, it returns `NULL`. Thus it lets you step through the entire database, one user at a time. The order in which records are returned is undefined.

`void setpwent(void)`

Reset the internal state such that the next call to `getpwent()` returns the first record in the password database.

`void endpwent(void)`

“Close the database,” so to speak, be it a simple file, network connection, or something else.

`struct passwd *getpwnam(const char *name)`

Look up the user with a `pw_name` member equal to `name`, returning a pointer to a static `struct passwd` describing the user or `NULL` if the user is not found.

`struct passwd *getpwuid(uid_t uid)`

Similarly, look up the user with the user ID number given by `uid`, returning a pointer to a static `struct passwd` describing the user or `NULL` if the user is not found.

`getpwuid()` is what’s needed when you have a user ID number (such as from a `struct stat`) and you wish to print the corresponding user name. It’s also useful if you want to look up your own information, such as home directory or login shell, based on the return value of `getuid()`. (`getuid()` is presented later, in Section 11.2, “Retrieving User and Group IDs,” page 385.) `getpwnam()` converts a name to a user ID number, for example, if you wish to use `chown()` or `fchown()` on a file. In theory, both of these routines do a linear search through the password database to find the desired information. This is true in practice when a password file is used; however, behind-the-scenes databases (network or otherwise, as on BSD systems) tend to use more efficient methods of storage, so these calls are possibly not as expensive in such cases.⁹

`getpwent()` is useful when you need to go through the entire password database. For instance, you might wish to read it all into memory, sort it, and then search it quickly with `bsearch()`. This is very useful for avoiding the multiple linear searches inherent in looking things up one at a time with `getpwuid()` or `getpwnam()`.

⁹Unfortunately, if performance is an issue, there’s no standard way to know how your library does things, and indeed, the way it works can vary at runtime! (See the `nsswitch.conf(5)` manpage on a GNU/Linux system.) On the other hand, the point of the API is, after all, to hide the details.

NOTE

The pointers returned by `getpwent()`, `getpwnam()`, and `getpwuid()` all point to internal static data. Thus, you should make a copy of their contents if you need to save the information.

Take a good look at the `struct passwd` definition. The members that represent character strings are pointers; they too point at internal static data, and if you're going to copy the structure, make sure to copy the data each member points to as well.

6.3.2 Group Database

The format of the `/etc/group` group database is similar to that of `/etc/passwd`, but with fewer fields:

```
$ grep arnold /etc/group | sort
adm:x:4:syslog,arnold,miriam
arnold:x:1000:
audio:x:29:pulse,arnold,miriam
cdrom:x:24:arnold,miriam,videos
dialout:x:20:miriam,arnold,videos
...
```

Again, there is one line per group, with fields separated by colons. The fields are as follows:

The group name

This is the name of the group, as shown in ‘`ls -l`’ or in any other context in which a group name is needed.

The group password

This field is historical. It is no longer used.

The group ID number

As with the user ID, this should be unique to each group.

The user list

This is a comma-separated list of users who are members of the group.

In the previous example, we see that user `arnold` is a member of multiple groups. This membership is reflected in practice in what is termed the *group set*. Besides the main user ID and group ID number that processes have, the group set is a set of additional group ID numbers that each process carries around with it. The system checks all of these group ID numbers against a file's group ID number when performing permission checking. This subject is discussed in more detail in Chapter 11, “Permissions and User and Group ID Numbers,” page 383.

The group database APIs are similar to those for the user database. The following functions are declared in `<grp.h>`:

```
#include <grp.h>
```

POSIX

```
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

```
struct group *getgrent(void);
void setgrent(void);
void endgrent(void);
```

POSIX XSI

The `struct group` corresponds to the records in `/etc/group`:

```
struct group {
    char    *gr_name;        /* group name */
    char    *gr_passwd;     /* group password */
    gid_t   gr_gid;        /* group id */
    char    **gr_mem;       /* group members */
};
```

The `gr_mem` field bears some explanation. While declared as a pointer to a pointer (`char **`), it is best thought of as an array of strings (like `argv`). The last element in the array is set to `NULL`. When no members are listed, the first element in the array is `NULL`.

`ch-general1-groupinfo.c` demonstrates how to use the `struct group` and the `gr_mem` field. The program accepts a single user name on the command line and prints all group records in which that user name appears:

```
1  /* ch-general1-groupinfo.c --- Demonstrate getgrent() and struct group */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <grp.h>
7
8  extern void print_group(const struct group *gr);
9
10 /* main --- print group lines for user named in argv[1] */
11
12 int
13 main(int argc, char **argv)
14 {
15     struct group *gr;
16     int i;
17
18     if (argc != 2) {
19         fprintf(stderr, "usage: %s user\n", argv[0]);
20         exit(1);
21     }
22
```

Check arguments

```

23     while ((gr = getgrent()) != NULL)                Get each group record
24         for (i = 0; gr->gr_mem[i] != NULL; i++)      Look at each member
25             if (strcmp(gr->gr_mem[i], argv[1]) == 0) If found the user ...
26                 print_group(gr);                    Print the record
27
28     endgrent();
29
30     exit(0);
31 }

```

The `main()` routine first does error checking (lines 18–21). The heart of the program is a nested loop. The outer loop (line 23) loops over all the group database records. The inner loop (line 24) loops over the members of the `gr_mem` array. If one of the members matches the name from the command line (line 25), then `print_group()` is called to print the record (line 26). Here is `print_group()`:

```

33 /* print_group --- print a group record */
34
35 void
36 print_group(const struct group *gr)
37 {
38     int i;
39
40     printf("%s:%s:%ld:", gr->gr_name, gr->gr_passwd, (long) gr->gr_gid);
41
42     for (i = 0; gr->gr_mem[i] != NULL; i++) {
43         printf("%s", gr->gr_mem[i]);
44         if (gr->gr_mem[i+1] != NULL)
45             putchar(',');
46     }
47
48     putchar('\n');
49 }

```

The `print_group()` function (lines 35–49) is straightforward, with logic similar to that of `main()` for printing the member list. Group list members are comma separated; thus the loop body has to check that the *next* element in the array is not `NULL` before printing a comma. This code works correctly, even if there are no members in the group. However, for this program, we know there are members, or `print_group()` wouldn't have been called! Here's what happens when the program is run:

```

$ ch-general11-groupinfo arnold | sort
adm:x:4:syslog,arnold,miriam
audio:x:29:pulse,arnold,miriam
cdrom:x:24:arnold,miriam,videos
dialout:x:20:miriam,arnold,videos
...

```

Interestingly, the line `'arnold:x:1000:'` is missing. This is because the group member list is empty, and our program examines only the members of that list.

6.4 Terminals: `isatty()`

The Linux/Unix standard input, standard output, standard error model discourages the special treatment of input and output devices. Programs generally should not need to know, or care, whether their output is a terminal, a file, a pipe, a network connection, a physical device, or whatever.

However, there are times when a program really does need to know what kind of a file a file descriptor is associated with. The `stat()` family of calls often provides enough information: regular file, directory, device, and so on. Sometimes, though, even that is not enough, and for interactive programs in particular, you may need to know if a file descriptor represents a `tty`.

A `tty` (short for Teletype, one of the early manufacturers of computer terminals) is any device that represents a terminal—that is, something that a human would use to interact with the computer. This may be either a hardware device, such as the keyboard and monitor of a personal computer, an old-fashioned video display terminal connected to a computer by a serial line or modem, or a software *pseudoterminal*, such as is used for windowing systems and network logins.

The discrimination can be made with `isatty()`:

```
#include <unistd.h>                                POSIX

int isatty(int desc);
```

This function returns 1 if the file descriptor `desc` represents a terminal and 0 otherwise. According to POSIX, `isatty()` may set `errno` to indicate an error; thus you should set `errno` to 0 before calling `isatty()` and then check its value if the return is 0. The POSIX standard also points out that `isatty()` returning 1 doesn't mean there's a human at the other end of the file descriptor!

One place where `isatty()` comes into use is in `ls`, in which the default is to print file names in columns if the standard output is a terminal and to print them one per line if not. Another is GNU `grep`'s `--color` option, which colorizes the matching text in the output if writing to a terminal.

There are a number of system calls and library functions for dealing specifically with terminals. We don't cover them in this book for reasons of space.

6.5 Suggested Reading

1. *Mastering Algorithms with C*, by Kyle Loudon. O'Reilly, 1999. ISBN-13: 978-1-56592-453-6.

This book provides a practical, down-to-earth introduction to algorithms and data structures using C, covering hash tables, trees, sorting, and searching, among other things.

2. *The Art of Computer Programming*, vol. 3: *Sorting and Searching*, 2nd ed., by Donald E. Knuth. Addison-Wesley, 1998. ISBN-13: 978-0-201-89685-5.

This book is usually cited as the final word on sorting and searching. Bear in mind that it is considerably denser and harder to read than the Loudon book.

3. The GTK project¹⁰ consists of several libraries that work together. GTK is the underlying toolkit used by the GNU GNOME Project.¹¹ At the base of the library hierarchy is GLib, a library of fundamental types and data structures and functions for working with them. GLib includes facilities for all the basic operations we've covered so far in this book, and many more, including linked lists and hash tables. To get started, see GLib's online documentation.¹²

6.6 Summary

- Times are stored internally as `time_t` values, representing “seconds since the Epoch.” The Epoch is midnight, January 1, 1970, UTC, for GNU/Linux and Unix systems. The current time is retrieved from the system by the `time()` system call, and `difftime()` returns the difference, in seconds, between two `time_t` values.
- The `struct tm` structure represents a “broken-down time,” which is a much more usable representation of a date and time. `gmtime()` and `localtime()` convert `time_t` values into `struct tm` values, and `mktime()` goes in the opposite direction.
- `asctime()` and `ctime()` do simplistic formatting of time values, returning a pointer to a fixed-size, fixed-format `static` character string. `strftime()` provides much more flexible formatting, including locale-based values. The `strptime()` function parses a formatted date and time, filling in the members of a `struct tm`.
- Time-zone information is made available by a call to `tzset()`. Since the standard routines act as if they call `tzset()` automatically, it is rare to need to call this function directly.
- The standard routine for sorting arrays is `qsort()`. By using a user-provided comparison function and being told the number of array elements and their size, `qsort()` can sort any kind of data. This provides considerable flexibility.
- `scandir()` reads an entire directory into an array of `struct dirent`. User-provided functions can be used to select which entries to include and can provide ordering of elements within the array. `alphasort()` is a standard function for sorting directory entries by name; `scandir()` passes the comparison function straight through to `qsort()`.
- The `bsearch()` function works similarly to `qsort()`. It does fast binary searching. Use it if the cost of linear searching outweighs the cost of sorting your data. (An additional

¹⁰<https://www.gtk.org>

¹¹<https://www.gnome.org>

¹²<https://docs.gtk.org/glib/>

API for searching data collections is described in Section 16.4, “Advanced Searching with Binary Trees,” page 575.)

- The user and group databases may be kept in local disk files or may be made available over a network. The standard API purposely hides this distinction. Each database provides both linear scanning of the entire database and direct queries for a user/group name or user/group ID.
- Finally, for those times when `stat()` just isn’t enough, `isatty()` can tell you whether or not an open file represents a terminal device.

Exercises

1. Write a simple version of the `date` command that accepts a format string on the command line and uses it to format and print the current time.
2. When a file is more than six months old, `'ls -l'` uses a simpler format for printing the modification time. The GNU version of `ls.c` uses this computation:

```

4449 struct timespec six_months_ago;
      ...
4458 /* Consider a time to be recent if it is within the past six months.
4459    A Gregorian year has 365.2425 * 24 * 60 * 60 == 31556952 seconds
4460    on the average. Write this value as an integer constant to
4461    avoid floating point hassles. */
4462 six_months_ago.tv_sec = current_time.tv_sec - 31556952 / 2;
4463 six_months_ago.tv_nsec = current_time.tv_nsec;

```

Compare this to our example computation for computing the time six months in the past. What are the advantages and disadvantages of each method?

3. Write your own version of `strftime()`. Don’t worry about locale differences; simply use the English names for the months and the days of the week. Be sure not to overflow the input buffer.

Compare your version to the author’s, available on GitHub.¹³

4. Write a simple version of the `touch` command that changes the modification time of the files named on the command line to the current time.
5. Add an option to your `touch` command that accepts a date and time specification on the command line and uses that value as the new modification time of the files named on the command line. Can you use `strftime()`?
6. Add another option to your version of `touch` that takes a file name and uses the modification time of the given file as the new modification time for the files named on the command line.

¹³<https://www.github.com/arnoldrobbins/strftime>

7. Enhance `ch-general11-sorttemp.c` to sort a separate array of pointers that point into the array of employees.
8. Add options to `ch-general11-sortdir.c` to sort by inode number, modification time, access time, and size. Add a “reverse option” such that time-based sorts make the most *recent* file first and other criteria (size, inode) sort by largest value first.
9. Write a simple version of the `chown` command. Its usage should be

```
chown user[:group] files ...
```

Here, *user* and *group* are user and group names representing the new user and group for the named files. The *group* is optional; if present, it is separated from the *user* by a colon.

To test your version on a GNU/Linux system, you will have to work as `root`. Do so carefully!

10. Enhance your `chown` to allow numeric user or group numbers, as well as names.
11. Write functions to copy user and group structures, including pointed-to data. Use `malloc()` to allocate storage as needed.
12. Write a specialized user-lookup library that reads the entire user database into a dynamically allocated array. Provide *fast* lookup of users, by both user ID number and name. Be sure to handle the case in which a requested user isn’t found.
13. Do the same thing for the group database.
14. Enhance `ch-general11-groupinfo.c` to also examine the `gr_name` field of the struct `group`.
15. Write a `stat` program that prints the contents of the struct `stat` for each file named on the command line. It should print all the values in human-readable format: `time_t` values as dates and times, `uid_t` and `gid_t` values as the corresponding names (if available), and the contents of symbolic links. Print the `st_mode` field the same way that `ls` would.

Compare your program to the GNU Coreutils `stat` program, both by comparing outputs and by looking at the source code.
16. Get a cup of coffee (or tea), sit down, and read the `stty(1)` manpage, which describes the myriad options and settings for terminals. When your head stops spinning, think about the history of terminal devices, from physical teletypes through video terminals and on to windowing systems. Which of the features that are still supported today are mostly irrelevant? Which are still necessary for day-to-day use?

Index

Note: Page references in bold denote occurrences in programming.

Symbols

- (dash)
 - as filename, 80, 92, 463, 465
 - in `n1_langinfo()`, 526
 - in options, 23–24, 26, 32
 - in permissions, 4, 130
 - in regular expressions, 514
- (dash-dash)
 - in long options, 26
 - as special argument, 25
- _() macro, 531–532, 535, 537, 554
- , (comma)
 - as decimal point, 517
 - in option arguments, 26
- ;(semicolon)
 - in `getopt_long()`, 35
 - in `n1_langinfo()`, 524
- :(colon)
 - in `getopt()`, 29–31
 - in `n1_langinfo()`, 524
 - in `optstring`, 29
 - in `PATH` variable, 285
 - in regular expressions, 514, 552
- ? (question mark), in `getopt()`, 29–31
- / (forward slash), as root directory, 8, 151, 222–224, 269
- .(dot)
 - as current working directory, 8, 116, 121, 124, 127, 151, 201, 452, 515
 - as decimal point, 507, 517
 - in filenames, 452
 - in format specifiers, 521
 - in `n1_langinfo()`, 526
 - in regular expressions, 461
- .. (dot-dot), parent directory, 116, 121, 124, 127, 152, 201, 270, 452
 - in the root of a mounted filesystem, 222
- ^ (hat), in regular expressions, 461–462
- ' (single quote), in format specifiers, 521–522, 535
- () (parentheses), in regular expressions, 462–463
- [] (square brackets), in regular expressions, 463, 514
- { } (braces), in regular expressions, 463
- \$ (dollar sign)
 - as currency sign, 509, 517, 520
 - in format specifiers, 533–534
 - as prompt, xxv
 - in regular expressions, 461
- \ (backslash), for continuation lines, 66, 69
- # (hash)
 - as comment specifier, 229
 - in format specifiers, 521
 - as prompt, 227
- #!, in scripts, 6, 284

- % (per cent sign), in format specifiers, 520
- + (plus sign)
 - in format specifiers, 520
 - in `n1_langinfo()`, 526
- = (equal sign), in option arguments, 26
- > (greater-than)
 - as operator, 104
 - as prompt, xxv
- >> operator, 104
- | (vertical bar)
 - as flag separator, 614–617
 - as pipe construct, 9
- |&, in `gawk`, 323

A

- `a.out` (Assembler OUTput) format, 6, 84, 339, 643
- `abort()`, 346, 347, 356, 420, 434, 468, 595, 596, 598
- `accept()`, 347, 483, 486, 488, 495, 502, 504
- access time, 133, 201, 237
 - changing, 148–152
 - formatting, 164
 - retrieving, 150, 586
- `access()`, 347, 388–390, 402
- Acorn Advanced Disc Filing System, 226
- `action_handler()`, 354
- actions. *See* signal actions
- `adb` debugger, 593
- `addmntent()`, 231, 232
- address sanitizer, 644–647
- address space. *See* memory
- Agans, David J., xxix, 653, 655
- `aio_error()`, 347
- `aio_return()`, 347
- `aio_suspend()`, 347
- alarm clocks, 363–364, 570
- `alarm()`, 281, 347, 363–364, 378–380, 382, 470, 570, 573
- `alloca()`, 73–76, 77
 - manpage of, 74
- `alphasort()`, 178–180, 197
- “always check the return value” principle, 90, 342
- Amiga Fast File System, 226
- Andrew File System, 226
- ANSI (American National Standards Institute), xxi
- arbitrary-length lines, 65–70, 77
- archives, 149
- arg library, 29–30, 50, 323
- argc parameter, 26–32, 40, 49, 80, 205, 206, 284, 456, 464, 602
- Argp library, 50
- arguments, 23–32
 - invalid, 83
 - lists of, 83
 - missing, 30–31
 - optional, 25, 32
 - whitespace in, 23, 25–26

Argv library, 49
 argv parameter, 26–32, 37, 39, 40, 49, 194, 205, 206, 284–288, 328, 406, 410, 412, 416, 498
 arrays
 compared to trees, 575
 element count computation, 99
 searching, 180–185
 sorting, 171–180
 artificial intelligence, 19
 ASCII encoding, 541, 553
 asctime(), 159–160, 197, 209, 524
 Assembler OUTPUT. *See* a.out
 assert(), 419–423, 434, 551, 572, 607, 653
 assertions, 419–423, 468
 AT&T, xxvi, 28, 384
 atexit(), 43, 291–293, 329, 330, 442
 atoi(), 444
 atomic write, 321
 Autoconf, 14, 17, 523, 540
 autoofs filesystem, 226
 Automake, 523, 540
 automounter daemon, 226
 Autoopts library, 50
 awk program, 6, 15, 185, 186, 318, 323, 326, 436, 459, 460, 468, 514, 588, 631, 632, 643, 658, 685
 GNU version of. *See* gawk

B

b, in permissions, 7, 131
 back doors, 630
 Bash shell, xxv, 314, 329, 598
 bc command, 100
 Beebe, Nelson H.F., xxix, 467
 beepers, 653
 Bell Labs, 20
 Berry, Karl, xxix
 bg command, 334, 365
 binary data, 100
 binary executables, 6, 51–52, 87, 230, 328, 392
 binary trees, 575–587
 depth of, 576
 insertions in, 578–579
 lookups in, 579–581
 nodes of, 576, 585–586
 pointers in, 578–581
 removals in, 577, 585–586
 subtrees of, 576
 traversals in, 577, 581–585
 bind(), 347, 481–483, 484, 487, 502, 504
 bindtextdomain(), 43, 534, 535, 537, 540, 554, 555
 binfmt_misc filesystem, 226, 230
 bitwise operators, 242, 612
 blocks, 113, 221
 bad, 224
 boot, 235
 comparing, 425
 copying, 423–424
 fragments of, 236
 functions for, 423
 indirect, 201
 number of, 133, 201, 210, 236, 271
 size of, 133, 236
 superblock, 235

Bloom, Benjamin, 666, 668
 Bourne shell, xxv, 335
 break statement, 436–437
 breakpoints, 599, 612, 630–632, 656
 inside macros, 609
 Brennan, Michael, xxix
 brk(), 72–73, 74, 76, 77, 441
 Brooks, Fred, 668
 BSD Fast Filesystem, 224–225, 227
 BSD Unix, 17, 108, 117, 120, 131
 core dumps in, 296
 debuggers in, 593–594
 directories in, 391
 dirfd(), 247
 file locking in, 558
 file ownership in, 146
 filesystems in, 124, 224
 4.4 BSD code, 687
 fts(), 252
 getpgrp(), 301
 group sets in, 384
 network databases in, 190
 setreuid() and setregid(), 396
 signal(), 340
 signals in, 342, 348, 350, 366, 379
 sorting functions in, 178
 st_blocks field, 211
 timezone(), 170
 wait3() and wait4(), 298, 329
 bsd_signal(), 340, 345, 354, 379, 381
 bsearch(), 180–189, 192, 197, 643
 BSS (Block Started by Symbol), 51
 BSS areas, 51, 52
 BSS sections, 52–54
 buffer cache, 105–106, 131
 buffers
 maintaining strategy, 65–70
 overrunning, 59, 67, 617
 size of, 71, 142–144, 152, 248, 617, 625

C

c, in permissions, 7, 131
 C language, xvii
 1999 Standard, xxi
 continuation lines in, 65
 NULL constant in, 56
 preprocessor in, 608
 type of character constants in, 212
 See also Original C, Standard C
 C++ language
 2024 Standards, xxi
 assignment of a pointer value in, 57
 const items in, 53
 function prototypes in, 9
 GNU programs in, 17
 main(), 290
 names in, 593
 preprocessor in, 608
 sorting arrays of objects in, 177
 type of character constants in, 212
 Caldera Ancient UNIX License, xx, 671–672
 callloc(), 55–56, 58–60, 63–64, 72, 77, 267
 Capey, T., 670

- carriage return character, 67
 - cat program, 79–80, 92–93, 95–96, 114, 119–121, 140–141, 174, 229, 234, 278, 307, 428–429, 533, 584, 595–596, 599
 - GNU version of, 96
 - V7 version of, 91–96, 140
 - cattr program, 125–127
 - catgets(), 508, 524, 553
 - cd command, 8, 146, 169, 191, 222, 228, 247–250, 314–315, 331, 391–393, 458, 603
 - CD-ROMs, 9, 225–227, 230–231
 - cfgetispeed(), 347
 - cfgetospeed(), 347
 - cfsetispeed(), 347
 - cfsetospeed(), 347
 - char type, 57, 212, 542
 - character sets, 508, 541, 554
 - characters
 - classes of, 514, 552
 - lowercase vs. uppercase, 509, 514, 552
 - order of, 541
 - wide, 541–545
 - chdir(), 247–248, 269, 270, 271, 347
 - “check every call for errors” principle, 15, 69
 - check_salary(), 630, 631
 - chgrp program, 4, 391
 - child(), 375, 376
 - client program, 477–478
 - connect(), 489
 - ftp, 489–492
 - socket(), 489
 - chmod program, 4, 101, 146, 384, 385, 391, 393, 566
 - chmod(), 102, 146, 147, 152, 153, 347
 - chown program, 4, 146, 199, 250, 272
 - chown(), 146, 147, 152, 192, 347, 392, 396
 - chroot(), 221, 269–271, 276, 288, 331
 - manpage of, 270
 - cleanup(), 341
 - clearenv(), 38, 39
 - manpage of, 39
 - clock_gettime(), 115, 347, 569–570
 - close(), 79, 80, 89–93, 103, 108, 109, 279, 303, 305, 307, 308, 313, 325, 326, 329, 331, 347, 485, 564
 - closedir(), 124–126, 129, 152, 454
 - close-on-exec flag, 316–318, 323, 327, 329
 - close-on-fork flag, 316–317
 - Cocker, Gail, xxx
 - coda filesystem, 226
 - code formatting, 14, 647
 - codeset, 534–535
 - coding style, 18–19
 - COFF (Common Object File Format), 6
 - Coherent filesystem, 227
 - collating sequences, 512
 - Collyer, Geoff, xxix, 57, 88, 360
 - command line processing, 14
 - command substitution, 458, 469
 - Common filesystems on x86 hardware, 225, 270
 - Common Object File Format. *See* COFF
 - compar(), 203, 216, 217, 253
 - compatibility with standards, 13
 - compile_pattern(), 464–466
 - compilers, 619, 632
 - conditions, 610
 - logging, 629
 - using variables for, 608
 - confstr(), 411–413, 417
 - connect(), 347, 489, 491, 494, 502, 504
 - const keyword, 13, 53, 86, 244, 428
 - consumers, 305, 329
 - cont command (GDB), 601, 612, 631, 656
 - continuation lines, 68
 - continue statement, 436
 - Coordinated Universal Time. *See* UTC
 - coprocesses. *See* pipes, two-way
 - copyright(), 531, 532
 - core dumps, 54, 296, 334, 421, 434, 468, 594, 595
 - core file, 594–596
 - Cox, Russ, xxix
 - cp program, 208, 249, 288, 338, 339, 486, 538, 540, 551, 596, 648
 - cpio program, 139, 147, 149
 - cramfs filesystem, 226
 - creat(), 101, 103–104, 108, 114, 121, 136, 147, 277, 291, 305, 308, 319, 347, 391
 - flags for, 319–320
 - critical sections, 350, 373
 - cryptography, 442, 447, 450
 - csh, manpage of, xxv, 418
 - ctime(), 159–160, 169, 197, 203, 209, 515, 523, 524
 - currency symbols, 509, 517, 520, 525–526
- D**
- d, in permissions, 4
 - daemons, 271, 306
 - data access model, 15, 20
 - datagrams, 474
 - data sections, 51–54
 - data segments, 51–54
 - dates, 155
 - current, 156
 - formatting, 509, 523
 - daylight-saving time. *See* DST
 - dbx debugger, 593
 - dcgettext(), 528
 - ddd debugger, 604
 - deadlocks, 83, 306
 - debug_dummy(), 631
 - debuggers, 334, 421, 434, 592–593, 633–650, 653, 657
 - graphical, 604, 656
 - machine-level vs. source-level, 593
 - debugging, 591–657
 - compilation for, 592–593, 599
 - macros for, 606–608, 657
 - memory allocation, 633–634
 - rules of, 653–655, 657
 - runtime, 622–632, 657
 - debugging files, 629–630
 - debugging symbols, 592, 606, 656
 - decimal point, 507, 517
 - delete operator, 55, 57, 586
 - DeMaille, Akim, xxix
 - demand paging, 392
 - denial of service attack, 624
 - determinism, 443
 - /dev/fd/xx files, 109, 314–315, 329, 331
 - /dev/random file, 447–449, 469
 - /dev/urandom file, 447–449, 469

- device numbers, 137–138, 222
 - devices, 7, 20, 112, 131
 - block, 7, 131, 133, 138, 152, 230
 - busy, 83
 - character, 7, 131, 133, 138, 152, 230
 - loopback, 227, 230
 - masks for, 137
 - slow, 342
 - types of, 137
 - devpts filesystem, 226, 229, 234
 - df program, 112, 235, 236, 241, 271
 - dgettext(), 527, 528
 - diff program, 16, 174, 314, 315, 425, 440
 - difftime(), 156, 174, 197
 - DIR* objects, 124–126, 128–129, 144, 452, 554
 - direct struct, 113, 123–124, 213
 - directories, 5, 112–115, 130, 152
 - changing, 247–248
 - creating, 121–123
 - current position in, 129–130, 152
 - current root, 8, 151, 222, 269–271, 276
 - current working, 8–9, 20, 116, 228, 247, 269–271, 276
 - absolute pathname to, 248
 - information about, 201
 - mask for, 137
 - moving, 120
 - parent, 116
 - reading, 123–130
 - removing, 118, 121
 - symbolic links to, 119
 - system root, 222–224, 269–270
 - for temporary files, 433, 468
 - walking, 250–261, 271
 - directory entries, 113–114, 117, 119, 124–127, 152, 201
 - file types in, 129
 - length of, 237
 - reading, 213
 - sorting, 177–180, 215
 - directory permissions. *See* permissions, directory
 - dirent struct, 124–127, 129, 132, 152, 178–179, 197, 213, 452, 454
 - dirfd(), 128, 144, 247, 272
 - discarding data, 7
 - dispositions. *See* signal actions, default
 - do_input(), 611
 - do_statfs(), 245, 246
 - do_statvfs(), 239, 245
 - Drepper, Ulrich, xxix, 360
 - DST (daylight-saving time), 158, 168–170
 - du program, 250
 - GNU version of, 261–269, 271
 - dup(), 307–314, 317, 319, 329, 347, 397, 565
 - dup2(), 307–314, 317–319, 326, 329, 331, 347, 492
 - dup3(), 318, 319, 329, 347
 - DVDs, 227, 230
 - dynamic data structures, 575
 - dynamic memory, 16, 51, 55–76
 - accessing after shrinking, 60
 - accessing outside the bounds, 59, 591, 634, 635
 - aligned, 557–558, 586
 - calculating size of, 56, 62
 - changing size of, 59–61, 73
 - debugging, 633–632, 657
 - freed, 58, 77, 442, 634, 637
 - initially allocating, 56–58, 386–387
 - leaks of, 59, 61, 77, 178, 440–442, 633, 635, 639
 - releasing, 58–59, 77
 - tracing, 634
 - unfreed, 633
 - uninitialized use of, 637, 640
 - zero-filling, 59, 63, 423
- ## E
- EBCDIC encoding, 540–541
 - echo program, 23, 27–28, 31
 - ed editor, 394, 435, 459, 460
 - ELF (Extensible Linking Format), 6, 284
 - Emacs editor, 14, 15, 459, 598, 661
 - emp_name_id_compare(), 173, 175, 579, 582–583
 - employee struct, 172–176, 177, 181, 182, 183, 184, 560–562, 579, 580, 581, 582, 583, 585, 586
 - ENABLE_NLS constant, 530, 531, 532, 533, 535, 536
 - encodings, 508, 541, 554
 - multibyte, 545
 - self-correcting, 545
 - endmntent(), 232, 234, 239
 - endpwent(), 191–192
 - entropy pool, 447
 - env program, 40–48
 - environ variable, 39–40, 46–47, 49, 284–286
 - environment variables, 8, 9, 37–39
 - adding, 38, 40
 - for debugging, 624–628
 - with empty values, 38
 - expansion of, 458, 469
 - for locales, 508–510
 - random order of, 40
 - removing, 39, 41
 - environments, 9, 37–48, 276
 - clearing, 39, 41
 - Epoch, 148, 155, 197, 567
 - eras, 525–526
 - errno variable, 56, 81–86, 89, 91, 93, 103, 108, 115, 118, 120, 121, 123, 125, 143, 156, 167, 196, 247, 249, 254, 283, 284, 291, 293, 294, 321, 325, 326, 341, 344, 348, 358, 379, 394, 398, 399, 407, 449, 453, 481, 482, 485, 521, 535, 546, 547, 550, 557, 562, 563, 569, 574, 629
 - examination of, 82
 - manpage of, 82
 - values for, 82–86
 - error messages, 14–16, 69, 86–87, 93, 108, 434
 - diagnostic identifiers for, 86–87
 - handling, 30
 - errors, 81
 - reporting functions for, 86
 - Ersoy, Alper, xxix
 - /etc/fstab file, 228–232, 234, 270, 271
 - /etc/mtab file, 229–232, 233, 234, 238, 240, 270
 - /etc/vfstab file, 228
 - euidaccess(), 390, 403
 - exclude option, 263, 264, 450, 470
 - exec(), 47, 105, 283–289, 293, 307, 316, 327, 328, 378–379, 381, 397, 400, 454, 660
 - exec1(), 285–286, 326, 347
 - execle(), 286, 347
 - execvp(), 285, 286, 297, 400
 - executable code, 51–54, 77

- execv(), 286, 287, 347, 453, 500, 501
 - execve(), 284–286, 347
 - manpage of, 284
 - execvp(), 47, 48, 286, 312, 313, 400, 453
 - exit(), 87, 291, 293, 295, 329, 434, 564
 - _exit(), 292, 293, 312, 329, 347
 - _Exit(), 292, 293, 329, 347
 - EXIT_FAILURE constant, 289, 290, 328, 465
 - exit status, 41, 43, 46, 48, 81, 95, 289–290, 293–300, 329, 331, 366, 375, 377, 564
 - EXIT_SUCCESS constant, 46, 47, 116, 122, 123, 167, 289, 290, 328, 465
 - ext2 filesystem, 225, 226, 392
 - ext3 filesystem, 225, 226, 229, 392
 - ext4 filesystem, 119, 133, 225, 226, 229–232, 234, 240, 246, 392
 - Extensible Linking Format. *See* ELF
 - Extensible Markup Language. *See* XML
- F**
- faccessat(), 145, 347, 390, 402, 403
 - FAT filesystem, 225, 227, 270
 - fchdir(), 247–248, 269, 271, 272, 276, 288, 347, 660
 - fchmod(), 102, 147, 153, 347
 - fchown(), 145–148, 192, 347, 396
 - fclose(), 125, 232
 - fcntl(), 307, 315–323, 329, 331, 347, 558–566, 586, 587
 - flags for, 318–320
 - manpage of, 315, 320, 560, 562
 - fexecve(), 105, 285, 286, 347
 - FD_CLOEXEC flag, 317–319, 323, 479
 - FD_CLOFORK flag, 317–319, 323, 480, 488
 - fdatasync(), 107, 108, 347
 - fdopen(), 327, 470
 - fflush(), 107, 606
 - fg command, 334, 365
 - fgets(), 67, 77, 78, 469
 - FIFO (first-in first-out) files, 131, 306–307, 329, 360
 - creating, 306
 - empty, 321
 - nonblocking I/O, 320–323
 - removing, 307
 - file descriptors, 87–96, 108, 132, 150, 196, 303
 - attributes of, 315–323, 329
 - bad, 83
 - closing, 309, 434
 - duplicating, 318–319
 - functions for, 107
 - leaks of, 325
 - lowest acceptable value of, 318
 - new, 88, 90
 - obtaining, 247
 - for open files, 276–279, 303, 318, 328, 432
 - copying, 307–314, 326, 329
 - shared, 277, 328
 - file modes. *See* permissions, file
 - file permissions. *See* permissions, file
 - file status flags, 319–320, 323
 - file table, 277
 - File Transfer Protocol. *See* FTP
 - filenames, 5
 - basing program’s behavior upon, 288
 - changing, 117–118, 148
 - functions for, 107–108
 - generating, 426–430
 - length of, 5, 113, 125, 254
 - fileno(), 90, 95, 107, 128, 470, 564
 - files, 3–7, 20
 - attributes of, 315–327, 329
 - byte positions within, 96, 104, 563
 - closing, 88–91, 279
 - copying onto themselves, 95
 - creating, 101–106, 108, 114–115
 - existing, 83, 104
 - information about. *See* metadata
 - locking, 558–559, 586
 - opening, 83, 88–91, 107, 108
 - reading, 89, 91–94, 105–106, 108
 - regular, 130, 342
 - mask for, 137
 - removing, 118–119
 - restoring from archives, 147, 149
 - shared, 90
 - size of, 85, 107–108, 133, 250
 - truncating, 104, 133
 - types of, 129–136, 139
 - macros for, 137, 138, 152
 - masks for, 136–137
 - writing, 89, 91–94, 104–106, 108, 133, 148
 - filesystems, 111–112, 124, 151, 221–228, 270
 - busy status of, 228
 - debugging, 201
 - information about, 234
 - journaling, 225, 270
 - mounting, 127, 132, 221, 225, 227, 228, 271, 385, 562, 566
 - read-only, 83, 226, 227, 230, 237
 - unmounting, 221, 228
 - find program, 139, 250, 450
 - manpage of, 139
 - finish command (GDB), 600, 612
 - first-in first-out. *See* FIFO files
 - flags, 33, 241, 345, 612
 - converting to a string, 613–618
 - values, 241–242
 - flags2str(), 614, 615, 616, 617, 618
 - flock struct, 560–562, 565
 - flock(), 558, 565–566, 586
 - manpage of, 566
 - fnmatch(), 451–453, 469
 - folders. *See* directories
 - fopen(), 9, 213, 431
 - fork(), 275–280, 281, 283, 284, 288, 289, 293, 297, 298, 307, 310, 313, 316, 317, 325, 328, 352, 366, 370, 372, 376, 378–379, 381, 397, 400, 478, 483, 484, 487, 499, 564, 565, 660
 - format_num(), 367, 371, 374
 - fortune program, 442
 - fpathconf(), 237, 249, 386, 405, 409–411, 417
 - fprintf(), 80, 86, 87, 98, 216, 338, 339, 543, 606, 607, 632
 - fpsync(), 107
 - Free Software Foundation. *See* FSF
 - free(), 55, 57–61, 64, 65, 72, 74, 77, 144, 178, 180, 218, 250, 427, 442, 558, 586, 633–634, 639
 - FreeBSD filesystem, 225
 - FSF (Free Software Foundation), xxii, xxvi, xxvii, 594
 - fstat(), 95, 109, 131, 132, 136, 140, 147, 248, 272, 279, 347, 390, 486

- fstatfs(), 242–246, 271
 - fstatvfs(), 235–243, 271
 - fsync(), 107, 108, 347
 - FTP (File Transfer Protocol), 270
 - ftp client code, 489–492
 - ftpd server code, 486–488
 - ftp program, xxvi, xxvii
 - ftruncate(), 107–109, 348
 - fts(), 221, 241, 252, 257, 261, 271, 272, 582
 - fts_open(), 250–261
 - FTW struct, 251
 - ftw(), 251
 - functions, xxi, xxv
 - callback, 291, 329, 581, 583
 - debugging, 614
 - declarations of, 13, 27
 - helper, 612–618
 - low-level, 73, 77
 - naming conventions for, 107–108
 - recursive, 52, 62
 - wrapper, 70
 - futimens(), 150–151, 348
- G**
- garbage collectors, 16
 - gawk program, xxii, 15–16, 18, 35, 61, 65, 318, 323–327, 387, 436, 437, 441, 527, 539, 550, 552, 591, 599–601, 602, 608, 610, 611, 613–615, 617, 620, 622, 624–628, 631, 632, 647–650, 658
 - numeric values formatting in, 521–523
 - two-way pipes, 323
 - GCC (GNU Compiler Collection), 14, 35, 74, 592, 656
 - macros in, 608–609, 621
 - GDB (GNU Debugger), 16, 593–605, 608–613, 631, 632, 636, 656
 - distributions of, 602
 - macros in, 609, 621
 - Gemmellaro, Anthony, xxx
 - Gemmellaro, Faye, xxx
 - General Public License. *See* GNU GPL
 - generality, 16
 - genflags2str(), 616, 617
 - getcwd(), 144, 248–250, 257, 259, 271, 272
 - manpage of, 250
 - getdelim(), 70–71, 77, 466
 - getdents(), 128
 - manpage of, 128
 - getdtablesize(), 87, 91, 108, 277, 289, 316
 - getegid(), 348, 387, 402
 - getenv(), 38, 49
 - geteuid(), 348, 386, 402
 - getgid(), 348, 386, 402
 - getgroups(), 348, 386–387, 402
 - getitimer(), 277, 365, 570–573, 586
 - manpage of, 571
 - getline(), 70–71, 77, 466, 469
 - getmntent(), 231–234, 237, 243, 271
 - getname(), 209, 210, 218
 - getopt(), 25, 28–37, 49, 205, 232, 237, 244
 - GNU version of, 25, 29, 31–32, 37, 49
 - manpage of, 32
 - getopt_long(), 14, 25–26, 28–37, 41, 44, 49, 536, 601
 - getopt_long_only(), 32, 49
 - getpeername(), 348, 486
 - getpgid(), 301, 329
 - getpgrp(), 301, 329, 348
 - getpid(), 277, 279–281, 328, 348, 447, 500
 - getppid(), 277, 279–281, 328, 348
 - getpwent(), 191, 192, 193
 - getpwnam(), 191, 192, 193
 - getpwuid(), 191, 192, 193, 210
 - getrandom(), 449–450, 469
 - getrlimit(), 414–418
 - getresgid(), 348, 398–399
 - getresuid(), 348, 398–399
 - getsid(), 302
 - getsockname(), 348, 486, 491
 - getsockopt(), 348
 - gettext program, 508, 509, 526–540, 555
 - gettext.h file, 67, 530–533, 535, 537, 554
 - gettext(), 527–535, 537, 554
 - gettext_noop(), 531, 537
 - gettimeofday(), 155, 567–569, 573, 586
 - manpage of, 567
 - getty program, 396, 397
 - getuid(), 192, 348, 386, 396, 402
 - GID (group ID), 4, 20, 102, 121, 133, 147, 191, 193, 198, 383
 - effective, 383–384, 386–388, 390–402
 - mask for. *See* setgid bit
 - real, 383–390, 393–402
 - saved set, 384, 386, 397–399, 402
 - gid_t type, 134, 155
 - Glub library, 197
 - GLIBC (GNU C Library), 14, 18, 37, 58
 - errno values, 82–86
 - euidaccess(), 390
 - f_flag values, 237, 241
 - glob() extensions, 454–455
 - libintl.h, 532–533
 - rand(), 445
 - superuser in, 383
 - TEMP_FAILURE_RETRY(), 344
 - glob(), 453–458, 469
 - globalization, 507
 - globerr(), 456, 457
 - globfree(), 453–458
 - glyphs, 540
 - GMT (Greenwich Mean Time), 134
 - gmtime(), 157–159, 197
 - GNOME Project, 197
 - GNU C Library. *See* GLIBC
 - GNU Coding Standards, 13–20, 26, 58, 64, 87, 288
 - GNU Compiler Collection. *See* GCC
 - GNU Coreutils, xxvi, 28, 43, 108, 115
 - distribution of, 45
 - du, 261–269, 271
 - fts(), 261
 - install, 297
 - safe_read() and safe_write(), 342–344
 - sort, 341
 - utime(), 149
 - wc, 426
 - xreadlink(), 144
 - GNU Debugger. *See* GDB
 - GNU Getopt library, 50
 - GNU GPL (General Public License), xxvii, 13, 673–684
 - GNU Lesser General Public License, 37
 - GNU programs, xxi–xxii, xxvii, 13–17, 27

- long options in, 26
 - wrapper functions in, 69
 - GNU Project, xxii
 - GNU/Linux, xvii, xviii, xix, xx, xxv, xxvi
 - block size in, 133
 - chroot(), 270
 - clearenv(), 39
 - core dumps in, 296
 - debuggers in, 593
 - /dev/fd/XX files in, 314–315
 - directories in, 391
 - dirfd(), 247
 - distributions of, xxvi, 226, 228, 334, 421, 508
 - Epoch in, 148
 - file formats in, 6
 - file types in, 136
 - filesystems in, 111, 124, 226–228, 270
 - ftw(), 251
 - inode numbers in, 224
 - locales in, 510
 - mounting in, 127, 222
 - numeric values formatting in, 521
 - preemptive multitasking in, 282
 - /proc/self/cwd, 250
 - remove(), 118, 152
 - renaming operation in, 118
 - rsync, xxv
 - rusage struct, 299
 - signal(), 340
 - signals in, 334–337, 348, 379
 - standard functions in, 72
 - statfs() and fstatfs(), 242–246
 - superuser in, 383
 - time slicing in, 429
 - time_t type, 155
 - time-zone information in, 169–170
 - versionsort(), 178
 - wait3() and wait4(), 298, 329
 - Gold, Yosef, xxix
 - goto statement, 435
 - Greenwich Mean Time. *See* GMT
 - grep program, 24, 459–463, 467, 468
 - groff program, 14
 - group, category of users, 4, 101–102, 383
 - changing, 146
 - databases of, 190, 193–196, 198
 - IDs of. *See* GID
 - lists of users of, 193
 - masks for, 136, 137
 - names of, 193, 198, 201, 210
 - passwords of, 193
 - group (struct), 194, 195, 199
 - group sets, 193, 384, 390, 402
 - changing, 393–394, 402
 - number of groups in, 386–387, 402
 - retrieving, 386
 - gstat(), 203, 206, 214–215
 - GTK project, 197
 - gzip program, xxvi, xxvii
 - to directories, 114, 117
 - to root directory, 222
 - hash tables, 265
 - hasmntopt(), 232
 - Hayes, John R., 666, 669
 - heap, 52–54, 441
 - Heisenberg, Werner, 422
 - heisenbugs, 422
 - help option, 14, 26, 42, 46, 636
 - here documents, 119
 - Hesiod, 190
 - Hierarchical File System, 226
 - High Performance File System, 226
 - Hoare, C.A.R., xx, xxix, 171, 419, 422
 - Hoare's law, xx
 - holes, 98, 100, 133
 - HOME environment variable, 9, 38
 - HURD kernel, 14
- I**
- I/O, 79, 93
 - asynchronous, 356
 - blocking, 105
 - nonblocking, 320–323
 - random access, 96, 100, 108
 - sequential, 96
 - standard functions for, 90
 - synchronous, 106, 108, 237
 - i18n. *See* internationalization
 - IBM, 226, 458, 541
 - idtype, 294, 296
 - IEEE Standard 1003.1–2024, xxii
 - #ifdef, 18, 68, 241, 242, 288, 335, 342, 343, 406, 408, 409, 508, 606, 607, 614, 620, 625, 627
 - ifind(), 180
 - IFS environment variable, 400
 - indentation, 256
 - index nodes. *See* inodes
 - indexing, 57, 62, 67
 - infinite loops, 441
 - init process, 8, 280, 293, 328, 359, 366, 396, 397
 - init_groupset(), 387, 388
 - initialized data, 51
 - initstate(), 445–447
 - inode change time, 133, 135, 201
 - changing, 148
 - inode numbers, 112–114, 117, 124, 132, 140, 152, 201
 - for root directory, 224, 270
 - inodes (index nodes), 5, 112–114, 152, 221
 - number of, 236, 271
 - install program, 297
 - interfaces, 17, 18
 - internationalized program, 535–536
 - International Organization for Standardization. *See* ISO
 - internationalization (i18n), 507, 553–554, 659
 - interpreters, 6, 619–620
 - interval timers. *See* timers
 - Introduction, manpage of, 28, 82, 128
 - invariants, 419
 - IPC (interprocess communication), 131, 302, 360, 473
 - internet building blocks, 474–475
 - addresses and interfaces, 476
 - IPv4, 475

- IPC (interprocess communication) (*continued*)
 - IPv6, 475–476
 - network byte order, 476–477
 - networking technologies, 474
 - using signals for, 360, 380
 - isatty(), 196, 198
 - ISO (International Organization for Standardization), xxi
 - ISO 9660 CD-ROM filesystem, 225, 226, 270
 - ISO C. *See* Standard C
 - ISO/IEC International Standard 9899, xxi
 - ISO/IEC International Standard 14882, xxi
 - iswalnum(), 542
 - iswlower(), 542
 - itimerval struct, 571, 572, 573, 586
- J**
- Java language, 9
 - job control, 300–301, 329, 334, 365
 - job control shells, 300, 359, 380
 - Johnson, Steve, 3
 - Journalized File System, 226
 - Journalized Flash Filesystem, 226
- K**
- K&R C. *See* Original C
 - K&R style of code formatting, 18
 - Kerberos network database, 190
 - Kernighan, Brian W., xvii, xxviii, 9, 18, 62, 629, 656, 661
 - kill program, 336, 365
 - kill(), 300, 326, 346, 348, 355, 358–360, 380
 - killpg(), 300, 326, 348, 358–360, 380
 - Kirsanov, Dmitry, xxx
 - Kirsanova, Alina, xxx
 - Knuth, Donald E., 197, 467, 655
 - ksh (Korn) shell, 323, 324, 329, 451
 - ksh88 shell, 314
 - ksh93 shell, xxv, 314, 382, 659
- L**
- l10n. *See* localization
 - LANG environment variable, 509
 - Lave, Jean, 669
 - lbuf struct, 202–203, 206–207, 208, 213–214, 215, 216, 218
 - invalid, 208
 - LC_ALL environment variable, 162–163, 509–511, 518–519, 521, 524, 540
 - lchmod(), 147
 - lchown(), 146, 396
 - lconv struct, 515–518, 519, 521–524
 - ld program, 6
 - LDAP (Lightweight Directory Access Protocol), 190
 - Lechlitner, Randy, xxix
 - LEDs, 653
 - Lehman, Manny, xxix
 - libintl.h file, 527–530, 532, 533, 534–535
 - libraries, 16
 - general-purpose, 340
 - POSIX standard for, 17
 - shared, 54, 318
 - Lightweight Directory Access Protocol. *See* LDAP
 - limits.h file, 124, 135, 142, 164, 249, 257, 283, 321, 330, 386, 405, 406–409, 411, 416, 427, 432, 517, 547
 - line-at-a-time jail, 603–604
 - line feed character, 67
 - line program, 278, 597
 - link count, 114, 118, 133, 152
 - link program, 115–116
 - link(), 115–116, 152, 347, 348
 - links, symbolic, 119–121, 130–132, 141–144, 151
 - creating, 141
 - to directories, 119–120, 250
 - levels of, 83
 - mask for, 137
 - ownership of, 146
 - permissions on, 147
 - lint program, 12, 632, 657
 - Linux. *See* GNU/Linux
 - Linux Journal, 13
 - list command (GDB), 598, 599, 600
 - listen(), 348, 482–483, 488, 502, 504
 - lldb debugger, 605
 - ln program, 114–115, 119–121, 288
 - locale program, 509–513, 521–522
 - manpage of, 552
 - localeconv(), 515–518, 553
 - locales, 507–508, 552–553
 - categories of, 508–510, 553
 - default, 508
 - setting, 510–511
 - localization (l10n), 161, 507, 553–554
 - localtime(), 157–158, 159, 160, 169, 197
 - lockf(), 558, 559–564, 565, 586
 - locks, 558–559, 586
 - advisory, 559, 564–565, 586
 - descriptor-based, 564–565
 - exclusive, 565
 - mandatory, 237, 559, 566–567, 586
 - obtaining, 562–564
 - process-based, 562–564
 - range, 563
 - read, 559, 561, 586
 - record, 558
 - releasing, 562–565
 - shared, 565–566
 - whole file, 558, 566
 - write, 559, 561, 586
 - log files, 628–630, 634, 653
 - logic analyzers, 653
 - login program, 191, 339, 396, 397
 - longjmp(), 348, 436–442, 468, 587, 646
 - ls command, 4, 7, 101, 130–131, 138–139, 153, 190, 193, 196, 198, 201, 202, 217, 219, 452, 545, 548, 659
 - modern versions of, 201–202, 218
 - V7 version of, 201–218
 - manpage of, 201, 210
 - lsearch(), 419–423
 - lseek(), 96–99, 104, 108, 210, 277, 303, 347, 560, 625
 - whence values for, 96
 - lstat(), 131, 132, 136, 141–142, 143, 144, 152, 250, 347, 454–455
- M**
- MacOS X NetInfo network database, 190
 - magic numbers, 6, 221
 - converting to printable strings, 245
 - main program, 180, 290–291, 292, 315, 488, 573, 583, 606
 - main(), 8, 26, 29, 40, 43, 49, 69, 76, 79–81, 95, 125, 180, 184, 195, 203, 206, 215, 234, 239, 244, 245, 284, 290,

- 293, 295, 311, 318, 329, 339, 344, 356, 363, 369, 372, 376, 420–421, 436, 440–441, 464, 510–511, 522, 527–528, 548, 554, 564, 583, 645, 657
 - declaring, 26
 - process exit status in, 80
 - major(), 138, 152, 209
 - make program, 322, 540, 652
 - GNU version of, 64–70
 - makedev(), 138
 - Makefile, 65, 66
 - makeuname(), 203, 212, 213, 218
 - malloc(), 52, 55–58, 59–65, 69, 70, 72, 73, 77, 142–144, 178, 214, 218, 249–250, 387, 412, 423, 434, 440–441, 497, 511, 514, 558, 575, 578, 633–634, 638–640
 - casting the return value, 57
 - manpage of, 55
 - MALLOC_TRACE environment variable, 634
 - man command, xix, xxv
 - manage(), 374–376
 - manifest constants, 88
 - manpages (manual pages), xxv, 653
 - Marti, Don, xxix
 - mb1en(), 545
 - mbr1en(), 545
 - mbrtowc(), 545, 546, 547, 548, 550, 554
 - mbsrtowcs(), 545, 546, 547, 550, 554
 - mbstowcs(), 545
 - mbtowc(), 545
 - McGary, Greg, xxix
 - McIlroy, Doug, xxix
 - McKusick, Marshall Kirk, 328
 - memalign(), 557–558, 586
 - memcpy(), 347, 423, 469, 491, 543
 - memchr(), 347, 423, 426, 543
 - memcmp(), 347, 423, 425, 469
 - memcpy(), 347, 423–425, 468, 491, 543
 - Memishian, Peter, xxix
 - memmove(), 18, 68, 347, 423–425, 468, 543
 - memory, 7–9, 52
 - address space, 51–54
 - dynamically allocated. *See* dynamic memory
 - overlapping areas of, 423
 - read-only, 53
 - setting, 423
 - use of, 16
 - memset(), 57, 63–64, 347, 423, 543, 546
 - message catalogs, 526, 529, 534, 554
 - message object files, 539, 554
 - metacharacter expansions. *See* wildcard expansions
 - metadata, 5, 111, 118, 131–134, 151–152
 - modification time of. *See* inode change time
 - Meyering, Jim, xxix, 28, 38, 143, 251
 - Microsoft Windows
 - convention of line ending in, 67
 - Epoch in, 148
 - filesystems in, 111, 225, 227
 - minicomputers, 4, 7, 301
 - Minix filesystem, 225–227
 - minor(), 138–139, 152, 209
 - mkdir(), 121–123, 152, 347
 - manpage of, 120
 - mkdtemp(), 430–433, 468
 - mke2fs program, 111
 - mkfifo program, 131, 307
 - mkfifo(), 307, 329, 347
 - manpage of, 307
 - mkfs program, 111
 - mknod program, manpage of, 138
 - mkostemp(), 430, 432
 - mkstemp(), 430–433, 468, 470
 - mktemp(), 426–430, 432, 470
 - mktime(), 165–167, 169, 197
 - mntent struct, 231–234, 238–239, 244–245
 - modification time, 133–135, 201
 - changing, 148–147, 152
 - formatting, 164
 - retrieving, 586
 - sorting by, 201
 - monetary formatting, 509, 515–520, 553
 - mount points, 222, 270
 - mount program, 119, 222, 227–231, 235, 243, 266, 270, 385, 567
 - manpage of, 210, 231
 - mount(), 222
 - mounting. *See* filesystems, mounting
 - MS-DOS, 225
 - msdos filesystem, 225–227
 - multiuser systems, 383
 - mv program, 117, 288, 538–539
- ## N
- N_() macro, 531–532, 535, 537, 554
 - named pipes. *See* FIFO files
 - nanosleep(), 573–575, 587
 - native language support. *See* NLS
 - nblock(), 203, 208, 210–211, 218
 - NetBSD filesystem, 225
 - network databases, 190
 - Network File System. *See* NFS
 - Network Information Service. *See* NIS
 - network technologies, 474
 - new operator, 55, 57, 63, 76, 276, 320, 323, 326, 328, 331, 366, 412, 484, 488, 597, 602, 607, 644
 - newfs program, 111
 - next command (GDB), 600, 656
 - NeXTStep system, 225
 - NFS (Network File System), 224, 270
 - nftw(), 251–264
 - flags for, 251–257
 - manpage of, 253
 - private version of, 272
 - ngettext(), 528–529, 536, 554
 - nice values, 276, 282–283, 328
 - nice(), 282–283, 328
 - NIS (Network Information Service), 190
 - n1_langinfo(), 524–526, 535, 553
 - NLS (native language support), 507, 553
 - “no arbitrary limits” principle, 15–16, 20, 51, 64–66, 613, 617, 661
 - nodots(), 179, 180
 - nonlocal gotos, 435, 468
 - Norvig, Peter, xxix, 665
 - NTFS filesystem, 226–227
 - NUL character, 5, 15–16, 66, 69
 - NULL constant, 43, 56–64, 77, 126, 167, 184, 192, 194, 215, 232, 272, 421, 424, 453, 514, 546, 585

numbers, 33
 formatting, 509, 515–524, 553
 grouping digits within, 517, 520, 522–523

O

`O_ACCMODE` flag, 320
`O_APPEND` flag, 104, 319, 320
`O_ASYNC` flag, 320
`O_CLOEXEC` flag, 105, 316
`O_CLOFORK` flag, 105, 317
`O_CREAT` flag, 99, 104, 108, 119, 319, 428
`O_DIRECT` flag, 320
`O_DIRECTORY` flag, 105
`O_DSYNC` flag, 105–107, 319, 320
`O_EXCL` flag, 104, 119, 319, 432
`O_EXEC` flag, 105, 285
`O_NOATIME` flag, 320
`O_NOCTTY` flag, 105, 319
`O_NOFOLLOW` flag, 105
`O_NONBLOCK` flag, 105, 319, 320–321, 322, 330, 480
`O_PATH` flag, 105, 145, 285
`O_RDONLY` flag, 89, 92, 104, 124, 248, 285, 319, 320, 321, 448
`O_RDWR` flag, 89, 99, 104, 119, 319, 320, 428
`O_RSYNC` flag, 105, 106, 319
`O_SEARCH` flag, 105
`O_SYNC` flag, 105–107, 109, 319, 320
`O_TMPFILE` flag, 145
`O_TRUNC` flag, 99, 104, 108, 119, 319, 428
`O_TTY_INIT` flag, 105
`O_WRONLY` flag, 89, 104, 319, 320, 321
`O_x SYNC` flags, 107
 object file formats, 6
 obstacks, 16
`off_t` type, 96–98, 107, 132, 560, 563–564, 625
 offsets, 96, 560
 Open Group, The, xxi
`open()`, 79–80, 88–91, 98–99, 101, 104–106, 108, 114, 121, 123, 127, 129, 141, 247, 256, 277, 285, 291, 308, 314, 321, 347, 390–391, 429, 432, 470, 565
 flags for, 89, 104–106, 108, 319–321
 manpage of, 106
`openat()`, 106, 144–145, 150, 347, 470
 OpenBSD filesystem, 225
`opendir()`, 124–126, 144, 152, 252, 272, 288, 453
 OpenVMS filesystem, 112
 operands
 order of, 25
 placement of, 26
 Opt library, 49
`optimal_bufsize()`, 625–628
 optimizations, 592, 599, 656
 option struct, 32, 34–35, 43, 601, 623
 options, 14, 23
 debugging, 622–624, 633
 invalid, 30–31
 long, 14, 26, 32–37
 names of, 24, 25
 placement of, 25–26
 undocumented, 623
 vendor-specific, 24, 35
 Original C, 9–13, 20
 function parameters in, 632
 GNU programs in, 13
 See also C language, Standard C

`os_close_on_exec()`, 327
 other, category of users, 4, 101–103, 383
 masks for, 136–137
 owner, of a file. *See* user
 ownership, 4, 111
 changing, 145–147, 152
 masks for, 137

P

`p`, in permissions, 131
 parameters, 26–28, 52
 lists of, 632
 parent process ID. *See* PPID
 parse trees, 619
`parse_debug()`, 623, 658
 partitions, 111, 151, 270
 Pascal language, 435
`passwd` struct, 191–193, 497
 PATH environment variable, 9, 38–39, 42, 285, 286, 328, 389, 400
`pathconf()`, 237, 249, 386, 409–410, 417
 pathnames, 112, 120
 absolute, 248, 271
 checking for validity, 390
 relative, 8, 247
`pause()`, 339, 347, 361–364, 380, 382
 PDP-11, 3, 113, 218, 392, 445
`prentry()`, 208
 Perl language, 459
 permission bits, 4, 212, 218, 248
 constants for, 101–102
 default, 102
 masks for, 136–137
 permissions, 20, 111, 132, 135–141, 212
 changing, 4, 102, 147–148, 152
 checking, 383, 402
 denied, 83
 directory, 5, 118
 expressed in octal, 101
 file, 4, 101–103, 118
 macros for, 135
`perror()`, 82, 108, 109
`pfatal_with_name()`, 70
 PGID (process group ID), 277, 294–302, 329
 PID (process ID), 7–9, 276–277, 279–281, 285, 302, 328–330
 parent. *See* PPID
 of the process that died, 294, 297
 using in seed values, 447
 wrapping around, 279
`pid_t` type, 275–276, 279–280, 292, 294, 296, 297–298, 301–302, 310, 354, 367, 371, 376, 478, 483–484, 560
 Pike, Rob, 20, 656, 661
 Pinard, François, xxix
`pipe()`, 303–305, 309, 324, 329, 347
 PIPE_BUF constant, 321–322, 409–410
 pipes, 9, 303, 329, 361
 blocking, 320
 broken, 83
 buffering, 305–306
 creating, 303–305
 empty, 321
 named. *See* FIFO files
 nonblocking, 320–323
 nonlinear, 314, 329

- synchronization of, 306
 - two-way, 323–327
- Plan 9 from Bell Labs, 50, 103
- `pmode()`, 212
- pointers
 - calculating, 55
 - dangling, 58, 634
 - declaring, 56
 - freeing twice, 58
 - generic, 55
 - guaranteed valid, 69
 - invalid, 56, 60–62
 - passing, 58
 - setting to `NULL`, 58
 - sorting, 177
- `poll()`, 347
- Popt library, 50
- portability, 14, 17–18, 21, 35, 56, 67, 74, 127, 327, 607, 659
- portable object files, 538, 554
- portable object templates, 538
- Portable Operating System Interface. *See* POSIX standard
- positional specifiers, 533–534
- POSIX standard, xxi
 - `bsd_signal()`, 340
 - character classes in, 514, 552
 - compatibility with, 13, 37
 - directories in, 392
 - `environ` variable, 39
 - errno values, 82–86
 - `_exit()`, 290–293
 - extensions, xxi
 - ADV, 557
 - FSC, 107
 - SIO, 107
 - XSI, xxi, 72, 107, 124, 235, 295–296
 - `FD_CLOEXEC` flag, 317
 - file locking in, 558
 - file ownership in, 4
 - filesystems in, 124
 - file-type and permission bitmasks in, 137
 - flags for `open()`, 104–106, 319–320
 - `ftw()`, 251
 - `isatty()`, 196
 - library and system call interfaces in, 17
 - nice values in, 283
 - option conventions in, 24–26, 35
 - `PIPE_BUF` constant, 321–322
 - `printf()`, 533–534
 - process group information in, 301
 - `rusage`, 299
 - signals in, 341, 349, 350, 352, 357–360, 366, 379
 - `st_blocks` field, 211
 - superuser in, 383
 - symbolic constants for permissions in, 101–102
 - `time_t` type, 156
 - timezones in, 168–169
 - `waitpid()`, 294
- `posix_memalign()`, 557–558, 586
- `posix_trace_event()`, 347
- `POSIXLY_CORRECT` environment variable, 31, 263, 411, 413
- postconditions, 419
- PPID (parent process ID), 7, 277, 279–281, 328
- preconditions, 419
- predictable algorithms (`mktemp()`), 429
- preemptive multitasking, 282
- `print_emp()`, 583
- `print_employee()`, 182
- `print_group()`, 195
- `print_mount()`, 234
- `printf()`, 9, 90, 122, 125, 160, 163–164, 178, 207, 261, 367, 439, 520–523, 528–529, 535, 553–554, 600, 653
 - manpage of, 521
 - POSIX version of, 533–534
- priority, 282–283
- private allocators, 64
- privileged operations, 394
- `proc` filesystem, 226, 229–231, 234
 - `/proc/mounts` file, 228–231, 234, 271
 - `/proc/self/cwd` file, 250
- process groups, 300–302, 329
 - background, 301
 - foreground, 301
 - IDs of. *See* PGID
 - leaders of, 300, 302, 329
 - orphaned, 301
 - sending signals to, 358–359
 - setting, 302
- process signal mask, 277, 349–353, 360, 380, 437–438, 468
 - starting out, 363
- process substitution, 314
- `process_file()`, 267, 269
- processes, 7–9, 20, 51–54
 - blocking, 350, 570
 - child, 8, 83, 275, 277, 279, 309–314, 324, 328
 - dead, 293, 366–378
 - nondeterministic order of, 312
 - continuing if stopped, 334, 359, 365, 375, 377, 379
 - creating, 275, 328
 - executing programs in, 281, 328
 - exiting, 81
 - IDs of. *See* PID
 - killing, 379, 422, 468
 - orphan, 280
 - parent, 7, 275, 277, 279, 293, 309–312, 324, 328, 366–378
 - polling, 295
 - reading, 305
 - reaping, 293
 - stopping, 294, 295, 334, 365, 379
 - suspending, 361, 365
 - synchronization of, 282
 - terminating, 289–293, 329, 333, 341, 364, 574, 586
 - writing, 305
- producers, 305, 329
- profiling, 571
- programs
 - basic structure of, 79–81
 - distributions of, 37
 - logging, 628–629
 - messages in, 526–540, 553, 629
 - names of, 24
 - production versions of, 592, 623, 624
 - running. *See* processes
 - testing, 651–652, 657, 660
 - undocumented features in, 630
- prompts, xxv
- prototypes, 9–12, 20, 632
- `ps` program, 52, 546, 550
- `pselect()`, 347, 501

pseudorandom numbers, 442–450, 467
 pseudoterminals (pseudo-ttys), 7, 196, 226
 ptrdiff_t type, 55, 74–75, 440
 putenv(), 39, 49
 GNU version of, 39, 49
 pwd program, 117, 119–120, 153, 256, 314–315, 497, 500

Q
 QNX4 filesystem, 226
 qsort(), 171–181, 188, 197, 206, 216, 577, 588, 644
 Quicksort algorithm, 171
 quote_n(), 116

R
 r, in permissions, 4
 race conditions, 117, 344–346, 349, 351, 362, 373, 380, 390, 429, 467–468, 573, 636
 radix point, 520, 525
 Rago, Stephen, 327
 raise(), 337, 341, 346–347, 355–356, 359, 365, 379
 RAM disks, 226
 Ramey, Chet, xxix, 38, 40, 87, 210, 249, 314, 326, 433, 591, 592, 633
 ramfs filesystem, 226
 rand(), 443–445, 469
 compared to random(), 447
 GLIBC version of, 445
 manpage of, 445
 random numbers, 442, 447, 467–469
 random(), 445–448, 450, 469
 compared to rand(), 446
 manpage of, 445, 448
 Raymond, Eric S., 661, 667
 read build-in shell command, 65, 77, 209, 267, 278, 466, 471
 read end (of pipe), 303, 305, 309
 read(), 79, 80, 91–94, 105–106, 108, 127, 141, 277, 305, 321–323, 341–342, 347, 469, 473, 485, 489, 492, 541
 readdir(), 124–128, 152, 207, 213–214, 452–454
 GNU/Linux version of, 127
 readline library, 60, 66, 598, 661
 readline(), 65, 70, 77
 readlink(), 141–146, 152, 250, 347
 readstring(), 66
 realloc(), 55, 59–64, 69, 77, 144, 249, 267, 551, 575, 579, 633
 GNU version of, 64
 Standard C version of, 61
 recurse(), 596, 598
 recv(), 347, 493–494, 504
 recvfrom(), 347, 493, 504
 recvmsg(), 347, 493
 regcomp(), 460–463, 469
 regerror(), 460, 465–466
 regexexec(), 460–463, 466, 469
 regfree(), 460, 465
 register keyword, 28, 50
 registers, 52
 regular expressions, 459–467, 469
 basic, 460
 extended, 460
 ranges in, 509, 514
 reiserfs filesystem, 225, 226
 Remote File System. *See* RFS

remove(), 118–119, 152, 307
 GNU/Linux version of, 118, 152
 rename(), 117, 118, 121, 152, 347
 reproducibility, 592
 return values, 52–53, 289–290, 328
 126 and 127 codes, 290, 293
 casting, 57, 60
 to void, 90
 checking, 57, 60, 90, 342
 negative, 290
 rewinddir(), 124–125, 152
 RFS (Remote File System), 224
 Ritchie, Dennis M., xvii, xxix, 9, 18, 384
 rm program, 5, 115, 120–121, 393, 429, 597
 rmdir program, 123
 rmdir(), 118, 121, 152, 347
 Robbins, Miriam, xxix
 Rock Ridge extensions, 226
 romfs filesystem, 227–228
 root (superuser), 5, 7, 36, 117, 138, 146–148, 225, 228–229, 231, 254, 269, 391–393, 359, 383–385, 388, 393–403, 567, 577–581, 584–586
 rr debugger, 604–605
 rsync program, xxvi
 run command (GDB), 41, 286, 315, 599
 runtime checks, 422, 468
 rusage struct, 298–299, 329

S
 S_ISXXX(), macros, 102, 135, 137, 141, 152, 153, 211
 sa_flags field, 354–355, 360, 366
 SA_NODEFER flag, 353–355, 360
 SA_NOMASK flag, 355
 SA_ONESHOT flag, 355
 SA_RESETHAND flag, 355, 360
 safe_read(), 342–344
 safe_write(), 342–344
 sbrk(), 72–74, 75, 76–77, 440, 441, 442
 scandir(), 178, 179, 180, 197
 scanf(), 166, 176
 SCO UnixWare Boot Filesystem, 226
 scripts, 6
 sdb debugger, 593
 searching
 binary, 180–190, 197–198, 426
 linear, 180, 197–198, 210, 419
 in user/group databases, 192
 sectors, of a disk, 236
 security, 147–148, 191, 385, 399–400, 402, 422, 432, 468
 sed program, 459, 460, 468
 seed values, 443, 446, 469
 seekdir(), 130, 152
 segmentation violation, 57–58
 select(), 178, 212, 347, 494–501, 504, 505
 sem_post(), 347
 send(), 347, 493–494, 504
 sendmsg(), 348, 493
 sendto(), 348, 493, 494, 504
 sentinel elements, 99
 server program
 accept(), 483–484
 bind(), 481–482
 close(), 485
 end of connection, 485–486

- ftpd, 486–488
- listen(), 482
- read()/write(), 484–485
- shutdown(), 485
- socket(), 478–481
- sessions, 300, 329, 359
 - IDs of, 302
 - leaders of, 300
- setegid(), 348, 394–395, 396, 398, 402
- setenv(), 38, 39–40, 49
- seteuid(), 348, 394, 395, 396, 398, 399, 402
- setjmp(), 435–438, 439, 440–442, 468, 587
- setgid bit, 137, 237, 248, 384–385, 396–398, 402, 566–567, 586
 - for directories, 391–392, 402
- setgid(), 348, 394–395, 396–398, 402
- setgroups(), 393–394, 397, 402
- setitimer(), 277, 364–365, 570–573, 586–587
- setjmp(), 435–438, 439, 440–442, 468, 587
- setlocale(), 43, 508, 510–512, 523, 535, 548, 552, 553
 - manpage of, 508
- setmntent(), 231–232
- setpgid(), 302, 329, 348
- setpgrp(), 302, 329
- setpwent(), 191–192
- setregid(), 348, 394–396, 398, 402
- setresgid(), 348, 398–399, 402
- setresuid(), 348, 398–399, 402
- setreuid(), 348, 394–396, 398, 402
- setrlimit(), 414–418
- setsid(), 302, 348, 500
- setsockopt(), 348
- setstate(), 445–447
- settimeofday(), 567
- setuid bit, 137, 237, 248, 384–385, 396–398, 402
 - running as root, 399, 403
- setuid(), 348, 395, 396–399, 402
- Seventh Edition Research UNIX System. *See* V7 Unix
- Seward, Julian, xxix
- shell escapes, 394
- shells, 659
 - separating arguments in, 23
 - sorting environment variables, 40
- shift states, 545
- shutdown(), 348, 485, 489, 504
- si_code field, 354–356, 373
- side effects, 423
- sig_atomic_t type, 344–346, 349, 380
- sigaction struct, 353, 354, 356, 357, 358, 366, 376, 380
- sigaction(), 296, 333, 340, 342, 346, 348, 350, 353–357, 358, 360, 362–363, 379–381, 498
 - manpage of, 353, 354
- sigaddset(), 348, 351–352, 360, 380, 381
- sigaltstack(), 355
- sigdelset(), 348, 351–352, 360, 380, 381
- sigemptyset(), 348, 351, 352, 360, 362–363, 369–370, 380, 381
- sigfillset(), 348, 351, 352, 360, 380, 381
- sighold(), 349, 350, 351, 381
- sigignore(), 350, 381
- siginterrupt(), 357–358, 360
- sigismember(), 348, 351–352, 360, 380, 381
- siglongjmp(), 348, 437–439, 442, 468
- signal actions, 333, 379
 - default, 333–341, 350, 378–379
 - restoring, 335
- signal handlers, 334, 337–350, 353–357, 379
 - functions that can be called from, 347–348
 - installing, 350, 364
 - reinstalling, 339–340, 344–345
 - restoring, 348
 - shell-level, 335
- signal numbers, 295, 335
- signal sets, 351–352
- signal(), 334–337, 340, 346, 348–350, 354, 357, 364–365, 379–382
 - BSD version of, 340, 496–501
 - GNU/Linux version of, 340
 - manpage of, 334, 335
- signals, 293–294, 329, 333, 346
 - available under GNU/Linux, 335–336
 - blocking, 349, 351, 353, 360, 362, 378, 380
 - catching, 334
 - death of child, 381
 - ignoring, 334, 335, 340–341, 357, 361, 378–379
 - interrupt, 7, 294, 301, 337, 348, 357–358
 - job control, 7, 294, 295, 301, 365–366, 380
 - pending, 352, 357, 366, 378, 380–381
 - real-time, 337
 - sending, 358–359, 379–380, 383
 - supported, list of, 336
 - using for IPC, 360–361, 380
- sigpause(), 350, 363, 381
- sigpending(), 348, 352, 357, 360, 380
- sigprocmask(), 348, 352–353, 360, 380
- sigqueue(), 348, 356
- sigrelse(), 350, 381
- sigset(), 349–351, 354, 381
- sigset_t type, 349, 351–353, 360, 380, 381
- sigsetjmp(), 437–439, 468
- sigsuspend(), 348, 352, 353, 360, 363, 376, 380–382
- sigvec(), 350, 498
- simplicity, 3, 7, 9, 16, 20
- single-stepping, 600, 656
- size program, 54, 61, 66, 76–77, 422
- size_t type, 55–56, 63–70, 71, 160, 171–172, 181, 344, 422–425, 446, 449, 453, 460, 546–547, 557
- sizeof operator, 56–58, 99, 172, 424, 487, 490–491, 557, 560–562, 617
- sleep(), 281–282, 348, 363–365, 380, 382, 470, 570, 573, 574
- SMB filesystem, 227
- SOCK_CLOEXEC flag, 479, 484
- SOCK_CLOFORK flag, 480, 484, 488
- SOCK_DGRAM flag, 479, 480, 494
- SOCK_NONBLOCK flag, 480, 484
- SOCK_RAW flag, 480
- SOCK_RDM flag, 480
- SOCK_SEQPACKET flag, 480
- SOCK_STREAM flag, 479–481, 487, 489, 491, 493, 499–502
- socket(), 348, 478–481, 484, 488, 489, 491, 494, 502, 504
- socketpair(), 348
- sockets, 84–85, 131
 - mask for, 137
 - UPD, 494
- SOCK_NONBLOCK, 480, 484

- soft links. *See* symbolic links
- Solaris, 271
 - core dumps in, 296
 - directories in, 391
 - filesystem in, 225, 228
 - gettext, 508
 - numeric values formatting in, 521
 - signals in, 338–339, 348, 379
- sort program, 171, 340, 426
 - manpage of, 171
- sorting, 171–190, 197
 - data, 185–190
 - by modification time, 201
 - of pointers, 177
 - stable, 173–174
- spaghetti code, 435
- SPARC system, 225
- speed, 16
- Spencer, Henry, xxix, 124
- splint (Secure Programming Lint) program, 632, 657
- sprintf(), 160, 164, 218, 520, 521, 543, 647
- srand(), 443–446, 469
- srandom(), 445–447, 469
- ssize_t type, 70–71, 79, 92–93, 128, 492
- st_ctime field, 134, 135, 148
- st_mode field, 132, 135–137, 140–141, 153
- st_size field, 133, 141, 142, 144, 152, 215–216, 250, 266, 625, 628
- stack, 52–54, 73
- stack frames, 598
- stack segments, 52–54
- stack traces, 598
- Stallman, Richard M., 21, 64
- Standard C, 20
 - 1990 ISO, xxi, 9–13, 17–20
 - 1999 ISO, xxi, 12, 17–20
 - const items in, 53
 - exiting functions in, 290–293
 - GNU programs in, 13
 - main(), 290
 - realloc(), 60
 - remove(), 118
 - signal functions in, 334–337
 - time_t type, 155
 - variadic macros in, 608
 - wide characters in, 541–542
 - See also* C language, Standard C
- standard error, xvii, 8, 20, 88, 108, 132, 196
 - sending debugging messages to, 606
- standard input, xvii, 8, 20, 25, 88, 93, 108, 132, 196, 303, 323, 462–463, 465
 - shared by two processes, 277–278
- standard output, xvii, 8, 20, 25, 88, 108, 132, 140, 196, 303, 323
 - shared by two processes, 277–278
- standards, xx
- stat struct, 95–96, 131–132, 134–135, 146–148, 150, 152, 153, 155, 192, 203, 211, 216, 218, 222, 250, 251, 255, 261, 267, 271, 279, 455, 625, 629
- stat(), 129, 131–133, 135, 136, 138, 140–141, 146, 147, 152, 153, 164, 196, 198, 205, 210, 214, 216, 250, 253, 255, 272, 348, 390, 402, 453, 455, 630
 - expensiveness of, 205
 - manpage of, 133
- statfs struct, 243–245, 246
- statfs(), 242–246, 271
- static tables, 16
- statvfs struct, 235, 236, 239, 241–243
- statvfs(), 235–243, 246, 271
- stderr variable, 7, 31, 36, 41, 69, 80, 86, 87, 90, 92–93, 95, 98, 99, 123, 126, 139, 140, 167, 179, 183, 194, 213, 214, 216, 233, 238, 239, 245, 257, 259, 280, 283, 287, 304, 325, 382, 389, 406, 410, 412, 416, 417, 444, 456, 465–467, 481, 482, 484, 486, 489–492, 494, 498–500, 536, 570, 606, 607, 632
- stdin variable, 7, 71, 88, 90, 95, 175, 184, 313, 325, 464–465, 487, 513, 572, 584
- stdio.h file, xvii, 68, 90, 92, 94, 107, 124–125, 128, 130, 291–292, 346, 389, 427, 430, 469, 542, 545, 564, 625, 661
- stdlib.h file, 56, 430–432, 443
- stdout variable, 7, 28, 43, 44, 46, 60, 92, 94–95, 116, 140, 203, 216, 313, 325, 487, 491, 513, 519, 543
- step command (GDB), 600, 612, 656
- Stevens, W. Richard, 327
- sticky bits, 5, 248, 392
 - for directories, 402
 - mask for, 137
- stopme(), 631–632
- strcasecmp(), 515, 543
- strcmp(), 123, 124, 172–173, 178, 218, 347, 425, 426, 512, 513, 514–515, 542, 543, 553, 582–583
- strcoll(), 178, 512–515, 543, 552, 553
- strcpy(), xxv, 347, 367–369, 374–375, 425, 543, 614, 645
- strdup(), 71–72, 77, 511, 543
- strerror(), 15, 86, 93, 98–99, 108, 125–126, 259–261, 304, 325, 536
- strfmon(), 519–521, 535, 553
- strftime(), 160–164, 166, 168–170, 197, 198, 519, 520, 523–526, 535, 553, 587
- string terminator, 38, 66, 425
- strings
 - comparing, 509, 512–515, 553
 - copying, 71
 - marking for translation, 529–530
- strip program, 54, 297
- strip(), 297
- strncmp(), 347, 515, 543, 624
- strtoul(), 628
- structs, in C, 601
 - arrays of, 618
 - nested, 620–622
 - size of, 56–57
- strverscmp(), 178
- strxfrm(), 178, 512–515, 543, 552, 553
- subshells, 278
- Sun Microsystems, 190, 224, 391
- superblocks, 221
- superuser. *See* root
- symbolic constants, 33, 88, 609, 612
 - using enums for, 610
- symbolic links, 119–121, 130–132, 141–144, 151
 - creating, 141
 - to directories, 119–120, 250
 - levels of, 83
 - mask for, 137
 - ownership of, 146
 - permissions on, 147

symbols, 54
 symlink(), 120, 141, 152, 347
 manpage of, 120
 sysconf(), 386, 405–410, 417, 418
 syslog(), 498, 499–501, 629
 system calls, xix, xxi, 8, 15–16, 79
 checking for errors, 93
 failing, 81, 108
 indirect, 127–128
 interrupted, 83, 341, 348
 POSIX standard for, 17
 restartable, 341–344
 system console, 7
 System III, 28
 debuggers in, 593
 executable files in, 288
 FIFOs in, 306
 System V, 131
 directories in, 391–392
 file locking in, 558
 filesystems in, 224, 226–228
 ftw(), 251
 signals in, 338–339, 341–342, 348, 350, 360, 366, 380
 st_blocks field, 211
 UIDs in, 384
 sysv filesystem, 225, 227

T

t, in permissions, 5
 Taber, Louis, xxix
 tar program, xxvi, xxvii, 139, 147, 149
 Taub, Mark, xxix–xxx
 tcdrain(), 347
 tcflow(), 347
 tcflush(), 347
 tcgetattr(), 347
 tcgetpgrp(), 347
 tcseendbreak(), 347
 tcsetattr(), 347
 tcsetpgrp(), 347
 tcsh, manpage of, xxv, 300
 tdelete(), 577–578, 585–586
 tdestroy(), 577, 585–586
 telldir(), 130, 152
 TEMP_FAILURE_RETRY() macro, 344, 348
 tmpnam(), 426–427
 temporary files, 16, 340, 426–434, 468
 directories for, 119, 433
 opening, 430–433
 terminals (ttyps), 7, 105, 131, 140–141, 196–198, 276, 365, 396–397
 controlling, 300–301
 nonblocking, 321
 reading data from, 571
 text domain, 527, 554
 text sections, 51–54
 text segments, 51–54, 76–77
 textdomain(), 43, 44, 527–528, 535, 554
 tfind(), 577–578, 579–581
 Thompson, Ken, 103
 thousands separators, 517, 521–523, 525
 thrashing, 59
 threads, 54
 tilde expansion, 455, 458, 469

Time Sharing Option. *See* TSO
 time slicing, 282, 429
 time zones, 168–170, 197
 time(), 156–157, 158–161, 164, 197, 347, 567
 time_t type, 134, 148, 150, 155–157, 158–161, 164–166, 174, 197, 536, 567
 timeouts, 571
 timer_getoverrun(), 347
 timer_gettime(), 347
 timer_settime(), 347
 timerclear() macro, 568, 572
 timercmp() macro, 568
 timerisset() macro, 568
 timers, 570–573, 586–587
 expiring, 571, 573
 setting, 573
 times, 155, 197
 broken-down, 157, 164, 197
 current, 156–157
 formatting, 159–164, 509, 523–524
 local, 157
 resolution of, 567–571, 575, 586
 times(), 347
 timespec struct, 132, 134, 150–151, 501, 569, 574, 587
 timestamps, 134, 148–151
 timeval struct, 149, 150, 298, 299, 496, 501, 567–569, 570–573, 574–575, 586
 timezone(), 170
 tm struct, 157–160, 161, 164, 165, 166–168, 197, 537
 tm_isdst field, 158, 165
 /tmp directory, 5, 117, 119–120, 122–123, 393, 433, 468, 534
 TMPDIR environment variable, 433–434, 468
 tmpfile(), 291, 430–431, 468, 470
 tmpfs filesystem, 227, 234–235
 tmpnam(), 426–427
 tokens, 543
 touch program, 164, 198
 translations, 526–540, 554
 creating, 536–540
 preparing, 535–536
 testing, 534
 updating, 540
 trap built-in shell command, 335
 traps, 335
 troff program, 14, 661
 Tromey, Tom, xxix
 truncate(), 107–108
 tsearch(), 577, 578–581, 584, 585
 TSO (Time Sharing Option), 3
 ttyps. *See* terminals
 TUHS (The UNIX Heritage Society), xxv
 tune2fs program, manpage of, 225, 236
 Turing, Alan, 419
 Turski, Wladyslaw M., xxix
 twalk(), 577–578, 581–585
 two_way_open(), 324
 type2str(), 244, 245–246
 TZ environment variable, 168, 170
 tzset(), 168–169, 197

U

UDF filesystem, 227
 UID (user ID), 4–5, 20, 102, 133, 191–193, 198, 383
 effective, 359, 383–384, 393–398

- UID (user ID) (*continued*)
 - mask for. *See* `setuid` bit
 - real, 359, 383–384, 393–398
 - saved `setuid`, 384, 386, 394–399, 402
 - `uid_t` type, 132, 134, 155, 191–192, 199, 383, 385–386, 394–396, 398–399
 - `ulimit` built-in shell command, 87–88, 109, 413–414, 417–418
 - `umask` built-in shell command, 101–103, 109
 - `umask()`, 102–103, 108, 109, 276, 288, 347
 - `umasks`, 102–104, 276
 - `umount` program, 222, 228, 229, 232, 270
 - `umsdos` filesystem, 225, 227
 - `uname()`, 225, 347
 - Uncertainty Principle, 422
 - Unicode character set, 508, 541
 - uninterruptible power supply. *See* UPS
 - unions, in C, 601, 618–622, 657
 - nested, 620–622
 - `unistd.h` file, 88, 344, 405, 406, 408–412
 - universally unique identifier (UUID), 229
 - Unix
 - archives of old versions, xxv
 - block size in, 133
 - `chroot()`, 270
 - convention of line ending in, 67
 - date, 161
 - domain sockets, 502
 - Epoch in, 134
 - file formats in, 6
 - file ownership in, 147
 - filesystems in, 111–112
 - `ftw()`, 251
 - inode numbers in, 224
 - mounting in, 127
 - preemptive multitasking in, 282
 - programs in, 16
 - reading directories in, 123
 - standard functions in, 72
 - time slicing in, 429
 - `time_t` type, 155
 - UNIX Heritage Society, The. *See* TUHS
 - Unix Support Group, 28–29
 - `unlink()`, 118, 119, 152, 307, 347
 - `unref()`, 622
 - `unsetenv()`, 39, 48, 49
 - UPS (uninterruptible power supply), 106
 - URLs
 - ANSI, xxi
 - Arqp, 50
 - The Art of Computer Programming*, 467
 - Autoopts, 50
 - Cargs, 50
 - ddd debugger, 604
 - debugging rules, 653
 - Free Software Foundation, 593–594
 - GNOME project, 197
 - GNU Coding Standards, 13
 - GNU Gengetopt, 50
 - GNU `gettext`, 540
 - GNU `grep`, 468
 - GNU Make, 65
 - GNU Project, xxii
 - GNU Project, The, article, 21
 - GTK+ project, 197
 - Hints On Programming Language Design*, 422
 - ISO, xxi
 - Linux Journal*, 13
 - Notes on Programming in C*, 20
 - Open Group, The, xxi
 - Plan 9 From Bell Labs, 50
 - Protection of Data File Contents, 384
 - Recommended C Style and Coding Standards*, 20
 - `rsync` web pages, xxv
 - `splint` program, 632
 - Teach Yourself Programming in Ten Years*, 665
 - TUHS, xxv
 - Unicode, 541
 - United States Patent and Trademark Office, 384
 - Valgrind, 642
 - XFS, 227
 - `usage()`, 43, 115–116, 464, 466, 467
 - user, category of users, 4–5, 101–103, 383
 - databases of, 190–193, 198
 - IDs of. *See* UID
 - masks for, 136–137
 - names of, 190, 198, 201, 210
 - passwords of, 191
 - See also* ownership
 - UTC (Coordinated Universal Time), 134, 169
 - UTF-8 encoding, 15, 508
 - `utimbuf` struct, 148
 - `utime()`, 148–149, 152, 156, 164, 569, 586
 - `utimensat()`, 145, 150–151, 164, 347
 - `utimes()`, 149–150, 299, 347, 496, 586
- V**
- V6 Unix, 457–458
 - V7 Unix, xx, xxv, 13
 - cat, 94, 140
 - debuggers in, 593
 - directories in, 391
 - distribution of, 27
 - filesystem in, 224
 - inode numbers in, 113
 - `ls` command, 201–218
 - mapping names to ID numbers in, 190
 - `rand()`, 445
 - `rmdir` command, 123
 - signals in, 338–339, 341–342, 348, 350, 361
 - `wait()`, 294
 - Valgrind debugger, 634–644, 657
 - van der Linden, Peter, xxix
 - variables
 - for important conditions, 608
 - local, 52
 - logging, 629
 - temporary for debugging, 611
 - variadic macros, 608
 - VAX system, 224
 - `--verbose` option, 14, 26, 636
 - Veritas VxFS journaling filesystem, 227
 - `--version` option, 14, 46
 - `versionsort()`, 178
 - vfat filesystem, 224, 225, 227, 271
 - `v1` editor, 459–460, 598–599, 661
 - `vim` editor, 365, 459–460, 603–604, 656

virtual circuits, 474
 void * type, 55, 171–175, 181–182, 254, 355, 423–425, 449–450, 557, 568, 577–585
 volatile keyword, 13, 345–346, 361, 380, 438, 439

W

w, in permissions, 4
 wait(), 282, 290, 294–300, 309, 311–313, 329–331, 342, 348, 359, 366–367
 wait3(), 298–300, 329
 wait4(), 298–300, 329
 waitid(), 282, 290, 294–298, 300, 329, 348, 359
 waitpid(), 282, 290, 294–298, 300, 329, 348, 359, 369, 377
 flags for, 295
 warm fuzzies, 139
 watchpoints, 601–602, 656
 wctomb(), 545, 550, 551, 554
 wcsrtombs(), 545, 550, 554
 wcstombs(), 545
 wctomb(), 545
 wget program, xxvi
 wide characters
 byte maximums, 547
 converting bytes, 546–547, 550–552
 GNU, 548–550
 languages, 552

 mbstate_t, 545–546
 wildcard expansions, 450–459, 469
 wordexp(), 458–459, 469
 wordfree(), 458–459
 words
 order of, 533–534, 554
 plural forms of, 528–529, 554
 wprintf(), 543
 write end (of pipe), 303, 305, 309
 write(), 79, 80, 91–92, 94, 105–106, 108, 127, 304, 305, 306, 320–322, 339, 341, 348, 368–369, 371, 374–375, 377, 484–485, 489, 492–494

X

x, in permissions, 5
 Xenix filesystem, 225, 227
 XFS filesystem, 227
 xgettext program, 530, 537–538
 xinetd program, manpage of, 361, 495
 XML (Extensible Markup Language), 629
 xreadlink(), 142, 144, 250, 389
 xrealloc(), 68, 69

Z

zero-initialized data, 51–52
 zombies, 293, 366