



SCO OpenServer™ Development System

Programming Tools Guide

SCO OpenServer™

SCO OpenServerTM
Development System
Programming Tools Guide

© 1983–1995 The Santa Cruz Operation, Inc.; © 1989–1994 UNIX System Laboratories, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, California, 95060, USA. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

This SCO OpenServer Development System software is commercial computer software and, together with any related documentation, is subject to the restrictions on U.S. Government use as set forth below. If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If this procurement is for a civilian government agency, the following FARs Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: This computer software is submitted with restricted rights under Government Contract No. ____ (and Subcontract No.____, if appropriate). It may not be used, reproduced, or disclosed by the Government except as provided in paragraph (g)(3)(i) of FAR Clause 52.227-14 alt III or as otherwise expressly stated in the contract. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. This SCO OpenServer Development System software includes software that is protected by these copyrights:

© 1983–1995 The Santa Cruz Operation, Inc.; © 1987 Apollo Computer Inc.; © 1988–1991 Apple Computer, Inc.; © 1976–1992 AT&T; © 1986–1987 Convergent Technologies Inc.; © 1987–1992 Digital Equipment Corporation; © 1993 D.I.S - Università' di Pavia; © 1987–1988 Don Bennett; © 1988 Evans & Sutherland Computer Corporation; © 1991–1992 Fuji Xerox Co., Ltd.; © 1991–1992 Fujitsu Limited; © 1990–1993 Groupe Bull; © 1986–1992 Hewlett-Packard Company; © 1990 Ing.C.Olivetti & C., S.p.A.; © 1990 Intel Corporation; © 1990 Interactive Systems Corporation, (ISC); © 1988–1994 IXI Limited; © 1991–1994 Karl Lehenbauer and Mark Diekhans; © 1990 Larry Medwin; © 1986–1988 Larry Wall; © 1987–1994 Legent Corporation; © 1988–1993 Locus Computing Corporation; © 1985–1993 Massachusetts Institute of Technology; © 1989 Matra Datavision Inc.; © 1990 Megatek Corporation; © 1980–1989 Microsoft Corporation; © 1990 Motorola Inc.; © 1989–1991

National Computer Graphics Association; © 1990 NCR Corporation; © 1989–1993 Network Computing Devices, Inc.; © 1990–1991 NTT Software Corporation; © 1989 Object Design, Inc.; © 1990 Oki Electric Industry Co., Ltd.; © 1992 Oki Technosystems Laboratory, Inc.; © 1990–1991 OMRON Corporation; © 1989–1994 Open Software Foundation, Inc.; © 1982 Pavel Curtis; © 1989 Prentice Hall; © 1979–1994 Regents of the University of California; © 1987–1990 Siemens Nixdorf Informationssysteme AG; © 1991–1992 Sony Corporation; © 1990–1992 SORD Computer Corp.; © 1986–1992 Sun Microsystems Inc.; © 1988–1991 Tektronix Inc.; © 1990–1992 Toshiba Corp.; © 1990–1991 UniSoft Group Limited; © 1990 Unisys Corporation; © 1986 University of Toronto; © 1978–1993 UNIX System Laboratories, Inc.; © 1988 Wyse Technology, Inc.; © 1988–1992 X Consortium. All Rights Reserved. SCO NFS and SCO TCP/IP were developed by Legent Corporation based on Lachman System V NFS. SCO TCP/IP is derived from Lachman System V STREAMS TCP, a joint development of Legent Corporation and Convergent Technologies.

SCO, the SCO logo, The Santa Cruz Operation, Multiscreen, Open Desktop, ODT, SCO Global Access, SCO OpenServer, SCO MPX, SCO MultiView, SCO Visual Tcl, and VP/ix are trademarks or registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries. UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company Limited. All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

Date: 31 January 1995

Document Version: 5.0

About this book	1
Using this guide	1
Typographical conventions	4
Related documentation	5
Commercial books and articles	7
How can we improve this book?	8

Chapter 1

Using programming tools	9
Creating the source code	10
Archiving the code	11
Analyzing the code	13
Using cb	13
Using cscope	14
Using lint	15
Using cflow	16
Using cxref	17
Archiving changes made during the analysis	18
Compiling the code	19
Using make	19
Testing the code	21
Debugging the code	22
Using dbxtra	22
Testing the correction	26
Program structure and guidelines	27
Program structure	28
Header files	31
Error handling	31

Chapter 2

C compilation system	33
Compiling and linking	34
Basic cc command line syntax	37
Commonly used cc command line options	40
Link editing	43
Libraries and header files	63

Header files	63
How to use library functions	64
C library (libc)	67
Math library (libm)	72
General purpose library (libgen)	75
Standard I/O	76

Chapter 3

C language compiler 83

Compilation modes	83
Phases of translation	85
Source files and tokenization	86
Tokens	86
Identifiers	86
Keywords	87
Constants	87
String literals	90
Wide string literals	90
Punctuators	90
Comments	91
Preprocessing	91
Trigraph sequences	91
Preprocessing tokens	92
Preprocessing directives	92
Declarations and definitions	99
Basic types	99
Scope	101
Storage duration	102
Storage class specifiers	102
Declarators	103
Function definitions	105
Conversions and expressions	106
Implicit conversions	106
Expressions	108
Operators	109
Associativity and precedence of operators	116
Constant expressions	116
Initialization	116
Statements	119

Expression statement	119
Compound statement	119
Selection statements	119
Iteration statements	121
Jump statements	122
Portability considerations	123

Chapter 4

COFF link editor **125**

Sections	126
Memory configuration	126
Addresses	126
Binding	127
Object files	127
Link editor command language	127
Expressions	128
Assignment statements	129
Specifying a memory configuration	131
Section definition directives	132
Changing the entry point	142
Using archive libraries	143
Allocation algorithm	145
Incremental link editing	146
DSECT, COPY, NOLOAD, INFO, and OVERLAY sections	147
Output file blocking	148
Nonrelocatable input files	148
Syntax diagram for input directives	149

Chapter 5

lint analyzer **153**

What lint does	155
Consistency checks	155
Portability checks	156
Suspicious constructs	158
Using lint	159
lint libraries	160
lint filters	161

Options and directives listed	162
Understanding lint-specific messages	165

Chapter 6

dbXtra and dbxtra 193

Invoking dbXtra and dbxtra from the command line	194
The dbXtra interface	196
Configuring dbXtra X11 and Motif resources	197
The dbxtra interface	198
Screen mode	199
dbxtra keyboard interface	201
dbxtra tutorial session	208
Preparing files for dbxtra	208
Starting dbxtra	208
The dbxtra screen	209
The dbxtra help facility	210
Printing the value of variables	210
Viewing source code and aliasing	210
Viewing the call stack	211
Traversing the call stack	211
Monitoring the contents of variables	211
Analyzing the results	213
Examining a different function	213
Using a breakpoint	213
Continuing after a breakpoint	214
Exiting dbxtra	214
dbXtra and dbxtra commands	215
File menu: saving and quitting	215
Events menu: managing execution time events	215
eXecute menu: executing the program being debugged	222
Source menu: accessing source files	224
Info menu: debugging registers, memory locations, and expressions	226
Options menu: controlling the debugging environment	228
The help menu	230
curses interface commands	231
Default command aliases	233
dbXtra and dbxtra expressions and conditions	233
C++ specific debugging features	235
Usage requirements	235

Overloaded functions and operators	236
The dbXtra and dbxtra cc command	237
Debugging child processes	238
Attaching to a running process	239

Chapter 7

sdb: symbolic debugger **241**

Using sdb	241
Printing a stack trace	243
Examining variables	243
Source file display and manipulation	245
Displaying the source file	246
A controlled environment for program testing	247
Setting and deleting breakpoints	247
Running the program	248
Calling functions	249
Machine language debugging	250
Displaying machine language statements	250
Manipulating registers	250
Other commands	251
An sdb session	251

Chapter 8

adb: absolute debugger **255**

Starting adb	255
Starting adb with a core image file	256
Exiting adb	256
Displaying program instructions and data	257
Forming addresses	257
Forming expressions	257
Choosing data formats	261
Using the = command	262
Using the ? and / commands	263
An example: simple formatting	264
Debugging program execution	265
Executing a program	265
Setting breakpoints	266

Displaying breakpoints	267
Continuing execution	267
Stopping a program with interrupt and quit	267
Single-stepping a program	268
Killing a program	268
Deleting breakpoints	268
Displaying the C stack backtrace	269
Displaying CPU registers	269
Displaying external variables	269
An example: tracing multiple functions	271
Using the adb memory maps	275
Displaying the memory maps	275
Changing the memory map	276
Creating new map entries	277
Validating addresses	277
Miscellaneous features	278
Combining commands on a single line	278
Creating adb scripts	278
Setting output width	279
Setting the maximum offset	279
Setting default input format	279
Using UNIX commands	280
Computing numbers and displaying text	280
An example: directory and inode dumps	281
Patching binary files and memory	283
Locating values in a file	283
Writing to a file	283
Making changes to memory	284

Chapter 9

C programmer's productivity tools **285**

lprof	286
Creating a profiled version of a program	286
Running the profiled program	286
The PROFOPTS environment variable	286
Examples of using PROFOPTS	288
Interpreting profiling output	290
Viewing the profiled source listing	290
Specifying program and data files to lprof	293

Files needed by lprof	293
Source listing for a subset of files	294
Summary option	295
Merging option	295
Cautionary notes on using lprof	296
Improving performance with prof and lprof	297
Improving test coverage with lprof	298
cscope	300
Configuring the environment	301
Using cscope	302
Running cscope	304
The cross-reference file	304
A tutorial example: locating the source of the error message	305
Conditional compilation directives	310
Examples of using cscope	311
Changing a text string	311
Adding an argument to a function	313
Changing the value of a variable	314

Chapter 10

make

315

Basic features	316
Parallel make	318
Recursive makefiles	318
makefiles and substitutions	319
Dependency line syntax	319
Dependency information	319
Macro definitions	320
Executable commands	322
Output translations	323
Suffixes and transformation rules	324
Implicit rules	324
Archive libraries	326
Tildes in SCCS filenames	328
The null suffix	329
Creating new suffix rules	330
include files	331
Dynamic dependency parameters	331
Environment variables	332

Suggestions and warnings	333
Makefile example	334
Internal rules	335

Chapter 11

Source code control system (SCCS) 341

SCCS for beginners	342
Terminology	342
Creating an SCCS file using admin	342
Retrieving a file by means of get	343
Recording changes by using delta	344
Additional information about get	344
Delta numbering	345
SCCS command conventions	348
x-files and z-files	348
SCCS commands	349
Error messages	350
help command	350
get command	350
delta command	360
admin command	363
prs command	365
sact command	366
rmdel command	367
cdc command	367
what command	368
sccsdiff command	369
comb command	369
val command	370
vc command	370
SCCS files	371
Protection	371
Formatting	372
Auditing	373

Chapter 12

lex

375

Writing lex programs	377
Rules section	377
Definitions section	382
The subroutines section	384
Advanced lex usage	385
Disambiguating rules	385
Context sensitivity	386
lex I/O routines	388
Routines to reprocess input	389
Using lex with yacc	391
Using lex under UNIX systems	394
Running lex	394
Compiling a lexical analyzer	395
Running the lexical analyzer	396
Using make with lex	396

Chapter 13

yacc

397

yacc specifications	398
Terminal and non-terminal symbols	399
Rules	400
Actions	402
Declarations	406
Subroutines	408
The yacc environment	409
Lexical analysis	410
lex and yylval	410
Token numbers	412
End-marker token	412
Reserved token names	412
Parsing	413
Compiling	413
Running the parser	413
Parser operation	414
Ambiguity and conflicts	419
Disambiguating rules	420

Precedence	421
Assigning a precedence to grammar rules	423
Error handling	424
Error token	425
Interactive error recovery	426
Hints for preparing specifications	427
Input style	427
Left recursion	428
Lexical tie-ins	429
Reserved words	430
Examples of using yacc	430
Simple example	430
Advanced example	433

Chapter 14

Shared libraries 443

What is a shared library?	443
Building an a.out file	445
Deciding whether to use a shared library	446
Space considerations	446
Saving space	446
Increase space usage in memory	449
Coding an application	450
Identifying a.out files that use shared libraries	450
Debugging a.out files that use shared libraries	450
Implementing shared libraries	451
The host library and target library	451
The branch table	452
Building a shared library	454
The building process	454
Guidelines for writing shared library code	457
Choosing library members	458
Include large, frequently used routines	458
Exclude infrequently used routines	458
Exclude routines that use much static data	458
Exclude routines that complicate maintenance	459
Include routines the library itself needs	459
Changing existing code for the shared library	459
Minimize global data	459

Using the specification file for compatibility	462
Importing symbols	464
Providing compatibility with non-shared libraries	471
Tuning the shared library code	471
Checking for compatibility	473
An example	476

Appendix A

ANSI implementation-defined behavior 485

Translation	486
Identifying diagnostics	486
Environment	486
Arguments to main()	486
Interactive devices	487
Identifiers	487
Significant characters without external linkage	487
Significant characters with external linkage	487
Significance of character case	487
Characters	488
Source and execution character sets	488
Bits per character	488
Mapping character sets	488
Constants with unrepresented characters and escape sequences	488
Constants with multiple or wide characters	489
Locale used for multi-byte conversion	489
Range of char values	489
Integers	490
Integer range and representation	490
Demotion of integers	490
Signed bitwise operations	490
Sign of division remainder	490
Right shift of negative-valued signed integer	491
Floating point	491
Floating-point range and representation	491
Converting an integer to a floating-point	491
Converting a floating-point to a narrower floating-point	491
Arrays and pointers	492
Largest array size	492
Casting pointers	492

Pointer subtraction	492
Registers	492
Using registers	492
Structures, unions, enumerations, and bit-fields	493
Improper access to a union	493
Padding and alignment of members of structures	493
Sign of bit-fields	493
Order of allocation of bit-fields	493
Alignment of bit-fields	494
The type of values of an enumerated type	494
Qualifiers	494
Access to volatile objects	494
Declarators	494
Maximum number of declarators	494
Statements	495
Maximum number of case values	495
Preprocessing directives	495
Character constants and conditional inclusion	495
Locating includable source files	495
Including files with quoted names	495
Character sequences	496
Pragmas	496
Definitions for date and time	497
Library functions	498
Expanding the NULL macro	498
Diagnostic printed by the assert function	498
Character testing	498
Math functions and domain errors	498
Underflow of floating-point values	499
Domain errors and the fmod function	499
The signal function	500
Default signals	501
Signal blocking	501
The SIGILL signal	501
Terminating new-line characters	501
Space characters before a new-line character	501
Null characters appended to a binary stream	501
File position in append mode	501
File buffering	502
The existence of zero-length files	502
Composing valid filenames	502

File access limits	502
Removing open files	502
Renaming with a name that exists	502
Output of pointer values	502
Input of pointer values	503
Reading ranges	503
File position errors	503
Messages generated by the perror function	503
Allocating zero memory	503
The abort function and open and temporary files	503
The exit function	504
Environment names	504
The system function	504
The strerror function	504
The time zone	508
The clock function	508
Locale-specific behavior	508
Content of execution character set	508
Direction of printing	508
Decimal point character	508
Character testing and case mapping	508
Collation sequence	510
Time and date formats	510
C Implementation limits description	511
Environmental limits	511
Translation limits	513

Appendix B

C compiler diagnostics **515**

Message types and applicable options	516
Operator names in messages	516
Messages	517
Operator names	599
Other error messages	600

Index	601
--------------------	-----

About this book

This guide explains how to use the SCO OpenServer™ Development System to create and maintain C programs. It provides useful information about programming tools and their use in developing software, including implementation-specific details about the ANSI-conforming C compiler provided with the Development System. It also contains information about debugging tools and their use in tracking down and eliminating problems in C and assembly language programs, and information about using **dbXtra**™ to debug C and C++ programs in an OSF/Motif® based windowing environment.

You will find the information you need more quickly if you are familiar with:

- the arrangement of material in this book (this page)
- the typographical conventions used in our books (page 4)
- what other documentation is available (page 5)

Although we try to present information in the most useful way, you are the ultimate judge of how well we succeed. Please let us know how we can improve this book (page 8).

Using this guide

This guide is intended for programmers who are familiar with the C programming language and the UNIX® system. The following list briefly describes each chapter.

Chapter 1, “Using programming tools” presents some of the tools available within the Development System. A sample program is created, using proper programming techniques, and analyzed with various tools. The rest of the chapter is devoted to debugging the sample program with the **dbxtra**

debugger. The debugging session is documented, accompanied by a description of the commands used in the session. Also covered in this chapter are basic program structure and guidelines.

Chapter 2, “C compilation system” describes the UNIX operating system tools used to generate an executable program from C language source files, including the compiler and linker. It also discusses static and dynamic linking, and standard libraries and header files.

Chapter 3, “C language compiler” is a guide to the SCO® C language compiler. It assumes that you have some experience with C, and are familiar with fundamental programming concepts.

Chapter 4, “COFF link editor” contains information about the Link Editor and the command language that can be used to modify the default behavior of **ld**(CP). The COFF link editor **ld** links relocatable object files to produce executable programs. Note that the **cc** command invokes the linker automatically, so the use of **ld** is optional.

Chapter 5, “lint analyzer”, describes the use of **lint** to examine C language source files to detect bugs and obscure code constructions. **lint** issues error and warning messages produced by the C compiler. It also issues “lint-specific” warnings about potential bugs and portability problems.

Chapter 6, “dbXtra and dbxtra” describes the graphical debugger **dbXtra** and the character-based debugger **dbxtra**. **dbxtra** supports debugging of both C and C++ products in conjunction with SCO C++ products. This debugger can follow the child or parent process when a fork is executed. **dbxtra** allows for the suspension of the parent debugger while debugging the child, and returning to the parent session after exiting the child. When using the X-based version of **dbXtra**, a programmer can simultaneously debug parent and child processes.

Chapter 7, “sdb: symbolic debugger” describes a debugger that is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled. **sdb** provides a means for displaying variables to be accessed symbolically and viewed in the correct format.

Chapter 8, “adb: absolute debugger” describes a source level debugger for C and assembly language programs. **adb** provides a method for viewing non-ASCII data files with line mode commands. **adb** can single step through a program, allowing program patches to be applied to a program at the source level.

Chapter 9, “C programmer’s productivity tools” teaches you how to use the C Programmer’s Productivity Tools (CPPT) **cscope**, a browser, and **lprof**, a line profiling tool. Step-by-step instructions are provided through basic examples, so you can start using these tools right away. Additional examples demonstrate various options that allow you to make the best use of the tools.

Chapter 10, “make” describes how to use **make** to keep track of file-to-file relationships, the order of command executions, and general file maintenance. **make** automates the steps required to create executable programs and provides a mechanism for ensuring that programs are up-to-date. This tool is especially useful for large programming projects, because it ensures that all necessary compilations are performed, and minimizes the time wasted by unnecessary compilation. It can also be used to advantage with smaller programming tasks.

Chapter 11, “Source code control system (SCCS)” explains how to control and maintain all versions of a project’s source files using the SCCS commands. The Source Code Control System (SCCS) is an integral part of the Development System. The SCCS commands allow you to maintain a history of the changes to your source files. SCCS keeps all previous versions of your source files so that you can trace changes, or even regenerate earlier versions as needed.

Chapter 12, “lex” explains how to create lexical analyzers using the program generator **lex**. Lexical analyzers in programs pick patterns out of complex input and convert these patterns into meaningful values or tokens. The **lex** program generates lexical analyzers, written in C, from given specification files.

Chapter 13, “yacc” explains how to create parsers using the program generator, **yacc**. The **yacc** program generates parsers, written in C, from given specification files. You can use **lex** and **yacc** together to make complete programs.

Chapter 14, “Shared libraries” describes what a shared library is and how to use one to build *a.out* files. It offers suggestions about when to use a shared library, and how to determine whether an *a.out* file uses a shared library. It also explains how to use **mkshlib**(CP), how to write C code for shared libraries, and how to use the tool **chkshlib**(CP) to check the compatibility of versions of shared libraries.

Appendix A, “ANSI implementation-defined behavior” provides the specification of the implementation-defined characteristics and extensions required by the standard as called out in ANSI Document Number X3J11/90-013.

Appendix B, “C compiler diagnostics” contains the text and explanation for all the warning and error messages produced by the compiler.

Typographical conventions

This publication presents commands, filenames, keystrokes, and other special elements in these typefaces:

Example:	Used for:
lp or lp(C)	commands, device drivers, programs, and utilities (names, icons, or windows); the letter in parentheses indicates the reference manual section in which the command, driver, program, or utility is documented
<i>/new/client.list</i>	files, directories, and desktops (names, icons, or windows)
<i>root</i>	system, network, or user names
<i>filename</i>	placeholders (replace with appropriate name or value)
<Esc>	keyboard keys
Exit program?	system output (prompts, messages)
yes or yes	user input
"Description"	field names or column headings (on screen or in database)
Cancel	button names
Edit	menu names
Copy	menu items
File ⇌ Find ⇌ Text	sequences of menus and menu items
open or open(S)	library routines, system calls, kernel functions, C keywords; the letter in parentheses indicates the reference manual section in which the file is documented
\$HOME	environment or shell variables
SIGHUP	named constants or signals
"adm3a"	data values
<i>employees</i>	database names
<i>orders</i>	database tables
buf	C program structures
<i>b_b.errno</i>	structure members

Related documentation

The SCO OpenServer™ Development System includes comprehensive documentation. Books are available in online and/or printed form. Access online books by double-clicking on the Desktop **Help** icon. Printed versions of the books are also available.

The contents of the SCO OpenServer Development System documentation set are illustrated here. In addition to these books, a copy of the *Development System Release and Installation Notes* is provided with the Development System software.

	Reference	Guide
UNIX® Applications	• C++ Library Manual • C++ Product Reference Manual • Programmer's Reference Manual (2 Volumes)	• C++ Selected Readings • Character User Interfaces Guide • Developer's Topics • Programming Tools Guide
Networking	• Network Programmer's Guide and Reference	
Graphical	• X Window System Programmer's Reference • OSF/Motif Programmer's Reference	
Optional	• Device Driver Writer's Guide	

The books included with the Development System are:

C++ Library Manual

describes the three C++ class libraries and their use.

C++ Product Reference Manual

provides a complete definition of the C++ language supported by Release 3.1 of the SCO® C++ Language System.

C++ Selected Readings

contains papers describing aspects of the C++ language.

Character User Interfaces Guide

introduces the facilities available for developing user interfaces, and gives detailed instructions for using **curses(S)** and the Extended Terminal Interface (ETI), and for writing user interfaces that can be run on ASCII terminals. This book includes numerous examples of **curses** programs.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include examples illustrating features that are extensions provided by SCO software.

Network Programmer's Guide and Reference

describes the libraries and application program interfaces (APIs) available (Internet domain sockets, TLI, XTI, ONC, RPC, and XDR), for developing applications that use the services of network-oriented communication protocols (TCP/IP, IPX/SPX, and NetBIOS). It also describes how to use UNIX domain sockets for interprocess communication, and an SMUX API to create SMUX peers for SNMP agents. The (NC), (NET), (NS), (SAPD), (SLIB), and (SSC) reference manual pages are included in this book.

OSF/Motif Programmer's Reference

consists of the OSF/Motif toolkit, window manager, and user interface language commands and functions: (Xm) reference manual pages.

Programmer's Reference Manual

a two-volume set of reference manual pages for the Development System, including sections (CDMT), (CP), (FP), and (S).

Programming Tools Guide

provides useful information about programming tools and their use in developing software, including implementation-specific details about the ANSI-conforming C compiler provided with the Development System. It also contains information about debugging tools and their use in tracking down and eliminating problems in C and assembly language programs, and information on using **dbXtra**[™] to debug C and C++ programs in an OSF/Motif[™] based windowing environment.

Release and Installation Notes

highlights new features, product changes, documentation changes, and known limitations for this release of the Development System. It also contains the installation notes.

X Window System Programmer's Reference

provides X Window System[™] programming commands and functions: X Library (XS), X Toolkit (Xt), X Miscellaneous Utilities (Xmu), and X Extensions (Xext) reference manual pages for programmers.

In addition, the following book is available on request:

Device Driver Writer's Guide

provides guide and reference information about writing device drivers for SCO systems. This book is available separately.

Commercial books and articles

A number of fine books and articles are published commercially that discuss how to develop software on the UNIX operating system. The *Developer's Topics* includes a large "Bibliography" that will interest users of the Development System.

In particular, we recommend that developers have the following standard textbooks on their shelves:

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition.

One of the following general books about programming on UNIX operating systems:

Donald Lewine, *POSIX Programmer's Guide*.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *Advanced Programming in the UNIX Environment*.

W. Richard Stevens, *UNIX Network Programming*. Useful for programmers who are writing distributed applications.

Maurice Bach, *Design of the UNIX Operating System*. Useful for programmers who want to thoroughly understand UNIX system internals.

See the "Bibliography" in *Developer's Topics* for full citations if you are not familiar with the books listed above.

How can we improve this book?

What did you find particularly helpful in this book? Are there mistakes in this book? Could it be organized more usefully? Did we leave out information you need or include unnecessary material? If so, please tell us.

To help us implement your suggestions, include relevant details, such as book title, section name, page number, and system component. We would appreciate information on how to contact you in case we need additional explanation.

To contact us, use the card at the back of the *SCO OpenServer Handbook* or write to us at:

Technical Publications
Attn: CFT
The Santa Cruz Operation, Inc.
PO Box 1900
Santa Cruz, California 95061-9969
USA

or e-mail us at:

techpubs@sco.com or ... *uunet!sco!techpubs*

Thank you.

Chapter 1

Using programming tools

The Development System includes a powerful set of programming tools to compile, analyze, and debug programs. This chapter describes how to use some of these tools to develop a small sample program. By the end of the chapter, you will have a working knowledge of **SCCS**, **cb**, **cscope**, **lint**, **cflow**, **cxref**, **cc**, **make**, and **dbXtra**.

For the purposes of this example, program development proceeds as follows:

- create the source code
- archive a copy of the original code
- analyze the code
- compile the code
- test, debug and retest the code
- archive any changes made during testing and debugging

The Development System provides tools to help at each stage of the development process. This chapter describes their use and also provides general information and guidelines for structuring programs and applications.

Creating the source code

Program development begins with the creation of code from specifications. Sometimes, the code has already been written and must only be tested and debugged. This tutorial includes a sample program.

The sample program, *testcase.c*, was written to these specifications:

- Read input from **stdin** and write output to **stdout**.
- For each line of input, output the words that comprise the line, one word to a line. A word is defined to be any sequence of alphabetic characters separated by a white space character.
- If the optional argument, **-c**, where **c** is a character, is specified, output only those words containing the character.

The file *testcase.c* contains the following code:

```

1      #include <stdio.h>
2      #include <ctype.h>
3      #include <string.h>
4
5      /* Manifests for state machine to parse input line. */
6      #define WORD      0
7      #define IGNORE    1
8
9      #define BUFFERSIZE 5000
10
11     /* Globals, used by both subroutines. */
12     char    *Words[BUFFSIZE/2]; /* Worst case, single letters. */
13     int      WordCount;
14
15     /* Walk through the array of words, find those with the
16      * matching character, printing them on stdout. Note that
17      * the null character will match all words. */
18     void PrintWords(wc, match)
19     int wc; /* Number of words in Words[] */
20     char match; /* Attempt to match this character. */
21     {
22         register int ix; /* Index in Words[] */
23         register char *cp; /* Pointer for searching. */
24         for(ix = 0; ix < wc; ix++) {
25             cp = Words[ix];
26             /* Try to match the given character.
27              * Scan the word, attempting to match,
28              * or until the end of the word is found. */
29             while((*cp) && (*cp++ != match));
30             if (*cp == match) /* Found a match? Write the word on stdout. */
31                 (void) printf("%s\n", Words[ix]); } return; }
32
33     /* Find words in the given buffer. The Words[] array is set
34      * to point at words in the buffer, and the buffer modified
35      * with NULL characters to delimit the words. */
36     int GetWords(buf)
37     char buf[]; /* The input buffer. */
38     {
39         register char *cp; /* Pointer for scanning. */
40         int end = strlen(buf); /* Length of the buffer. */

```

```

39         register int      wc = 0;           /* Number of words found. */
40         int      state = IGNORE;           /* Current state. */
41         /* For each character in the buffer. */
42         for(cp = &buf[0]; cp < &buf[end]; cp++) {
43             /* A simple state machine to process
44              * the current character in the buffer.
45              */
46             switch(state) {
47                 case IGNORE:
48                     if (!isspace(*cp)) {
49                         Words[wc++] = cp; /* Just started a word?
50                         Save it. */
51                         state = WORD; /* Reset the state. */ } break;
52                 case WORD:
53                     if (isspace(*cp)) {
54                         *cp = '\0'; /* Just completed a word?
55                         terminate it. */
56                         state = IGNORE; /* Reset the state. */ } break; } }
57         return wc; /* Return the word count. */ }
58
59     int main(argc, argv) int argc; char *argv[]; { char   buf[BUFSIZ], match;
60     /* Check command line arguments. */
61     if (argc < 2) match = ' ';
62     /* No command line argument, match all words. */
63     else match = *++argv[1]; /* Match the char after the first - */
64     /* Until no more input on stdin. */
65     while(gets(buf) != (char *)NULL) {
66         WordCount = GetWords(buf); /* Parse the input buffer. */
67         PrintWords(WordCount, match); /* Print the matching words. */ }
68     return(0); /* Return success to the shell. */ }

```

Archiving the code

During a software development project it is often desirable to track the changes made to the source code, who made them, and the reasons for the change. In addition, if more than one person is working on the project, it is important that two or more people don't try to make changes to the code at the same time.

SCCS (Source Code Control System) is a maintenance and enhancement tracking tool that provides these capabilities. When changes are made, SCCS identifies and stores them in the file with the original source code or documentation. Any previous version of the file can be reconstructed for inspection or for additional modification. History data can be stored with each version to help remember why, when, and by whom changes were made.

Storing source files under SCCS also prevents two people from accidentally modifying the same file at the same time. To change a file, it must explicitly be checked out for editing. If a version of the file is already checked out for editing, other developers are prevented from checking it out.

If you want a copy of the original version of *testcase.c* at a later date, it is a good idea to place it under SCCS. This will let you track the changes made to *testcase.c* as it proceeds through the development procedure. To check *testcase.c* into SCCS, issue the following command:

```
admin -itestcase.c -r1 s.testcase.c
```

The following message is displayed:

```
UX:admin:WARNING: No id keywords (cm7)
```

NOTE **admin** issues a warning message: UX:admin:WARNING:No id keywords (cm7). Study the SCCS documentation for information about how to supply an ID keyword.

SCCS creates a file with the name specified on the command line, in this case *s.testcase*. SCCS adds some lines to the beginning of the file:

```
^Ah22947
^As 00117/00000/00000
^Ad D 1.1 94/10/07 13:26:29 username 1 0
^Ac date and time created 94/10/07 13:26:29 by username
^Ae
^Au
^AU
^At
^AT
^AI 1

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

NOTE In the previous output, the **^A** denotes a (Ctrl)A.

Once *s.testcase.c* is under SCCS control, the file must be checked out for editing before any changes can be made. To make changes to *s.testcase.c*, check it out for editing with the following command:

```
rm -e s.testcase.c
get -e s.testcase.c
```

SCCS responds with the following message:

```
1.1
new delta 1.2
66 lines
```

The result is a file called *testcase.c*, which is the file we will edit, and a file called *p.testcase.c*, which SCCS uses to control file access. If anyone tries to access the file once it has been checked out for editing, SCCS issues an error message indicating that the file is already checked out and by whom.

Analyzing the code

If you are working on source code that you didn't write, the purpose of various constructs may not be clear. The Development System provides a number of tools to help make sense out of the source code. In this section, we will use the following tools to analyze the *testcase.c* source code:

- cb** a tool to format C source code
- cscope** a tool to examine functions contained with a program
- lint** a tool to identify code that may not pass through the compiler without an error message or that may be coded using questionable coding practices
- cflow** a tool to identify the nesting levels of functions within a program
- cxref** a tool to create a cross reference of functions and symbols used within a program

Using cb

When developing software, it is easy to forget to format the code so it is both easy to read and the program logic is apparent. Running a program through **cb** (C program beautifier) corrects this problem. For example, some of the code in *testcase.c* is less than beautiful:

```

57     int main(argc, argv) int argc; char *argv[]; { char   buf[BUFFERSIZE], match;
58     /* Check command line arguments. */
59     if (argc < 2) match = ' ';
60     /* No command line argument, match all words. */
61     else  match = ***argv[1];      /* Match the char after the first - */

```

While this code fragment uses legal C syntax that is readily understood by the compiler, it is hard for a human to understand. When the code is run through **cb** with the **-s** option specified, the reformatted code is easier to follow and understand:

```

73     int      main(argc, argv)
74     int
75     char      *argv[];
76     {
77         char   buf[BUFFERSIZE], match;
78         /* Check command line arguments. */
79         if (argc < 2)
80             match = ' ';
81         /* No command line argument, match all words. */
82         else
83             match = ***argv[1]; /* Match the char after the first - */

```

For more information, see **cb(CP)**.

Using **cscope**

The purpose of **cscope** is to build a database containing information about functions, function calls, macros, and variables. **cscope** allows you to look at functions and examine the lines of code around those functions. In the *testcase.c* example, all the functions are contained in the one file. If, however, functions that are called from *testcase.c* were located in other files (excluding system library functions), **cscope** would allow you to look directly at those functions without exiting **cscope**. While using **cscope** on *testcase.c* might not be extremely helpful, it may still be of interest.

To examine the functions within *testcase.c*, run **cscope** as follows:

```
cscope testcase.c
```

To move the cursor to a menu option, press the <Return> key. To examine the function **main**, move the cursor to "List functions called by this function:", enter **main**, and press <Return>:

```
cscope                                     Press the ? key for help
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function: main
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

The display lists the three functions called by **main**: **gets**, **GetWords** and **PrintWords**. Displayed along with the function names is the source code line that contains the function.

```
Functions called by this function: main
```

```
File      Function  Line
1 testcase.c gets      85 while(gets(buf) != (char *)NULL) {
2 testcase.c GetWords   86 WordCount = GetWords(buf);
3 testcase.c PrintWords 87 PrintWords(WordCount, match); }
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function: GetWords
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

By examining the **GetWords** function, we can move deeper into the code:

```

Functions called by this function: GetWords

File      Function Line
1 testcase.c strlen  45 int end = strlen(buf);
2 testcase.c isspace 55 if (!isspace(*cp)) {
3 testcase.c isspace 61 if (isspace(*cp)) {

List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:

```

In analyzing the results, we see that **GetWords** calls three functions displayed with their line numbers in the source code.

Using **cscope** is particularly helpful when examining a large or complex program that has many files that span several directories. For more information, see Chapter 9, “C programmer’s productivity tools” (page 285) and **cscope**(CP).

Using lint

The **lint**(CP) program examines source code to identify sections of code that the compiler may flag as incorrect. In many cases, **lint** identifies as incorrect sections of code that the compiler accepts but that uses poor programming practice that may affect portability and robustness.

To find out how portable and robust our sample program is, run *testcase.c* through **lint** with the command:

```
lint testcase.c
```

lint produces the following display:

```

static unused
(61) __c

declared global, could be static
PrintWords      testcase.c(21)
GetWords         testcase.c(43)
Words            testcase.c(12)
WordCount        testcase.c(13)

```

The warning messages indicate that several functions could have been declared. It also points that variable ‘__c’ has been declared but never used. This is the result of the variable being declared in the system header file *ctype.h* but never used by *testcase.c*. We also never use the macro defined in *ctype.h* in *testcase.c*, which references that symbol. For more information on using **lint**, see Chapter 5, “lint analyzer” (page 153) and **lint**(CP).

Using cflow

The **cflow** command presents a list of the functions and function calls within a program. This list can be helpful when debugging a program, especially if there are external references. The information displayed by **cflow** indicates the source filename and the line number of the function, and lists additional functions that are in turn called by that function. To see a list of the functions called within *testcase.c*, we can run **cflow**:

```
cflow testcase.c
```

```
1      main: int (), <testcase.c 76>
2          gets: <>
3          GetWords: int (), <testcase.c 43>
4              strlen: <>
5          PrintWords: void (), <testcase.c 21>
6              printf: <>
```

Output is displayed in the following fields:

line number	the number of the line in the output
tabs	the level at which the symbol is defined
symbol name	the name of the symbol under consideration followed by a colon
symbol information	the type of the symbol and the arguments to the symbol, if any
location	the name of the file and the number of the line within the file in which the symbol is defined

In analyzing the output, we see that the function **main** is the first function called by *testcase.c* and is defined on line 76 as shown in the angle brackets (<>). Notice the function **main** returns a type **int** and it calls the functions **gets**, **GetWords**, and **PrintWords**. The <> next to the **gets** function are empty because **cflow** only looks in the source file to resolve function call references. The **gets** function is not in the source code (*testcase.c*) and, therefore, **cflow** does not have any information on it. The function **main** calls two additional functions, **GetWords** and **PrintWords**. Both of these functions are found in the source file and, therefore, their locations are displayed. The two additional functions, **strlen** and **printf**, called respectively from **GetWords** and **PrintWords**, do not exist within *testcase.c*; hence, no line number information is available for them.

As a result of analyzing *testcase.c* with **cflow**, we now know the names of all of the functions referenced in the source file and the location of their definitions in the source file. If we need to change any of them, they should be easy to find. For a larger more complex program, you can make decisions on where best to optimize code, either by reducing the number of function calls or by

identifying obsolete functions and removing them. For more information on using **cflow**, see **cflow(CP)**.

Using **cxref**

The **cxref** tool produces a list of all functions and symbols (**auto**, **static** and **global**) that can be used as a cross-reference table for large or complex programs. **cxref** is more informative than **cflow** because not only are the function calls that exist within the source file displayed, with line numbers, but all external function calls also have that information. To see a list of the function calls and symbol references used by *testcase.c*, run **cxref**:

```
cxref testcase.c
```

The output is displayed in the following format:

```
testcase.c:
NAME      FILE      FUNCTION  LINE
BUFFERSIZE  testcase.c  ---      9*   12   77
BUFSIZ      stdio.h     ---      67*
EOF         stdio.h     ---      56*
FILENAME_MAX stdio.h     ---      81*
FOPEN_MAX   stdio.h     ---      74*
GetWords    testcase.c  ---      43*   86
IGNORE      testcase.c  ---      7*   47   54   63
L_ctermid   stdio.h     ---      97*
L_cuserid   stdio.h     ---      98*
L_tmpnam    stdio.h     ---     105*
M_I386      predefined  ---      0*
M_UNIX      predefined  ---      0*
M_XENIX     predefined  -       0*
NULL        stdio.h     ---      52*
            string.h     ---      34
            testcase.c  ---      85
P_tmpdir    stdio.h     ---     100*
PrintWords  testcase.c  ---      21*   87
SCO_DS      predefined  ---      0*
SEEK_CUR    stdio.h     ---      60*
SEEK_END    stdio.h     ---      61*
SEEK_SET    stdio.h     ---      59*
TMP_MAX     stdio.h     ---      64*
WORD        testcase.c  ---      6*   57   60
WordCount   testcase.c  ---     13*   86=   87
Words       testcase.c  ---     12*   25   31   56=
_A          ctype.h     ---      43*
_B          ctype.h     ---      36*
.
.
.
wscanf     stdio.h     ---     262-
```

The first column lists the name of the symbol. The second column lists the name of the file(s) in which the symbol appears. The third column lists the name of the function in which the symbol appears, and the fourth column provides the line number within the file in which the symbol appears.

To illustrate, let's examine the symbol **BUFFERSIZE** in the **cxref** output. The first entry names the symbol, **BUFFERSIZE**, and the file where it is declared, indicated by an asterisk (*) next to the line number. The second entry indicates its first use in *testcase* denoted by only a line number. The third and final entry for **BUFFERSIZE** indicates its use in the function **main** of *testcase.c* on line 77.

The listing produced by **cxref** is helpful when debugging a large complex program. Using the **cxref** output, external function calls can be tracked down with ease. For more information, see **cxref**(CP).

Archiving changes made during the analysis

During our analysis of the code for *testcase.c*, we modified the sample program; specifically, we beautified the code using **cb**. Because it is always a good idea to track changes to a project, we should check the modified code back into **SCCS**.

To check the modified code back into **SCCS**, issue the command:

```
delta s.testcase.c
```

Supply a brief comment describing the changes in response to the **comment?** prompt.

After the comment has been added, **SCCS** responds with a brief message indicating what has happened to the file:

```
UX:delta: WARNING: No id keywords (cm7)
1.2
57 inserted
31 deleted
35 unchanged
```

To list the history of changes to the *testcase* file, issue the following command:

```
prs s.testcase.c
```

You see:

```
D 1.2 94/10/07 14:32:51 username 2 1      00000/00000/00117
MRs:
COMMENTS:
sample editing changes made to this file

D 1.1 94/10/07 13:26:29 username 1 0      00117/00000/00000
MRs:
COMMENTS:
date and time created 94/10/07 13:26:29 by username
```

There are other tools available to retrieve an earlier version of the file, compare two versions of the file, and so forth. The “Source Code Control System (SCCS)” chapter in the *Programming Tools Guide* includes a detailed explanation and tutorial for using SCCS.

Compiling the code

Because we have already performed an analysis of the program, the program should compile without errors. Compile *testcase* with the following command line:

```
cc -o testcase testcase.c
```

As expected, the compiler compiles the program without errors and names the resulting executable file *testcase*.

The compiler supports a large number of options that can be used to control how the program is compiled and linked. One that is commonly used is the **-l** option, which specifies the libraries to link in. The default library is the standard C library; if your program uses any routines from other libraries, you will need to specify those libraries on the compile line. The section (S) manual pages indicate which library is required for which routine.

For more information, see Chapter 2, “C compilation system” (page 33), Chapter 3, “C language compiler” (page 83), and **cc(CP)**.

Using make

While the procedure for compiling our sample program is relatively simple, the procedure for compiling larger projects can be considerably more complex. Fortunately, the Development System provides a tool to automate the procedure. There are a number of reasons for using the **make(CP)** utility to automate compilation. The manual page the **cc(CP)** describes a large number of options for the compilers. In some instances, the appropriate assortment of libraries, indicated with the **-l** option, needs to be selected each time a program is compiled. Remembering all the options can become tedious; that is

one reason for creating a makefile to use when compiling and linking the program. Another advantage of a makefile is that when several files are linked together into one executable file, **make**(CP) only recompiles those files that have been modified (or “touched”) since the last link. This reduces the time required to recompile code.

We can use the following makefile to automate the compilation of *testcase.c*:

```
#
# Makefile for testcase.c
#

# Compile testcase.c normally

testcase: testcase.c
    cc -o testcase testcase.c

# Compile testcase.c with debugging information

testcasedb: testcase.c
    cc -otestcasedb -g testcase.c
```

The above makefile provides two ways of compiling *testcase.c*. Invoking the makefile with the following command line compiles the file with no debugging information contained in the binary:

make *testcase*

However, if **make** is invoked with the command line below, debugging information is included.

make *testcasedb*

For more information, see Chapter 10, “make” (page 315) and **make**(CP).

Testing the code

Even though *testcase* has compiled without errors, we cannot be sure that the program logic is correct. The only way to ensure that the logic is correct and the program does what is intended, is to test it thoroughly. Any errors revealed by testing must, of course, be corrected. Because *testcase* is a small, simple program, we can test it by simply running it on selected input and verifying that the output is what we expect. Larger, more complex programs benefit from a more rigorous testing procedure.

Test *testcase* with the following command line and input (bold typeface indicates user input):

```
testcase
this is a test of the testcase program
this
is
a
test
of
the
testcase
program
```

With no character argument specified, *testcase* responds with one word per line, as expected. Test the option to *testcase* with the following command line and input:

```
testcase -t
this is a test of the testcase program
```

In this example, with the *-t* as an argument, *testcase* should have printed out only words with the character *t* in it; it did not. This deserves further exploration. Run *testcase* with the following input:

```
testcase -t
this is a ttest of the testcase program
ttest
```

testcase seems to be printing out words that have double occurrences of the specified character, *t*. Try it with the following input just to make sure:

```
testcase -t
attention, the ttime is at hand
attention,
ttime
```

That seems to be the problem with our program. Now that we have identified the problem, we have to find out where in the source code the problem occurs. Fortunately, the Development System provides the tools to debug source code.

Debugging the code

The SCO Development System supports three debuggers, **adb**, **sdb** and **dbxtra** that can be used to help locate errors in source code. Each debugger has its special uses, but in most cases choosing between them is largely a matter of taste.

Using dbxtra

To debug *testcase*, use **dbxtra**. For *testcase* to be debugged, it must be compiled with the **cc -g** option specified. This option specifies that the compiler should include symbolic information in the binary. Compile *testcase.c* with debugging information using **make**:

```
make testcasedb
```

To initiate **dbxtra** issue the following command:

```
dbxtra testcasedb
```

The following **dbxtra** screen is displayed.

```

77      char    buf[BUFFERSIZE], match;
78      /* Check command line arguments. */
79      if (argc < 2)
80          match = '\0';
81      /* No command line argument, match all words. */
82      else
83          match = ***argv[1];      /* Match the char after the first
t - */
84      /* Until no more input on stdin. */
85      while (gets(buf) != (char *)NULL) {
86          WordCount = GetWords(buf); /* Parse the input buffer. */
87          PrintWords(WordCount, match); /* Print the matching words
C forwep.c file: testcase.c                               Func: +       File: 26718
dbxtra version 3.2.5Ja IR391
Reading symbolic information ...
Type 'help' for help.
(dbxtra) 
```

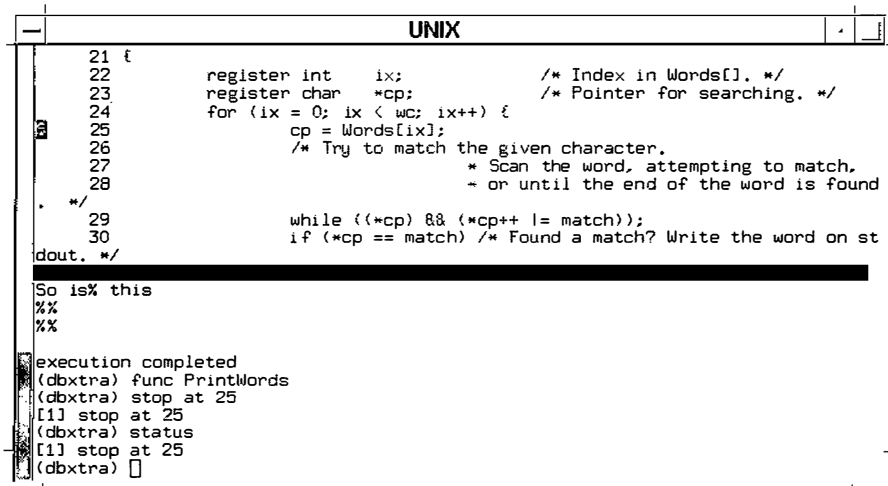
To begin debugging *testcase*, it must be run under **dbxtra**'s control. To execute *testcase* within **dbxtra**, issue the **dbxtra run** command:

```
(dbxtra) run -%
This is a test
So is% this
%%
%%
<Ctrl>d
execution completed
(dbxtra)
```

In this example, we had *testcase* look for the argument *%*, which it failed to match in the second line, but did in the third.

The **cp** pointer points to the characters in the input buffer. Therefore, the **cp** pointer might be a place to start searching for the problem. Because we know the **cp** pointer is used in the **PrintWords** function from the **cxref** output, examining the function **PrintWords**, and monitoring **cp**, might lead us to discover why *testcase* is not behaving correctly. At this point, a breakpoint should be set in the **PrintWords** function. This allows control over the program's execution, enabling us to monitor the **cp** pointer. The following commands set a breakpoint and check to ensure that breakpoint has been set:

```
(dbxtra) func PrintWords
(dbxtra) stop at 22
[1] stop at 22
(dbxtra) status
[1] stop at 22
(dbxtra)
```



```

UNIX
21 {
22     register int    ix;          /* Index in Words[], */
23     register char  *cp;          /* Pointer for searching. */
24     for (ix = 0; ix < wc; ix++) {
25         cp = Words[ix];
26         /* Try to match the given character.
27          * Scan the word, attempting to match,
28          * or until the end of the word is found
29          */
29         while ((*cp) && (*cp++ != match));
30         if (*cp == match) /* Found a match? Write the word on st
31             dout. */
32
33     }
34 }
35
36 So is% this
37 %%
38 %%
39
40 execution completed
41 (dbxtra) func PrintWords
42 (dbxtra) stop at 25
43 [1] stop at 25
44 (dbxtra) status
45 [1] stop at 25
46 (dbxtra)

```

Notice the **S** beside line 25; this indicates that a breakpoint has been set.

The next action is to rerun the program with the breakpoint in effect:

```
(dbxtra) rerun  
Running: testcasedb -%  
% a b c  
[1] stopped in PrintWords at line 25  
(dbxtra)
```

The **where** command displays the current location in the stack frame:

```
(dbxtra) where  
PrintWords(wc = 4, match = '%'), line 25 in "testcase.c"  
main(argc = 2, argv = 0x7ffff81c, 0x7ffff828), line 87 in "testcase.c"  
(dbxtra)
```

The following two **dbxtra** commands display the type of variable (character) and what the match character is (%):

```
(dbxtra) whatis match  
register char match;  
(dbxtra) print match  
'%'  
(dbxtra)
```

Because the breakpoint was set on line 25, the pointer **cp** has not been set yet. To execute the next step and set **cp**, enter **next** on the **dbxtra** command line. If the pointer **cp** is checked now:

```
(dbxtra) next  
(dbxtra) print cp  
"%"
```

It displays the correct character. If one more line of code is executed, the **cp** pointer should display the same character:

```
(dbxtra) next  
(dbxtra) print cp  
" "  
(dbxtra)
```

Notice that there is no value currently in **cp**. This may be the problem; it would seem that the **cp** pointer is being incremented even if a match is found. The result is that only double occurrences of the match variable character produces the correct behavior.

The next step in debugging *testcase* is to check if the problem assessment is correct. *testcasedb* needs to be rerun, and the **cp** pointer set:

```
(dbxtra) rerun
Running: testcasedb -%
% a b c
[1] stopped in PrintWords at line 25
(dbxtra) next 2
(dbxtra) print cp
" "
(dbxtra)
```

Notice the **cp** pointer is indicating a null character. The next step is to reset the **cp** pointer.

The **assign** command allows variable values to be changed. In this example, by subtracting 1 from **cp**, the correct character variable should be displayed.

```
(dbxtra) assign cp = cp - 1
(dbxtra) next
(dbxtra) next
%
(dbxtra)
```

This indicates that the problem assessment was correct. The pointer was being incremented regardless of whether or not a match condition existed.

The last step is to delete the breakpoint and complete the program's execution:

```
(dbxtra) cont
[1] stopped in PrintWords at line 25
(dbxtra) delete 1
(dbxtra) cont
<ctrl>d
execution completed
(dbxtra) q
```

Correcting the code

To fix the problem within the *testcase.c* code, *testcase.c* must be checked out of SCCS, changed, recompiled, and retested. This ensures that the logic problem located through the debugger is the only problem with the code.

Check *s.testcase* out of SCCS:

```
get -e s.testcase
```

SCCS responds with the following message:

```
1.2
new delta 1.3
92 lines
```

Edit *testcase.c* by locating the following line in the **PrintWords** function:

```
while((*cp) && (*cp++ != match))
if (*cp == match) /* Found a match? Write the word on stdout. */
    (void) printf("%s\n", Words[ix]); } return; }
```

Change the line to read as follows:

```
while((*cp) && (*cp != match))
    ++cp;
if (*cp == match) /*Found a match? Write the word on stdout. */
    (void) printf("%s\n", Words[ix]); } return; }
```

Save the file, exit the editor, and check *testcase* back into SCCS:

```
delta s.testcase.c
```

The new version of *testcase.c* now needs to be recompiled. Execute the makefile to ensure all compiling and linking steps are executed:

```
make testcase
```

Testing the correction

The program *testcase* should be retested to ensure the corrected pointer causes the correct behavior.

```
testcase
this is a test of the testcase program
this
is
a
test
of
the
testcase
program
```

With no character argument specified, *testcase* responds with one word per line. This is the correct behavior.

```
testcase -t
this is a test of the testcase program
this
test
the
testcase
```

In this example, with the **-t** as an argument, *testcase* has printed out, one word per line, only words with the letter **t**. This is the correct behavior. The assumption that the pointer was incrementing regardless of whether or not a match condition existed was correct. Changing it solved the problem within *testcase*.

Program structure and guidelines

As illustrated through *testcase*, specification of applications must define the actual work to be accomplished. There are many issues that should be considered in designing applications written in C, including:

- modularity Good C programming style suggests that large applications be coded in several small component parts. Several C codes can be linked together to form one executable file; use the **make**(CP) utility to simplify the process.
- functions Functions that could run parallel to each other should be coded in separate executable files. This is especially useful if you ever port the application to a multiprocessor distributed system.
- archives If your application uses certain routines frequently in different programs, it is advisable to create an archive library or shared library for those routines. Shared libraries usually reduce the amount of disk space and memory occupied by the executable modules and enable you to update all programs that call a certain routine without recompiling those programs. There are limits to the number of shared libraries that any executable program can access. For more information, see Chapter 14, "Shared libraries" (page 443).

Program structure

The descriptions that follow use the line numbers within *testcase* to indicate their location. This is an example of correct programming structure.

```

1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4
5 /* Manifests for state machine to parse input line. */
6 #define WORD 0
7 #define IGNORE 1
8
9 /* Globals, used by both subroutines. */
10 char *Words[BUFFERSIZE/2]; /* Worst case, single letters. */
11 int WordCount;
12
13 /* Walk through the array of words, find those with the
14  * matching character, printing them on stdout. Note that
15  * the null character will match all words. */
16 void PrintWords(wc, match)
17 int wc; /* Number of words in Words[] */
18 char match; /* Attempt to match this character. */
19 {
20     register int ix; /* Index in Words[] */
21     register char *cp; /* Pointer for searching. */
22     for (ix = 0; ix < wc; ix++) {
23         cp = Words[ix];
24         /* Try to match the given character.
25          * Scan the word, attempting to match,
26          * or until the end of the word is found. */
27         while ((*cp) && (*cp++ != match))
28             ++cp;
29         if (*cp == match) /* Found a match? Write the word on stdout. */
30             (void) printf("%s\n", Words[ix]);
31     }
32     return;
33 }
34
35

```

```

36 /* Find words in the given buffer. The Words[] array is set
37 * to point at words in the buffer, and the buffer modified
38 * with null characters to delimit the words. */
39 int      GetWords(buf)
40 char      buf[];                                /* The input buffer. */
41 {
42     register char      *cp;                        /* Pointer for scanning. */
43     int      end = strlen(buf); /* Length of the buffer. */
44     register int      wc = 0;                      /* Number of words found. */
45     int      state = IGNORE;                       /* Current state. */
46     /* For each character in the buffer. */
47     for (cp = &buf[0]; cp < &buf[end]; cp++) {
48         /* A simple state machine to process
49          * the current character in the buffer.
50          */
51         switch (state) {
52             case IGNORE:
53                 if (!isspace(*cp)) {
54                     Words[wc++] = cp; /* Just started a word? Save it. */
55                     state = WORD; /* Reset the state. */
56                 }
57                 break;
58             case WORD:
59                 if (isspace(*cp)) {
60                     *cp = '\0'; /* Just completed a word? terminate it. */
61                     state = IGNORE; /* Reset the state. */
62                 }
63                 break;
64             }
65     }
66     return wc; /* Return the word count. */
67 }
68
69
70
71 int      main(argc, argv)
72 int      argc;
73 char      *argv[];
74 {
75     char      buf[BUFFERSIZE], match;
76     /* Check command line arguments. */
77     if (argc < 2)
78         match = ' ';
79     /* No command line argument, match all words. */
80     else
81         match = *++argv[1]; /* Match the char after the first - */
82     /* Until no more input on stdin. */
83     while (gets(buf) != (char *)NULL) {
84         WordCount = GetWords(buf); /* Parse the input buffer. */
85         PrintWords(WordCount, match); /* Print the matching words. */
86     }
87     return(0); /* Return success to the shell. */
88 }

```


Lines 1-3 Begin the code with **#include** statements for the header files that are required. Each section (S) page lists the header files that are required for that particular call; use a **#include** line for each one that is needed by the calls you are using.

There are some rules about the order in which header files are included; see “Header files” (page 31). Application-specific header files should always be included after the standard header files.

Lines 74-91 The **main()** routine controls the order in which the routines execute. It is best to keep **main()** fairly succinct. Keep complex functionality in subroutines, located either before or after **main()**. This makes it easier to look at **main()** to get the general notion of program flow. If the application has several programs that call some of the same routines, those shared routines can be located in an archive or shared library for retrieval in all programs.

Line 74 The *argc* and *argv* parameters to **main()** capture arguments given on the command line. When a program begins execution, *argc* contains the argument count and *argv* is an array of pointers to the argument values. An argument is stored as a null-terminated string. The argument count is never less than 0.

Lines 18-70 Subroutines are coded outside the **main()** routine. Some people prefer to put them before **main()**, and others prefer to put them afterwards. Either approach works equally well.

Each subroutine should have a block comment preceding it to describe the subroutine and provide information about any non-obvious coding that may have been used. Well-written comments will significantly reduce the effort required to maintain the code in the future. Good C programming style dictates that block comments are not spread over several lines to the right of the actual code. Comments can be put on the line with the code, but these should be short comments that are specific to that line of code, such as shown in lines 17 and 18.

Header files

Header files contain information such as the following:

- global **defines** that may need to be changed or that are shared by several programs and/or drivers
- structure definitions
- macro definitions
- hardware-dependent information that may need to be modified when porting to another system

Header files are called on **#include** lines at the beginning of the code. They must conform to C programming style, since they are accessed as part of the code by the compiler. In general, **#include** statements are placed at the beginning of the code to ensure that they affect the entire code. However, if the intended scope of a particular **#include** line is only specific to a few lines of code, that **#include** line can be located anywhere throughout the code.

The Development System contains many header files under the */usr/include* directory. Header files accessed only by user-level code are not in the */usr/include* directory itself, and header files accessed by drivers and other kernel-level code are in the */usr/include/sys* directory. All header files created specifically for an application should be stored in that source tree so they can be retrieved easily after a software upgrade replaces the contents of the */usr/include* tree. This is important if the product is to be distributed to all users.

Error handling

All programs should be coded to handle any errors that could arise; failure to do this can corrupt your data. Errors can result from user errors or system failures. For the purposes of this discussion, we will consider an error to be anything that happens that should not have happened.

Chapter 2

C compilation system

This chapter describes the UNIX operating system tools used to generate an executable program from C language source files.

The first section, “Compiling and linking” (page 34) details the command line syntax used to produce a binary representation of a program — an executable file. It concentrates on the options to the `cc(CP)` command that control the process in which object files are created from source files, then linked with each other and with the library functions that are called in your program. The major focus of the section is on static vs. dynamic linking; how each model is implemented and invoked, and its relative merits.

Standard libraries are the focus of the second section of the chapter, “Libraries and header files” (page 63). Because the C language contains no intrinsic input/output facility, I/O must be carried out by explicitly called functions. The UNIX operating system, the functions that perform these and other high-level tasks have been standardized and grouped in libraries. They are convenient, portable, and optimized for your machine. The contents of some important standard libraries are described later in this chapter.

Header files contain definitions and declarations that serve as the interface between your program and the functions in these libraries. They also contain several functions, such as `getc()` and `putc()`, that actually are defined as macros. The manual page will generally tell you whether what you are using is a macro or a function. Macros and functions are both used in the same way in your program. The descriptions of standard libraries in this chapter show the header files that you need to include in your program if you call a function in those libraries. The manual page for each function also lists the required header files. “How to use library functions” (page 64) shows how to use library functions in your program and how to include header files.

Compiling and linking

The C compilation system consists of a compiler, assembler, and link editor. The `cc` command invokes each of these components automatically unless you use command line options to specify otherwise. An executable C program is created with the following four steps:

1. The preprocessor component of the compiler reads lines in your source files that direct it to replace a name with a token string (**#define**), perhaps conditionally (**#if**, for example). It also accepts directives in your source files to include the contents of a named file in your program (**#include**). Included header files for the most part consist of **#define** directives and declarations of external symbols, definitions and declarations that you want to make available to more than one source file. See “Libraries and header files” (page 63) for details.
2. The compiler proper, `cc(CP)`, translates the C language code in your source files, which now contain the preprocessed contents of any included header files, into assembly language code.
3. The assembler, `as(CP)`, translates the assembly language code into the machine instructions of the computer your program is to run on. These instructions are stored in object files that correspond to each of your source files. Each object file contains a binary representation of the C language code in the corresponding source file. Object files are made up of sections, of which there are usually at least two. The text section consists mainly of program instructions. Text sections normally have read and execute, but not write, permissions. Data sections normally have read, write, and execute permissions.
4. See Chapter 7, “Common Object File Format (COFF)”, and Chapter 8, “ELF object files” in *Developer’s Topics* for details of the supported object file formats.
5. The link editor, `ld(CP)` links these object files with each other and with any library functions that you have called in your program. The link editing model you have chosen determines when the library functions are linked:
 - An archive, or statically linked library, is a collection of object files each of which contains the code for a function or a group of related functions in the library. When you use a library function in your program, a copy of the object file that contains the function is incorporated in your executable at link time.

- A static shared library is a file containing object code that several processes may use simultaneously while executing. When a program is link edited with a static shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called *.lib* that identifies the library code is created in the object file. When the resulting *a.out* file is executed, it uses the information in this section to bring the required shared library code into the address space of the process. It should be noted that only one copy of the static shared library code is in memory and is shared by all processes using that library.
- A shared object, or dynamically linked library as in our case here, is a single object file that contains the code for every function in the library. When you call a library function in your program and specify a dynamic linking option (**-dy**) on the **cc** command line, the entire contents of the dynamically linked library are mapped into the virtual address space of your process at run time. As its name implies, a dynamically linked library contains code that can be used simultaneously by different programs at run time.

See "Link editing" (page 43) for more information on how statically and dynamically linked libraries are implemented, and details on how to combine the static and dynamic linking approaches in different ways according to your needs.

NOTE You can build dynamically linked libraries only for ELF binaries. See Chapter 14, "Shared libraries" (page 443) for more information on how to build a statically shared library in COFF file format.

Figure 2-1, "Organization of C Compilation System" (this page) shows the organization of the C compilation system. Details concerning the optimizer are omitted here because it is optional. See "Commonly used cc command line options" (page 40) for more information concerning invoking the optimizer.

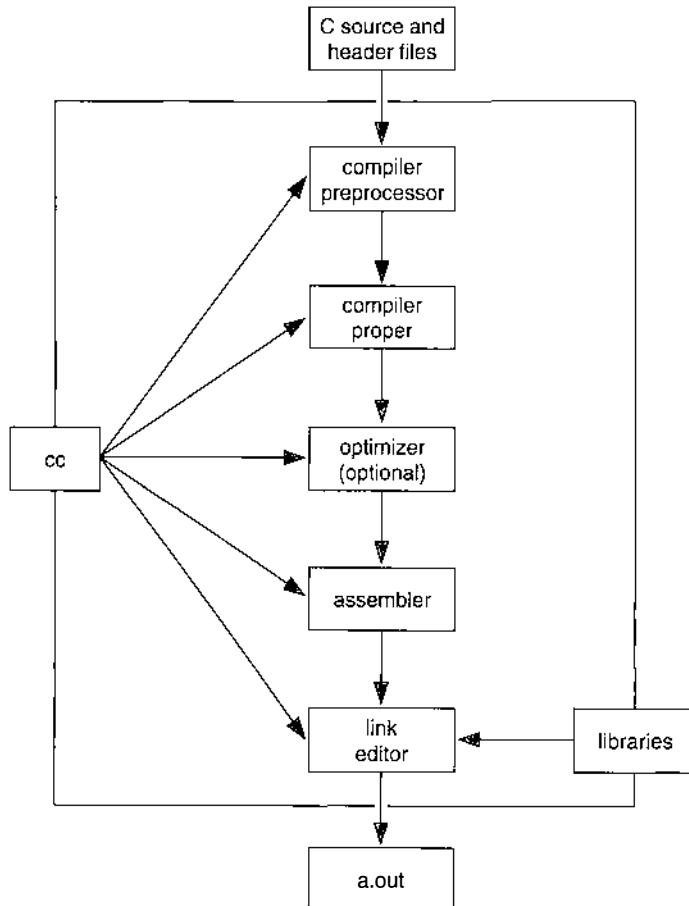


Figure 2-1 Organization of C Compilation System

Basic cc command line syntax

Now let's look at how this process works for a C language program to print the words `hello, world`. Here is the source code for the program, which we have written in the file `hello.c`:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

As indicated in Chapter 1, “Using programming tools” (page 9), the UNIX operating system command to create an executable program from C language source files is `cc`:

```
$ cc hello.c
```

The source files to be compiled must have names that end in the suffix `.c`.

Because there aren't any syntactic or semantic errors in the source code, the above command will create an executable program in the file `a.out` in the current directory:

```
$ ls -l
a.out
hello.c
```

Note that a `.o` file is not created when you compile a single source file.

Execute the program by entering its name after the system prompt:

```
$ a.out
hello, world
```

Because the name `a.out` is only of temporary usefulness, we'll rename the executable to `hello`:

```
$ mv a.out hello
```

You can also give the program the name `hello` when you compile it, with the `-o` option to the `cc` command:

```
$ cc -o hello hello.c
```

In either case, execute the program by entering its name after the system prompt:

```
$ hello
hello, world
```


Now let's look at how the `cc` command controls the four-step process described in the previous section. When you specify the `-P` option to `cc`, only the preprocessor component of the compiler is invoked:

```
$ cc -P hello.c
```

The preprocessor's output — the source code plus the preprocessed contents of *stdio.h* — is left in the file *hello.i* in our current directory:

```
$ ls -l
hello.c
hello.i
```

That output could be useful if, for example, you received a compiler error message for the undefined symbol "a" in the following fragment of source code:

```
if (i > 4)
{
    /* declaration follows
    int a; /* end of declaration */
    a = 4;
}
```

The unterminated comment on the third line will cause the compiler to treat the declaration that follows it as part of a comment. Because the preprocessor removes comments, its output:

```
if (i > 4)
{

    a = 4;
}
```

will clearly show the effect of the unterminated comment on the declaration. You can also use the preprocessed output to examine the results of conditional compilation and macro expansion.

If you specify the `-S` option to the `cc` command, only the preprocessor and compiler phases are invoked:

```
$ cc -S hello.c
```

The output — the assembly language code for the compiled source — is left in the file *hello.s* in our current directory. That output could be useful if you were writing an assembly language routine and wanted to see how the compiler went about a similar task.

If, finally, you specify the `-c` option to `cc`, all the components but the link editor are invoked:

```
$ cc -c hello.c
```

The output — the assembled object code for the program — is left in the object file `hello.o` in the current directory. You would typically want this output when using `make`(CP). See Chapter 10, “make” (page 315) for further information regarding `make`.

Enter the command:

```
$ cc hello.o
```

to create the executable object file `a.out`. By default, the link editor `ld`(CP) arranges for the standard C library function that was called in your program — `printf` — to be linked with the executable at link editing time. In the default arrangement described here, the standard C library is an archive library.

The outputs described above are, of course, inputs to the components of the compilation system. They are not the only inputs, however. The link editor, for example, supplies code that runs just before and just after your program to do startup and cleanup tasks. This code is automatically linked with your program only when the link editor is invoked through `cc`. Therefore, `cc hello.o` was specified in the previous example rather than `ld hello.o`. For similar reasons, you should invoke the assembler through `cc`(CP) rather than `as`(CP):

```
$ cc hello.s
```

The compilation process is largely identical if your program is in multiple source files. The only difference is that the default `cc` command line creates object files, as well as the executable object file `a.out`, in your current directory:

```
$ cc file1.c file2.c file3.c
$ ls -l
a.out
file1.c
file1.o
file2.c
file2.o
file3.c
file3.o
```

If one of your source files fails to compile, you need not recompile the others. If you receive a compiler error diagnostic for *file1.c* in the above command line, your current directory looks like this:

```
$ ls -l
file1.c
file2.c
file2.o
file3.c
file3.o
```

Compilation proceeds but linking is suppressed. Assuming you have fixed the error, the following command:

```
$ cc file1.c file2.o file3.o
```

creates the object file *file1.o* and links it with *file2.o* and *file3.o* to produce the executable program *a.out*. As the example suggests, C source files are compiled separately and independently. To create an executable program, the link editor **ld**(CP) must connect the definition of a symbol in one source file with external references to it in another.

Note that not all the **cc** command line options discussed are compiler options. The **-o** option is actually an **ld** option that is accepted by the **cc** command and passed to the link editor which creates the executable program. See the **cc**(CP) and **ld**(CP) manual pages for more information.

Commonly used cc command line options

This section details **cc** command line options that let you:

- specify the order in which directories are searched for included header files
- prepare your program for symbolic debugging or profiling
- optimize your program

Searching for a header file

The first line of our sample program was:

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

The format of that directive is the one you should use to include any of the standard header files that are supplied with the C compilation system. The angle brackets (< >) tell the preprocessor to search for the header file in the standard place for header files on your system, usually the */usr/include* directory.

The format is different for header files that you have stored in your own directories:

```
#include "header.h"
```

The double quotes ("") tell the preprocessor to search for *header.h* first in the directory of the file containing the `#include` line, which will usually be your current directory, then in the standard place.

If your header file is not in the current directory, specify the path of the directory in which it is stored with the `-I` option to `cc`. For instance, if you have included both *stdio.h* and *header.h* in the source file *test.c*:

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

and *header.h* is stored in the directory *./defs*, the command:

```
$ cc -I../defs test.c
```

will direct the preprocessor to search for *header.h* first in the current directory, then in the directory *./defs*, and finally in the standard place. It will also direct the preprocessor to search for *stdio.h* first in *./defs*, then in the standard place. The only difference is that the current directory is searched only for header files whose name you have enclosed in quotation marks.

You can specify the `-I` option more than once on the `cc` command line. The preprocessor searches the specified directories in the order they appear on the command line. You can therefore specify multiple options to `cc`(CP) on the same command line:

```
$ cc -o prog -I../defs test.c
```

Preparing your program for debugging

When you specify the `-g` option to `cc`:

```
$ cc -g test.c
```

you arrange for the compiler to generate information about program variables and statements that will be used by the debugger **dbxtra**(CP), or its Motif/X11 interface-oriented version, **dbXtra**(CP). The information supplied to **dbxtra** will allow you to use it to trace function calls, display the values of variables, and set breakpoints. See Chapter 6, “dbXtra and dbxtra” (page 193) for further information.

Preparing your program for profiling

To use either of the profilers that are supplied with the C compilation system, you must do two things:

1. Compile and link your program with a profiling option:

```
for prof: cc -qp test.c  
for lprof: cc -ql test.c
```

2. Run the profiled program:

```
$ a.out
```

At the end of execution, data about your program's run-time behavior is written to a file in your current directory:

```
for prof: mon.out  
for lprof: prog.cnt
```

where *prog* is the name of the profiled program. The data files are inputs to the profilers. See "lprof" (page 286) and "Interpreting profiling output" (page 290) in Chapter 9, "C programmer's productivity tools" for further information.

Optimizing your program

The **-O** option to **cc** invokes the optimizer:

```
$cc -O test.c
```

The optimizer improves the efficiency of the assembly language code generated by the compiler. That, in turn, speeds the execution time of your object code. Use the optimizer when you have finished debugging and profiling your program.

If you know the processor your code is normally run on, you can specify that **cc** should generate code that is optimized specifically for that processor (for example, **-Ki486**). For more information, see the **-K** option on the **cc**(CP) manual page.

Link editing

NOTE Because this section tries to cover the widest possible audience, it may provide more background than many users will need to link their programs with a C language library. If you are interested only in the how-to, and are comfortable with a purely formal presentation that scants motivation and background alike, you may want to skip to the quick-reference guide (page 61) in the last subsection.

Link editing refers to the process in which a symbol referenced in one module of your program is connected with its definition in another — more concretely, the process by which the symbol `printf()` in our sample source file *hello.c* is connected with its definition in the standard C library. Whichever link editing model you choose, static or dynamic, the link editor will search each module of your program, including any libraries you have used, for definitions of undefined external symbols in the other modules. If it does not find a definition for a symbol, the link editor will report an error by default, and fail to create an executable program. Multiply defined symbols are treated differently, however, under each approach. For details, see “Multiply defined symbols” (page 60). The principal difference between static and dynamic linking lies in what happens after this search is completed:

- Under static linking, external references in your program are connected with their definitions — assigned addresses in memory — when the executable is created. If the object code is in an archive library, copies of the archive library object files that satisfy still unresolved external references in your program are incorporated in your executable at link time; if the object code is in a static shared library, a special section called *.lib* that identifies the library code is created in the object file and the required shared library code is brought into the address space of the process only at the time the resulting *a.out* file is executed.
- Under dynamic linking, the contents of a dynamically linked library are mapped into the virtual address space of your process at run time. External references in your program are connected with their definitions when the program is executed.

This section examines the link editing process in detail. It starts with the default arrangement, and with the basics of linking your program with the standard libraries supplied by the C compilation system. It then details the implementation of the dynamic linking mechanism, and looks at some coding guidelines and maintenance tips for development of dynamically linked libraries. It also considers the reasons why you might prefer dynamic to static linking, for example:

- Dynamically linked programs save disk storage and system process memory by sharing library code at run time.

- Dynamically linked code can be fixed or enhanced without having to relink applications that depend on it.

Linking with archive libraries

An archive library is a collection of object files. Each function, or small group of related functions (typically, the related functions that you will sometimes find on the same manual page), is stored in its own object file. These object files are then collected in archives that are searched by the link editor, `ld`(CP), when you specify the necessary options on the `cc` command line. The link editor makes available to your program only the object files in these archives that contain a function you have called in your program.

The `cc` command line:

```
cc file1.c file2.c file3.c
```

creates object files corresponding to each of your source files, and links them with each other to create an executable program. These object files are called relocatable object files because they contain references to symbols that have not yet been connected with their definitions — have not yet been assigned addresses in memory.

Ideally, this command line would arrange for the standard C library functions that you have called in your program to be linked with your executable automatically. The standard C library is, in this default arrangement, an archive library called *libc.a*, which means that the functions you have called are linked with your program at link editing time. The standard C library contains system calls and C language functions. See “C library (libc)” (page 67) for details.

Under the default arrangement, the `cc` command creates and then links relocatable object files to generate an executable program. Copies of the object files in *libc.a* that resolve still unresolved external references in your program are incorporated in your executable at link time. If you are satisfied with this arrangement, you need make no other provision for link editing on the `cc` command line.

The next section, shows you how to dynamically link your program with the dynamically linked library version of *libc* to override the static linking default.

Linking with dynamically linked libraries

The `cc` command line:

```
cc -dy file1.c file2.c file3.c
```

creates object files corresponding to each of your source files, and links them with each other to create an executable program. The standard C library that is first searched for in this arrangement (**-dy**) is a dynamically linked library called *libc.so*, which means that the functions you have called are linked with your program at run time. (There are some exceptions. Many C library functions have been left out of *libc.so* by design. If you use one of these functions in your program, the code for the function is incorporated in your executable at link time. The function still automatically linked with your program, only statically rather than dynamically.)

NOTE The option **-dy** to `cc` is necessary to turn on dynamic linking. This option also instructs the compiler to produce object files in ELF format, the only object file format for which dynamic linking is supported.

Now consider the formal basis for this arrangement:

1. By convention, shared objects, or dynamically linked libraries, are designated by the prefix *lib* and the suffix *.so*; archives, or statically linked libraries, are designated by the prefix *lib* and the suffix *.a*; static shared libraries are designated by the prefix *lib* and the suffix *_s*. The dynamically linked library version of the standard C library is, then, *libc.so*; *libc.a* is the archive version; and *libc_s.a* is the static shared library version.
2. These conventions are recognized, in turn, by the **-l** option to the `cc` command.

```
cc -dy file1.c file2.c file3.c -lx -ly_s
```

directs the link editor to search the dynamically linked library *libx.so*, the archive library *libx.a*, or the static shared library *liby_s*. The `cc` command automatically passes **-lc** to the link editor.

3. Because of the **-dy** option, the link editor chooses the dynamically linked library implementation of a library, *libx.so*, in preference to the archive library implementation, *libx.a*, in the same directory.
4. By default, the link editor searches for libraries in the standard places on your system, */usr/ccs/lib* and */usr/lib*, in that order. The standard libraries supplied by the compilation system normally are kept in */usr/ccs/lib*.

Therefore, the `cc` command line directs the link editor to search */usr/ccs/lib/libc.so* rather than its archive library counterpart.

libc.so is one of the dynamically linked libraries supplied by SCO UNIX Development System. Other dynamically linked libraries are also supplied with the operating system, and usually are kept in the standard places. Of course, you can link your program with libraries that perform other tasks as well. Finally, you can create your own dynamically or statically shared libraries, and archive libraries.

Linking with standard libraries

libc is always searched automatically. *libc.a* is an archive library. It contains object files for every function in the standard C library. When you call a function in that library, only the object file in the archive that contains the called function is incorporated in your executable at linking time. Dynamically linked libraries are configured differently. *libc.so* is a single object file that contains the code for every function in the dynamically linked C library. When you call a function in that library, and dynamically link your program with it, the entire contents of *libc.so* are mapped into the virtual address space of your process at run time.

If you need to point the link editor to standard libraries that are not searched automatically, you specify the **-l** option explicitly on the **cc** command line. As shown previously, **-lx** directs the link editor to search the dynamically linked library *libx.so* or the archive library *libx.a*. So if your program calls the function **sin()**, for example, in the standard math library *libm*, the command

```
cc -dy file1.c file2.c file3.c -lm
```

will direct the link editor to search for */usr/ccs/lib/libm.so* and, if it does not find it, */usr/ccs/lib/libm.a* to satisfy references to **sin()** in your program. Because the compilation system does not supply a dynamically linked library version of *libm*, the above command will direct the link editor to search *libm.a* unless you have installed a dynamically linked library version of *libm* in the standard place. Note that because dynamic linking was turned on with the **-dy** option, the above command will direct the link editor to first search *libc.so* rather than *libc.a*. Use the same command without the **-dy** option to link your program statically with *libm.a* and *libc.a*. The contents of *libm* (page 72) are described in the second part of this chapter.

Note that because the link editor searches an archive library only to resolve undefined external references it has previously seen, the placement of the **-l** option on the **cc** command line is important. The command

```
cc file1.c -lm file2.c file3.c
```

will direct the link editor to search *libm.a* only for definitions that satisfy still unresolved external references in *file1.c*. As a rule, then, it's best to put **-l** at the end of the command line.

Creating and linking with archive and dynamically linked libraries

This section describes the basic mechanisms by which archives and dynamically linked libraries are built. It will give you some sense of where these libraries come from, as a basis for understanding how they are implemented and linked with your programs. If you are developing a library, you need to know the material in this section.

The following commands:

```
cc -c function1.c function2.c function3.c  
ar -r libfoo.a function1.o function2.o function3.o
```

creates an archive library, *libfoo.a*, that consists of the named object files. See **ar**(CP) for details of usage. When you use the **-l** option to link your program with *libfoo.a*

```
cc -Ldir file1.c file2.c file3.c -lfoo
```

the link editor incorporatess in your executable only the object files in this archive that contain a function you have called in your program.

Create a dynamically linked library by specifying the **-dy -G** option:

```
cc -dy -G -o libfoo.so function1.o function2.o function3.o
```

This command creates the dynamically linked library *libfoo.so* consisting of the object code for the functions contained in the named files. See “Implementation” (page 53) for details of compiler option **-K PIC**.

When you use the **-l** option to link your program with *libfoo.so*

```
cc -dy -Ldir file1.c file2.c file3.c -lfoo
```

the link editor records in your executable the name of the dynamically linked library and a small amount of bookkeeping information for use by the system at run time. Another component of the system — the dynamic linker — does the actual linking. Note, again, that because dynamic linking was turned on with the **-dy** option, the above command directs the link editor to search *libc.so* as well as *libfoo.a*.

NOTE Because code in a dynamically linked library is not copied into your executable object file at link time, a dynamically linked executable normally uses less disk space than a statically linked executable. For the same reason, code in a dynamically linked library can be changed without breaking executables that depend on it. Even if the dynamically linked C library were enhanced in the future, you would not have to relink programs that depended on it as long as the enhancements were compatible with your code. See “Checking for run-time compatibility” (page 52) for more information.

The dynamic linker would simply use the definitions in the new version of the library to resolve external references in your executables at run time.

You can specify the name of the dynamically linked library that you want to create under the **-G** option. The following command, for example, will create a dynamically linked library called *a.out* by default:

```
cc -dy -G function1.o function2.o function3.o
```

You can then rename the dynamically linked library to *libfoo.so*:

```
mv a.out libfoo.so
```

As noted, you use the *lib* prefix and the *.so* suffix because they are conventions recognized by **-l**, just as are *lib* and *.a* for archive libraries. So while it is legitimate to create a dynamically linked library that does not follow the naming convention, and to link it with your program

```
cc -dy -G -o sharedob function1.o function2.o function3.o  
cc -dy file1.c file2.c file3.c path/sharedob
```

we recommend against it. Not only will you have to enter a pathname on the **cc** command line every time you use **sharedob** in a program, that pathname will be hard-coded in your executables.

The command line:

```
cc -Ldir file1.c file2.c file3.c -lfoo
```

directs the link editor to record in your executable the filename of the dynamically linked library with which it is to be linked at run time.

NOTE **cc** links the *filename*, the last component of a pathname, of the dynamically linked library, not its pathname.

When you use the **-l** option to link your program with a dynamically linked library, not only must the link editor be told which directory to search for that library, so must the dynamic linker (unless the directory is the standard place, which the dynamic linker searches by default). See “Specifying directories to be searched by the dynamic linker” (page 51) for more information about pointing to the dynamic linker. However, as long as the pathname of a dynamically linked library is not hard-coded in your executable, you can move the dynamically linked library to a different directory without breaking your program. You should avoid using pathnames of dynamically linked libraries on the **cc** command line. Those pathnames will be hard-coded in your executable. They won’t be if you use **-l**.

Finally, the `cc -G` command not only creates a dynamically linked library, it accepts a dynamically linked library or archive library as input. When you create *libfoo.so*, you can link it with a library you have already created, such as *libsharedob.so*:

```
cc -dy -G -o libfoo.so -Ldir function1.o \
    function2.o function3.o -lsharedob
```

This command arranges for *libsharedob.so* to be linked with *libfoo.so* when, at run time, *libfoo.so* is linked with your program. Note that here you will have to point the dynamic linker to the directories in which both *libfoo.so* and *libsharedob.so* are stored.

Specifying directories to be searched by the link editor

In the previous section you created the archive library *libfoo.a* and the dynamically linked library *libfoo.so*. For this example, both of these libraries are stored in the directory */home/mylibs*, and the executable is being created in a different directory. This example reflects the way most programmers organize their work on the UNIX operating system.

In order to link your program with either of these libraries, the link editor must access the */home/mylibs* directory. Specify the directory's pathname with the `-L` option:

```
cc -dy -L/home/mylibs file1.c file2.c file3.c -lfoo
```

The `-L` option directs the link editor to search for the libraries named with `-l` first in the specified directory, then in the standard places. In this example, having found the directory */home/mylibs*, the link editor will search *libfoo.so* rather than *libfoo.a*. As shown earlier, when the link editor encounters otherwise identically named dynamically linked library and archive libraries in the same directory, it searches the library with the *.so* suffix first because of the `-dy` option. For the same reason, it will search *libc.so* here rather than *libc.a*. Note that you must specify `-L` to direct the link editor to search for libraries in your current directory. You can use a period (.) to represent the current directory.

To direct the link editor to search *libfoo.a*, you can either rely on the compiler defaults or explicitly turn off dynamic linking:

```
cc -dn -L/home/mylibs file1.c file2.c file3.c -lfoo
```

Under `-dn`, the link editor does not accept dynamically linked libraries as input. Here, then, it searches *libfoo.a* rather than *libfoo.so*, and *libc.a* rather than *libc.so*.

To link your program statically with *libfoo.a* and dynamically with *libc.so*, you can do either of two things. First, you can move *libfoo.a* to a different directory — */home/archives*, for example — then specify */home/archives* with the `-L` option:

```
cc -dy -L/home/archives -L/home/mylibs file1.c file2.c file3.c -lfoo
```

As long as the link editor encounters the */home/archives* directory before it encounters the */home/mylibs* directory, it searches *libfoo.a* rather than *libfoo.so*. When otherwise identically named *.so* and *.a* libraries exist in different directories, the link editor searches the first one it finds. The same thing is true, by the way, for identically named libraries of either type. If you have different versions of *libfoo.a* in your directories, the link editor searches the first one it finds.

A better alternative might be to leave *libfoo.a* where you had it in the first place and use the **-Bstatic** and **-Bdynamic** options to turn dynamic linking off and on. The following command links your program statically with *libfoo.a* and dynamically with *libc.so*:

```
cc -dy -L/home/mylibs file1.c file2.c file3.c -Bstatic -lfoo -Bdynamic
```

When you specify **-Bstatic**, the link editor will not accept a dynamically linked library as input until you specify **-Bdynamic**. You can use these options as toggles as often as needed on the **cc** command line:

```
cc -dy -L/home/mylibs file1.c file2.c -Bstatic -lfoo \
    file3.c -Bdynamic -lsharedob
```

This command will direct the link editor to search the following libraries:

1. *libfoo.a* to resolve still unresolved external references in *file1.c* and *file2.c*
2. *libsharedob.so* to resolve still unresolved external references in all three files and in *libfoo.a*
3. *libc.so* to resolve still unresolved external references in all three files and the preceding libraries
4. *libc.a* to resolve still unresolved external references in all three files, the preceding libraries, and *libc.so*

Files, including libraries, are searched for definitions in the order they are listed on the **cc** command line. The standard C library is always searched last.

You can add to the list of directories to be searched by the link editor by using the environment variable **LD_LIBRARY_PATH**. **LD_LIBRARY_PATH** must be a list of colon-separated directory names. An optional second list is separated from the first by a semicolon:

```
LD_LIBRARY_PATH=dir:dir;dir:dir export LD_LIBRARY_PATH
```

The directories specified before the semicolon are searched, in order, before the directories specified with **-L**; the directories specified after the semicolon are searched, in order, after the directories specified with **-L**. Note that you can use **LD_LIBRARY_PATH** in place of **-L** altogether. In that case the link editor searches for libraries named with **-l** first in the directories specified before the semicolon, next in the directories specified after the semicolon, and last in the standard places. You should use absolute pathnames when you set this environment variable.

NOTE `LD_LIBRARY_PATH` is also used by the dynamic linker. If `LD_LIBRARY_PATH` exists in your environment, the dynamic linker searches the directories named in it for dynamically linked libraries to be linked with your program at execution. In using `LD_LIBRARY_PATH` with the link editor or the dynamic linker, then, you should keep in mind that any directories you give to one you are also giving to the other.

Specifying directories to be searched by the dynamic linker

When you use the `-l` option, you must point the dynamic linker to the directories of the dynamically linked libraries that are to be linked with your program at execution. The environment variable `LD_RUN_PATH` lets you do that at link time. To set `LD_RUN_PATH`, list the absolute pathnames of the directories you want searched in the order you want them searched. Separate pathnames with a colon, as shown in the following example:

```
LD_RUN_PATH=/home/mylibs export LD_RUN_PATH
```

The command:

```
cc -dy -o prog -L/home/mylibs file1.c file2.c file3.c -lfoo
```

directs the dynamic linker to search for `libfoo.so` in `/home/mylibs` when you execute your program:

```
prog
```

The dynamic linker searches the standard place by default, after the directories you have assigned to `LD_RUN_PATH`. Note that as far as the dynamic linker is concerned, the standard place for libraries is `/usr/lib`. Any executable versions of libraries supplied by the compilation system kept in `/usr/lib`.

The environment variable `LD_LIBRARY_PATH` lets you do the same thing at run time. Suppose you have moved `libfoo.so` to `/home/sharedobs`. It is too late to replace `/home/mylibs` with `/home/sharedobs` in `LD_RUN_PATH`, at least without link editing your program again. You can, however, assign the new directory to `LD_LIBRARY_PATH`, as follows:

```
LD_LIBRARY_PATH=/home/sharedobs export LD_LIBRARY_PATH
```

Now when you execute your program

```
prog
```

the dynamic linker searches for `libfoo.so` first in `/home/mylibs` and, not finding it there, in `/home/sharedobs`. The directory assigned to `LD_RUN_PATH` is searched before the directory assigned to `LD_LIBRARY_PATH`. The important point is that because the pathname of `libfoo.so` is not hard-coded in *prog*, you can direct the dynamic linker to search a different directory when you execute your program. You can move a dynamically linked library without breaking your application.

You can set `LD_LIBRARY_PATH` without first having set `LD_RUN_PATH`. The main difference between them is that once you have used `LD_RUN_PATH` for an application, the dynamic linker searches the specified directories every time the application is executed (unless you have relinked the application in a different environment). In contrast, you can assign different directories to `LD_LIBRARY_PATH` each time you execute the application. `LD_LIBRARY_PATH` directs the dynamic linker to search the assigned directories before it searches the standard place. Directories, including those in the optional second list, are searched in the order listed. See the previous section for the syntax.

Note, finally, that when linking a set-user or set-group program, the dynamic linker ignores any directories that are not built into the dynamic linker. Currently, the only built-in directory is `/usr/lib`.

Checking for run-time compatibility

Suppose you have been supplied with an updated version of a dynamically linked library. You have already compiled your program with the previous version; the link editor has checked it for undefined symbols, found none, and created an executable. Therefore, you should not have to link your program again. The dynamic linker will simply use the definitions in the new version of the dynamically linked library to satisfy unresolved external references in the executable.

Suppose further that this is a database update program that takes several days to run. You want to be sure that your program does not fail in a critical section because a symbol that was defined by the previous version of the dynamically linked library is no longer defined by the new version. You want the information that the link editor gives you — that your executable is compatible with the dynamically linked library — without having to link edit it again.

There are two ways you can check for run-time compatibility. The command `ldd` (“list dynamic dependencies”) directs the dynamic linker to print the pathnames of the dynamically linked library on which your program depends:

`ldd prog`

When you specify the `-d` option to `ldd`, the dynamic linker prints a diagnostic message for each unresolved data reference it would encounter if *prog* were executed. When you specify the `-r` option, it prints a diagnostic message for each unresolved data or function reference it would encounter if *prog* were executed.

You can do the same thing when you execute your program. Whereas the dynamic linker resolves data references immediately at run time, it normally delays resolving function references until a function is invoked for the first time. Normally, then, the lack of a definition for a function is not apparent

until the function is invoked. By setting the environment variable `LD_BIND_NOW`

```
LD_BIND_NOW=1 export LD_BIND_NOW
```

before you execute your program, you direct the dynamic linker to resolve all references immediately. In that way, you can learn before execution of `main()` begins that the functions invoked by your process actually are defined.

Dynamic linking programming interface

You can use a programming interface to the dynamic linking mechanism to attach a dynamically linked library to the address space of your process during execution, look up the address of a function in the library, call that function, and then detach the library when it is no longer needed. See, for example, `dlsym(S)` for more information.

Implementation

The following lists the basic implementation of the static and dynamic linking mechanisms:

- When you use an archive library function, a copy of the object file that contains the function is incorporated in your executable at link time. External references to the function are assigned virtual addresses when the executable is created.
- When you use a function in a dynamically linked library, the entire contents of the library are mapped into the virtual address space of your process at run time. External references to the function are assigned virtual addresses when you execute the program. The link editor records in your executable only the filename of the dynamically linked library and a small amount of bookkeeping information for use by the dynamic linker at run time.

There are one or two cases in which you might not want to use dynamic linking. Because code in a dynamically linked library is not copied into your executable object file at link time, a dynamically linked executable normally uses less disk space than a statically linked executable. If your program calls only a few small library functions, however, the bookkeeping information to be used by the dynamic linker may take up more space in your executable than the code for those functions. You can use the `size` command to determine the difference. See `size(CP)` for more information.

In a similar way, using a dynamically linked library may occasionally add to the memory requirements of a process. Although the text segment of a dynamically linked library is shared by all processes that use it, its writable data typically is not. See “Guidelines for building dynamically linked libraries” (page 54) for details. Every process that uses a dynamically linked

library usually gets a private copy of its entire data segment, regardless of how much of the data is needed. If an application uses only a small portion of text and data segments of a dynamically linked library, executing the application might require more memory with a dynamically linked library than without one. For example, it would waste memory to use the dynamically linked library version of standard C library to access only `strcmp()`. Although sharing `strcmp()` saves space on your disk and memory on the system, the memory cost to your process of having a private copy of the C library's data segment would make the archive version of `strcmp()` the more appropriate choice.

Now consider dynamic linking in more detail. First, each process that uses a dynamically linked library references a single copy of its code in memory. That means that when other users on your system call a function in a dynamically linked library, the entire contents of that library are mapped into the virtual address space of their processes as well. If they have called the same function as you, external references to the function in their programs will, in all likelihood, be assigned different virtual addresses. Because the function may be loaded at a different virtual address for each process that uses it, the system cannot calculate absolute addresses in memory until run time.

Secondly, the memory management scheme underlying dynamic linking shares memory among processes at the granularity of a page. Memory pages can be shared as long as they are not modified at run time. If a process writes to a shared page while relocating a reference to a dynamically linked library, it gets a private copy of that page and loses the benefits of code sharing without affecting other users of the page.

Thirdly, to create programs that require the least possible amount of page modification at run time, the compiler generates position-independent code under the **-K PIC** option. Whereas executable code normally must be tied to a fixed address in memory, position-independent code can be loaded anywhere in the address space of a process. Because the code is not tied to specific addresses, it will execute correctly — without page modification — at a different address in each process that uses it. As we have indicated, you should specify **-K PIC** when you create a dynamically linked library:

```
cc -dy -K PIC -G -o libfoo.so function1.c function2.c function3.c
```

Relocatable references in your object code are moved from its text segment to tables in the data segment. See Chapter 8, “ELF object files” in *Developer's Topics* for details.

Guidelines for building dynamically linked libraries

This section gives coding guidelines and maintenance tips for development of dynamically linked libraries. Before getting down to specifics, we should emphasize that if you plan to develop a commercial dynamic shared library,

you ought to consider providing a compatible archive as well. As noted previously, some users may not find a dynamically linked library appropriate for their applications. Others may want their applications to run on UNIX operating system releases without shared library support. Dynamically linked (or shared) library code is completely compatible with archive library code. You can use the same source files to build archive and dynamically linked library versions of a library. You can use dynamically linked libraries for the following reasons:

- Because library code is not copied into the executables that use it, they require less disk space.
- Because library code is shared at run time, the dynamic memory needs of systems are reduced.
- Because symbol resolution is put off until run time, dynamically linked libraries can be updated without having to relink applications that depend on them.
- As long as its pathname is not hard-coded in an executable, a dynamically linked library can be moved to a different directory without breaking an application.

To enhance performance of a dynamically linked library, you should:

1. Minimize the library's data segment (this page)
2. Minimize paging activity (page 58)

Minimize the library's data segment

As noted, only the text segment of a dynamically linked library is shared by all processes that use it; its data segment typically is not. Every process that uses a dynamically linked library usually gets a private memory copy of its entire data segment, regardless of how much data is needed. You can cut down the size of the data segment in several ways:

- Try to use automatic (stack) variables
- Don't use permanent storage if automatic variables will work
- Use functional interfaces rather than global variables

Generally speaking, these make library interfaces and code easier to maintain. Moreover, defining functional interfaces often eliminates global variables entirely, which in turn eliminates global copy of data. The following illustrates the differences between the function interface approach (**strerror(S)**) and the global variable approach (**sys_errlist[X]**).

In some other implementations, system error messages are made available to applications only through two global variables:

```
extern int      sys_nerr;
extern char    *sys_errlist[];
```

sys_errlist[X] gives a character string for the error **X**, if **X** is a non-negative value less than **sys_nerr**. Now if the current list of messages were made available to applications only through a lookup table in an archive library, applications that used the table obviously would not be able to access new messages as they were added to the system unless they were relinked with the library. Errors might occur for which these applications could not produce useful diagnostics. Something similar happens when you use a global lookup table in a dynamically linked library:

1. The compilation system sets aside memory for the table in the address space of each executable that uses it, even though it does not know yet where the table will be loaded.
2. After the table is loaded, the dynamic linker copies it into the space that has been set aside.
3. Each process that uses the table gets a private copy of the library's data segment, including the table, and an additional copy of the table in its own data segment.
4. Each process pays a performance penalty for the overhead of copying the table at run time.
5. Because the space for the table is allocated when the executable is built, the application will not have enough room to hold any new messages you might want to add in the future. A functional interface overcomes these difficulties.

strerror() might be implemented as follows:

```
static const char *msg[] = {
    "Error 0",
    "Not owner",
    "No such file or directory",
    ...
};

char *
strerror(int err)
{
    if (err < 0 || err >= sizeof(msg)/sizeof(msg[0]))
        return 0;
    return (char *)msg[err];
}
```

The message array is static, so no application space is allocated to hold a separate copy. Because no application copy exists, the dynamic linker does not waste time moving the table. New messages can be added, because only the library knows how many messages exist. Finally, note the use of the type qualifier **const** to identify data as read-only. Whereas writable data is stored in the data segment of a dynamically linked library, read-only data can be stored in its text segment. For more information on **const**, see “Type qualifiers” (page 99) in Chapter 3, “C language compiler”, and the **-Krodata** option to **cc**(CP).

In a similar way, try to allocate buffers dynamically — at run time — instead of defining them at link time. This saves memory because only the processes that need the buffers get them. It also allows the size of the buffers to change from one release of the library to the next without affecting compatibility as shown in the following:

```
char *
buffer()
{
    static char *buf = 0;

    if (buf == 0)
    {
        if ((buf = malloc(BUFSIZE)) == 0)
            return 0;
    }
    ...
    return buf;
}
```

- Exclude functions that use large amounts of global data if you cannot rewrite them in the ways described previously. If an infrequently used routine defines large amounts of static data, it probably does not belong in a dynamically linked library.
- Make the library self-contained. If a dynamically linked library imports definitions from another dynamically linked library, each process that uses it gets a private copy not only of its data segment, but of the data segment of the dynamically linked library from which the definitions were imported. In cases of conflict, this guideline should probably take precedence over the preceding one.

Minimize paging activity

Although processes that use dynamically linked libraries do not write to shared pages, they still may incur page faults. To the extent they do, their performance degrades. You can minimize paging activity in the following ways:

- Organize to improve locality of reference.
 - a) Exclude infrequently used routines on which the library itself does not depend. Traditional *a.out* files contain all the code they need at run time. So if a process calls a function, it may already be in memory because of its proximity to other text in the process. If the function is in a dynamically linked library, however, the surrounding library code may be unrelated to the calling process. Only rarely, for example, does any single executable use everything in the dynamically linked C library. If a dynamically linked library has unrelated functions, and if unrelated processes make random calls to those functions, locality of reference may be decreased, leading to more paging activity. The point is that functions used by only a few *a.out* files do not save much disk space by being in a dynamically linked library, and can degrade performance.
 - b) Try to improve locality of reference by grouping dynamically related functions. If every call to **funcA()** generates calls to **funcB()** and **funcC()**, try to put them in the same page. **cflow(CP)** generates this type of static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.
- Align for paging. Try to arrange object files in the dynamically linked library so that frequently used functions do not unnecessarily cross page boundaries. First, determine where the page boundaries fall. You can use the **nm(CP)** command to determine how symbol values relate to page boundaries. After grouping related functions, break them up into page-sized chunks. Although some object files and functions are larger than a page, many are not. Then use the less frequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault is decreased. You can put frequently used, unrelated functions together because they will probably be called randomly enough to keep the pages in memory.
- Avoid hardware thrashing.

Maintenance

Allocating buffers dynamically can ease the job of library maintenance. As a general rule, ensure that updated versions of a dynamically linked library are compatible with its previous versions so that users do not have to recompile their applications.

NOTE Avoid changing the names of library symbols from one release to the next.

However, there may be instances in which you need to release a library version that is incompatible with its predecessor. On the one hand, you will want to maintain the older version for dynamically linked executables that depend on it. On the other hand, you will want newly created executables to be linked with the updated version. Moreover, you will probably want both versions to be stored in the same directory. In this example, you could give the new release a different name, rewrite your documentation, and so forth. A better alternative would be to plan for the contingency in the first instance by using the following sequence of commands when you create the original version of the dynamically linked library:

```
cc -dy -K PIC -G -h libfoo.1 -o libfoo.1 function1.c \
                                     function2.c function3.c
```

```
ln libfoo.1 libfoo.so
```

In the first command **-h** stores the filename given to it, *libfoo.1*, in the dynamically linked library itself. You then use the UNIX operating system command **ln(C)** to create a link between the name *libfoo.1* and the name *libfoo.so*. *libfoo.so* is the name the link editor, **ld(CP)**, looks for when users of your library specify:

```
cc -dy -Ldir file1.c file2.c file3.c -lfoo
```

However, the link editor records in the user's executable the name you gave to **-h**, *libfoo.1*, rather than the name *libfoo.so*. That means that when you release a subsequent, incompatible version of the library, *libfoo.2*, executables that depend on *libfoo.1* continues to be linked with it at run time.

As shown earlier, the dynamic linker uses the filename of the dynamically linked library that is stored in the executable to satisfy unresolved external references at run time.

You use the same sequence of commands when you create *libfoo.2*:

```
cc -dy -K PIC -G -h libfoo.2 -o libfoo.2 function1.c \  
function2.c function4.c
```

In *libfoo.2* *libfoo.so*

When users specify:

```
cc -dy -L dir file1.c file2.c file3.c -lfoo
```

the filename *libfoo.2* is stored in their executables, and their programs are linked with the new library version at run time.

Multiply defined symbols

Multiply defined symbols — except for different-sized initialized data objects — are not reported as errors under dynamic linking. The link editor will not report an error for multiple definitions of a function or a same-sized data object when each such definition resides within a different dynamically linked library or within a dynamically linked executable and different dynamically linked libraries. The dynamic linker will use the definition in whichever object occurs first on the `cc` command line. You can, however, specify **-Bsymbolic** when you create a dynamically linked library:

```
cc -dy -K PIC -G -Bsymbolic -o libfoo.so function1.c \  
function2.c function3.c
```

to insure that the dynamic linker uses the dynamically linked library's definition of one of its own symbols, rather than a definition of the same symbol in an executable or another library.

In contrast, multiply defined symbols are generally reported as errors under static linking, because definitions of so-called weak symbols can be hidden from the link editor by a definition of a global symbol. If a defined global symbol exists, the appearance of a weak symbol with the same name does not cause an error.

To illustrate this, our own implementation of the standard C library. This library provides services that users are allowed to redefine and replace. At the same time, however, ANSI C defines standard services that must be present on the system and cannot be replaced in a strictly conforming program. For example, **fread(S)** is an ANSI C library function; the system function **read(S)** is not. So a conforming program may redefine **read()** and still use **fread()** in a predictable way.

The problem with this is that **read()** underlies the **fread()** implementation in the standard C library. A program that redefines **read()** could “confuse” the **fread()** implementation. To guard against this, ANSI C states that an implementation cannot use a name that is not reserved to it. Therefore **_read()** — note the leading underscore — is used to implement **fread()** in the standard C library.

Now suppose that a program you have written calls `read()`. If your program is going to work, the definition for `read()` has to exist in the C library. It is identical to the definition for `_read()` and is contained in the same object file.

Suppose further that another program you have written redefines `read()`, and that this same program calls `fread()`. Because you get our definitions of both `_read()` and `read()` when you use `fread()`, we would expect the link editor to report the multiply defined symbol `read()` as an error, and fail to create an executable program. To prevent that, use the `#pragma` directive in your source code for the library as follows:

```
#pragma weak read = _read
```

Because our `read()` is defined as a weak symbol, your own definition of `read()` overrides the definition in the standard C library. You can use the `#pragma` directive in the same way in your own library code.

There is a second use for weak symbols:

```
#pragma weak read
```

tells the link editor not to complain if it does not find a definition for the weak symbol `read`. References to the symbol use the symbol value if defined, 0 otherwise. The link editor does not extract archive members to resolve undefined weak symbols. The mechanism is intended to be used primarily with functions. Although it will work for most data objects, it should not be used with uninitialized global data (“common” symbols) or with data objects in dynamically linked libraries that are exported to executables.

Quick-reference guide

1. By convention, dynamically linked libraries, or shared objects, are designated by the prefix *lib* and the suffix *.so*; archives, or statically linked libraries, are designated by the prefix *lib* and the suffix *.a*. The archive version of the standard C library is *libc.a*; *libc.so* is the dynamically linked library version.
2. These conventions are recognized, in turn, by the `-l` option to the `cc` command. `-lx` directs the link editor to search the archive library *libx.a* or the dynamically linked library *libx.so*. The `cc` command automatically passes `-lc` to the link editor. Therefore, the compilation system arranges for the standard C library to be linked with your program transparently.
3. By default, the link editor chooses the archive library implementation, *libx.a*. Use option `-y` to direct the link editor to choose the dynamically linked library implementation of a library, *libx.so*, in preference to *libx.a* in the same directory.
4. By default, the link editor searches for libraries in the standard places on your system, */usr/ccs/lib* and */usr/lib*, in that order. The standard libraries supplied by the compilation system normally are kept in */usr/ccs/lib*.

In this arrangement, then, C programs are statically linked with *libc.a* automatically:

```
cc file1.c file2.c file3.c
```

To link your program dynamically with *libc.so*, turn on dynamic linking with the **-dy** option:

```
cc -dy file1.c file2.c file3.c
```

Specify the **-l** option explicitly to link your program with any other library. If the library is in the standard place, the command:

```
cc file1.c file2.c file3.c -lx
```

directs the link editor to search for *libx.a* in the standard place. With the **-dy** option, the command:

```
cc -dy file1.c file2.c file3.c -lx
```

directs the link editor to search for *libx.so*, then *libx.a* in the standard place. Note that the compilation system supplies dynamically linked library version of *libc*. (Other dynamically linked libraries are supplied with the operating system, and usually are kept in the standard places.) Note, too, that, as a rule, it's best to place **-l** at the end of the command line.

If the library is not in the standard place, specify the path of the directory in which it is stored with the **-L** option:

```
cc -L dir file1.c file2.c file3.c -lx
```

or the environment variable **LD_LIBRARY_PATH**:

```
LD_LIBRARY_PATH=dir export LD_LIBRARY_PATH  
cc file1.c file2.c file3.c -lx
```

If the library is a dynamically linked library and is not in the standard place, you must also specify the path of the directory in which it is stored with either the environment variable **LD_RUN_PATH** at link time, or the environment variable **LD_LIBRARY_PATH** at run time:

```
LD_RUN_PATH=dir export LD_RUN_PATH  
$ LD_LIBRARY_PATH=dir export LD_LIBRARY_PATH
```

Use an absolute path when you set these environment variables. Note that **LD_LIBRARY_PATH** is read both at link time and at run time.

By default, the link editor ignores *libx.so*. To direct the link editor to search *libx.so* and then *libx.a* in the same directory, turn on dynamic linking with the **-dy** option:

```
cc -dy -Ldir file1.c file2.c file3.c -lx
```

That command directs the link editor to search *libc.so* well as *libx.so*. To link your program statically with *libx.a* and dynamically with *libc.so*, use the **-Bstatic** and **-Bdynamic** options to turn dynamic linking off and on:

```
cc -dy -Ldir file1.c file2.c file3.c -Bstatic -lx -Bdynamic
```

Files, including libraries, are searched for definitions in the order they are listed on the `cc` command line. The standard C library is always searched last.

Libraries and header files

The standard libraries supplied by the C compilation system contain functions that you can use in your program to perform input/output, string handling, and other high-level operations that are not explicitly provided by the C language. Header files contain definitions and declarations that your program will need if it calls a library function. They also contain function-like macros that you can use in your program as you would a function.

The first section of this chapter shows you how to link your program with these standard libraries and how to include a header file. This section illustrates the use of header files and library functions in your program. It also describes the contents and location of some of the more important standard libraries (*libc*, *libm*, and *libgen*), along with a brief discussion of standard I/O.

Header files

Header files serve as the interface between your program and the libraries supplied by the C compilation system. Because the functions that perform standard I/O often use the same definitions and declarations, the system supplies a common interface to the functions in the header file *stdio.h*. If you have definitions or declarations that you want to make available to several source files, you can create a header file with any editor, store it in a convenient directory, and include it in your program as described in the first part of this chapter.

Header files traditionally are designated by the suffix *.h*, and are brought into a program at compile time. The preprocessor component of the compiler interprets the **#include** statement in your program as a directive. The two most commonly used directives are **#include** and **#define**. The **#include** directive is used to call in and process the contents of the named file. The **#define** directive is used to define the replacement token string for an identifier. For example:

```
#define NULL 0
```

defines the macro `NULL` to have the replacement token sequence `"0"`. See Chapter 3, "C language compiler" (page 83) for the complete list of preprocessing directives. The most commonly used *.h* files are listed in Table 2-1,

“Header files” (page 64) to illustrate the range of tasks you can perform with header files and library functions. When you use a library function in your program, the manual page tells you which header file, if any, needs to be included. If a header file is mentioned, it should be included before you use any of the associated functions or declarations in your program. Put the **#include** right at the top of a source file.

Table 2-1 Header files

<code>assert.h</code>	assertion checking
<code>ctype.h</code>	character handling
<code>errno.h</code>	error conditions
<code>float.h</code>	floating point limits
<code>limits.h</code>	other data type limits
<code>locale.h</code>	program’s locale
<code>math.h</code>	mathematics
<code>setjmp.h</code>	nonlocal jumps
<code>signal.h</code>	signal handling
<code>stdarg.h</code>	variable arguments
<code>stddef.h</code>	common definitions
<code>stdio.h</code>	standard input/output
<code>stdlib.h</code>	general utilities
<code>string.h</code>	string handling
<code>time.h</code>	date and time
<code>unistd.h</code>	system calls

How to use library functions

The manual page for each function describes how you should use the function in your program. As an example, consider the **strcmp()** routine, which compares character strings. See **string(S)** for more information. Related functions are described there as well, but only the sections relevant to **strcmp()** are shown in Example 2-1, “Excerpt from string(S) manual page” (page 65).

Example 2-1 Excerpt from string(S) manual page

Name

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok - string operations.

Syntax

```
cc . . . -lc

#include <string.h>

...

int strcmp (s1, s2)
char *s1, *s2;

...
```

Description

```
...
```

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as s1 is lexicographically less than, equal to, or greater than s2. strncmp makes the same comparison but looks at most n characters.

```
...
```

The Description section tells you what the function or macro does. The Syntax section contains the critical information about how you use the function or macro in your program.

Note that the line in the Syntax section:

```
#include <string.h>
```

means that you should include the header file *string.h* in your program because it contains useful definitions or declarations relating to **strcmp**. *string.h* contains the line:

```
extern int strcmp(const char *, const char *);
```

that describes the kinds of arguments expected and returned by **strcmp**(. This line is called a function prototype. Function prototypes afford a greater degree of argument type checking than old-style function declarations, so you lessen your chance of using the function incorrectly. By including *string.h*, you assure that the compiler checks calls to **strcmp**(. against the official

interface. You can, of course, examine *string.h* in the standard place for header files on your system, usually the */usr/include* directory.

The next thing in the Syntax section is the formal declaration of the function. The formal declaration tells you:

- the type of value returned by the function
- the arguments the function expects to receive when called, if any
- the argument types

By way of illustration, consider how you might use **strcmp()** in your own code. Example 2-2, “How strcmp() is used in a program” (this page) shows a program fragment that will find the bird of your choice in an array of birds.

Example 2-2 How strcmp() is used in a program

```
#include <string.h>

/* birds must be in alphabetical order */
char *birds[] = { "albatross", "canary", "cardinal", "ostrich", "penguin" };

/* Return the index of the bird in the array. */
/* If the bird is not in the array, return -1 */

int is_bird(const char *string)
{
    int low, high, midpoint;
    int cmp_value;

    /* use a binary search to find the bird */
    low = 0;
    high = sizeof(birds)/sizeof(char *) - 1;
    while(low <= high)
    {
        midpoint = (low + high)/2;
        cmp_value = strcmp(string, birds[midpoint]);
        if (cmp_value < 0)
            high = midpoint - 1;
        else if (cmp_value > 0)
            low = midpoint + 1;
        else /* found a match */
            return midpoint;
    }
    return -1;
}
```

C library (libc)

As indicated in the first part of this chapter, *libc* contains system calls and C language functions, described in the S section of the manual pages. This section contains some of the more important routines in the standard C library. Discussed here are:

- Standard I/O routines
- String, character, environment and memory routines
- System calls

Standard I/O routines

Frequently, one manual page describes several related functions or macros. In Table 2-2, “Standard I/O functions and macros” (page 68) the left column contains the name that appears at the top of the manual page; the other names in the same row are related functions or macros described on the same manual page. Programs that use these routines should include the header file *stdio.h*.

Table 2-2 Standard I/O functions and macros

<code>fclose</code>	<code>fflush</code>			Close or flush a stream.
<code>ferror</code>	<code>feof</code>	<code>clearerr</code>	<code>fileno</code>	Stream status inquiries.
<code>fopen</code>	<code>freopen</code>	<code>fdopen</code>		Open a stream.
<code>fread</code>	<code>fwrite</code>			Input/output.
<code>fseek</code>	<code>rewind</code>	<code>ftell</code>		Reposition a file pointer in a stream.
<code>getc</code>	<code>getchar</code>	<code>fgetc</code>	<code>getw</code>	Get a character or word from a stream.
<code>gets</code>	<code>fgets</code>			Get a string from a stream.
<code>popen</code>	<code>pclose</code>			Begin or end a pipe to/from a process.
<code>printf</code>	<code>fprintf</code>	<code>sprintf</code>		Print formatted output.
<code>putc</code>	<code>putchar</code>	<code>fputc</code>	<code>putw</code>	Put a character or word on a stream.
<code>puts</code>	<code>fputs</code>			Put a string on a stream.
<code>scanf</code>	<code>fscanf</code>	<code>sscanf</code>		Convert formatted input.
<code>setbuf</code>	<code>setvbuf</code>			Assign buffering to a stream.
<code>system</code>				Issue a command through the shell.
<code>tmpfile</code>				Create a temporary file.
<code>tmpnam</code>	<code>tempnam</code>			Create a name for a temporary file.
<code>ungetc</code>				Push character back into input stream.
<code>vprintf</code>	<code>vfprintf</code>	<code>vsprintf</code>		Print formatted output of a <i>varargs</i> argument list.

String, character, environment and memory routines

These are functions and macros that perform a variety of tasks:

- string manipulation
- character classification
- character conversion
- environment management
- memory management

The following tables list functions and macros that perform the first three tasks.

Table 2-3, “String operations” (this page) lists string-handling functions that appear in **string(S)**. Programs that use these functions should include the header file *string.h*.

Table 2-3 String operations

<code>strcat</code>	Append a copy of one string to the end of another.
<code>strncat</code>	Append no more than a given amount of characters from one string to the end of another.
<code>strcmp</code>	Compare two strings. Returns an integer less than, greater than, or equal to 0 to show that one is lexicographically less than, greater than, or equal to the other.
<code>strncmp</code>	Compare no more than a given amount of characters from the two strings. Results are otherwise identical to <i>strcmp</i> .
<code>strcpy</code>	Copy a string.
<code>strncpy</code>	Copy a given amount of characters from one string to another. The destination string will be truncated if it is longer than the given amount of characters, or padded with null characters if it is shorter.
<code>strdup</code>	Return a pointer to a newly allocated string that is a duplicate of a string pointed to.
<code>strchr</code>	Return a pointer to the first occurrence of a character in a string, or a null pointer if the character is not in the string.
<code>strrchr</code>	Return a pointer to the last occurrence of a character in a string, or a null pointer if the character is not in the string.
<code>strlen</code>	Return the number of characters in a string.
<code>strpbrk</code>	Return a pointer to the first occurrence in one string of any character from the second, or a null pointer if no character from the second occurs in the first.
<code>strspn</code>	Return the length of the initial segment of one string that consists entirely of characters from the second string.
<code>strcspn</code>	Return the length of the initial segment of one string that consists entirely of characters not from the second string.
<code>strstr</code>	Return a pointer to the first occurrence of the second string in the first string, or a null pointer if the second string is not found.
<code>strtok</code>	Break up the first string into a sequence of tokens, each of which is delimited by one or more characters from the second string. Return a pointer to the token, or a null pointer if no token is found.

Table 2-4, “Classifying 8-bit character-coded integer values” (this page) lists functions and macros that classify 8-bit character-coded integer values. These routines appear in **toascii(S)** and **ctype(S)**. Programs that use these routines should include the header file *ctype.h*.

Table 2-4 Classifying 8-bit character-coded integer values

<code>isalpha</code>	Is <i>c</i> a letter?
<code>isupper</code>	Is <i>c</i> an uppercase letter?
<code>islower</code>	Is <i>c</i> a lowercase letter?
<code>isdigit</code>	Is <i>c</i> a digit [0-9]?
<code>isxdigit</code>	Is <i>c</i> a hexadecimal digit [0-9], [A-F], or [a-f]?
<code>isalnum</code>	Is <i>c</i> alphanumeric (a letter or digit)?
<code>isspace</code>	Is <i>c</i> a space, horizontal tab, carriage return, new-line, vertical tab, or form-feed?
<code>ispunct</code>	Is <i>c</i> a punctuation character (neither control nor alphanumeric)?
<code>isprint</code>	Is <i>c</i> a printing character?
<code>isgraph</code>	Same as <i>isprint</i> except false for a space.
<code>iscntrl</code>	Is <i>c</i> a control character or a delete character?
<code>isascii</code>	Is <i>c</i> an ASCII character?
<code>toupper</code>	Change lowercase to uppercase.
<code>_toupper</code>	Macro version of <i>toupper</i> .
<code>tolower</code>	Change uppercase to lowercase.
<code>_tolower</code>	Macro version of <i>tolower</i> .
<code>toascii</code>	Turn off all bits that are not part of a standard ASCII character; intended for compatibility with other systems.

Table 2-5, “Converting characters, integers, or strings” (this page) lists functions and macros that are used to convert characters, integers, or strings from one representation to another. The left column contains the name that appears at the top of the manual page; the other names in the same row are related functions or macros described on the same manual page. Programs that use these routines should include the header file *stdlib.h*.

Table 2-5 Converting characters, integers, or strings

a64l	l64a		Convert between long integer and base-64 ASCII string.
ecvt	fcvt	gcv	Convert floating point number (long double number) to string.
	ecvtl	fcvtl	
		gcvtl	
l3tol	ltol3		Convert between 3-byte packed integer and long integer.
strtod	atof	strtold	Convert string to double-precision number.
strtol	atol	atoi	Convert string to integer.
strtoul			Convert string to unsigned long.

System calls

UNIX operating system calls are the interface between the kernel and the user programs that run on top of it. **read(S)**, **write(S)**, and the other system calls (see **Routines(S)**) define the UNIX operating system. Everything else is built on their foundation. Strictly speaking, they are the only way to access such facilities as the file system, interprocess communication primitives, and multitasking mechanisms.

Of course, most programs do not need to invoke system calls directly to gain access to these facilities. If you are performing input/output, for example, you can use the standard I/O functions described earlier. When you use these functions, the details of their implementation on the UNIX operating system — for example, that the system call **read()** underlies the **fread()** implementation in the standard C library — are transparent to the program. Therefore, the program is generally be portable to any system with a conforming C implementation.

In contrast, programs that invoke system calls directly are portable only to other systems similar to the UNIX operating system. Therefore, you would not use **read()** in a program that performed a simple I/O operation. Other operations, however, including most multitasking mechanisms, do require direct interaction with the UNIX operating system kernel. For general discussion about UNIX operating systems see *Advanced Programming in the UNIX Environment* by W. Richard Stevens.

Math library (libm)

The math library, *libm*, contains the mathematics functions supplied by the C compilation system. This section describes some of the major functions, organized by the manual page on which they appear. Note that functions whose names end with the letter “f” are single-precision versions, which means that their argument and return types are **float**. Functions whose names end with the letter “l” are long double-precision versions, which means that their argument and return types are **long double**. The header file *math.h* should be included in programs that use math functions.

Table 2-6 Math functions

exp(S)		
<code>exp</code>	<code>expf</code>	Return e^x .
<code>cbrt</code>		Return cube root of x .
<code>log</code>	<code>logf</code>	Return the natural logarithm of x . The value of x must be positive.
<code>log10</code>	<code>log10f</code>	Return the base-ten logarithm of x . The value of x must be positive.
<code>pow</code>	<code>powf</code>	Return x^y . If x is zero, y must be positive. If x is negative, y must be an integer.
<code>sqrt</code>	<code>sqrtf</code>	Return the non-negative square root of x . The value of x must be non-negative.
hypot(S)		
<code>hypot</code>		Return $\sqrt{x * x + y * y}$, taking precautions against overflows.
gamma(S)		
<code>gamma</code>	<code>lgamma</code>	Return $\ln(\Gamma(x))$, where $\Gamma(x)$ is defined as $\int_0^x e^{-t} t^{x-1} dt$.
trig(S)		
<code>sin</code>	<code>sinf</code>	Return, respectively, the sine, cosine, and tangent of x , measured in radians.
<code>cos</code>	<code>cosf</code>	
<code>tan</code>	<code>tanf</code>	

(Continued on next page)

Table 2-6 Math functions*(Continued)*

asin	asinf	Return the arcsine of x , in the range $[-\pi/2, +\pi/2]$.
acos	acosf	Return the arccosine of x , in the range $[0,+\pi]$.
atan	atanf	Return the arctangent of x , in the range $(-\pi/2, +\pi/2)$.
atan2	atan2f	Return the arctangent of y/x , in the range $(-\pi, +\pi]$, using the signs of both arguments to determine the quadrant of the return value.
sinh(S)		
sinh	sinhf	Return, respectively, the hyperbolic sine, cosine, and tangent of their argument.
cosh	coshf	
tanh	tanhf	
asinh acosh atanh		
Return, respectively, the inverse hyperbolic sine, cosine, and tangent of their argument.		
matherr(S)		
matherr		Error handling.
erf(S)		
erf		Returns the error function of x , defined as $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.
erfc		erfc , which returns $1.0 - \text{erf}(x)$, is provided because of the extreme loss of relative accuracy if erf is called for large x and the result subtracted from 1.0 (for example, for $x = 5$, 12 places are lost).
floor(S)		
floor	floorf	Return the largest integer not greater than x .
ceil	ceilf	Return the smallest integer not less than x .

(Continued on next page)

Table 2-6 Math functions
(Continued)

copysign		Return x but with the sign of y .
fmod	fmodf	Return the floating point remainder of the division of x by y : x if y is zero, otherwise the number f with same sign as x , such that $x = iy + f$ for some integer i , and $ f < y $.
fabs	fabsf	Return the absolute value of x , $ x $.
rint		Return the integer value nearest to the double-precision floating point argument x as a double-precision floating point number. The returned value is rounded according to the currently set machine rounding mode. If round-to-nearest (the default mode) is set and the difference between the function argument and the rounded result is exactly 0.5, then the result will be rounded to the nearest even integer.
remainder		Return the floating point remainder of the division of x by y : NaN if y is zero, otherwise the value $r = x - yn$, where n is the integer nearest the exact value of x . Whenever $ n - x/y = 1/2$, then n is even.

General purpose library (libgen)

libgen contains general purpose functions. Table 2-7, “libgen functions” (this page) describes some of the more frequently used functions in *libgen*.

Table 2-7 libgen functions

advance	step	Execute a regular expression on a string.
basename		Return a pointer to the last element of a path-name.
bgets		Read a specified number of characters into a buffer from a stream until a specified character is reached.
bufsplit		Split the buffer into fields delimited by tabs and new-lines.
compile		Return a pointer to a compiled regular expression that uses the same syntax as ed .
copylist		Copy a file into a block of memory, replacing new-lines with null characters. It returns a pointer to the copy.
dirname		Return a pointer to the parent directory name of the file pathname.
eaccess		Determine if the effective user ID has the appropriate permissions on a file.
gmatch		Check if name matches shell filename pattern.
isencrypt		Use heuristics to determine if contents of a character buffer are encrypted.
mkdirp		Create a directory and its parents.
p2open	p2close	p2open is similar to popen(S) . It establishes a two-way connection between the parent and the child. p2close closes the pipe.
pathfind		Search the directories in a given path for a named file with given mode characteristics. If the file is found, a pointer is returned to a string that corresponds to the pathname of the file. A null pointer is returned if no file is found.
regcmp		Compile a regular expression and return a pointer to the compiled form.
regex		Compare a compiled regular expression against a subject string.

(Continued on next page)

Table 2-7 libgen functions
(Continued)

<code>rmdirp</code>		Remove the directories in the specified path.
<code>strccpy</code>	<code>strcadd</code>	strccpy copies the input string to the output string, compressing any C-like escape sequences to the real character. strcadd is a similar function that returns the address of the null byte at the end of the output string.
<code>strecpy</code>		Copy the input string to the output string, expanding any non-graphic characters with the C escape sequence. Characters in a third argument are not expanded.
<code>strfind</code>		Return the offset of the first occurrence of the second string in the first string. -1 is returned if the second string does not occur in the first.
<code>strrspn</code>		Trim trailing characters from a string. It returns a pointer to the last character in the string not in a list of trailing characters.
<code>strtrns</code>		Return a pointer to the string that results from replacing any character found in two strings with a character from a third string. This function is similar to the tr command.

Standard I/O

The functions and macros listed in Table 2-2, “Standard I/O functions and macros” (page 68) constitute the standard I/O library for C programs. I/O involves:

- reading information from a file or device to your program
- writing information from your program to a file or device
- opening and closing files that your program reads from or writes to

This section discusses:

- Standard files
- Named files
- Passing command line arguments

Standard files

Programs automatically start off with three open files: standard input, standard output, and standard error. These files with their associated buffering are called streams, and are designated **stdin**, **stdout**, and **stderr**, respectively. The shell associates all three files with your terminal by default.

You can use functions and macros that deal with **stdin**, **stdout**, or **stderr** without having to open or close files. **gets()**, for example, reads a string from **stdin**; **puts()** writes a string to **stdout**. Other functions and macros read from or write to files in different ways: one character at a time, **getc()** and **putc()**; formatted, **scanf()** and **printf()**; and so on. You can specify that output be directed to **stderr** by using a function such as **fprintf()**. **fprintf()** works the same way as **printf()** except that it delivers its formatted output to a named stream, such as **stderr**.

Named files

Any file other than standard input, standard output, and standard error must be explicitly opened before your program can read from or write to the file. You open a file with the standard library function **fopen()**. **fopen()** takes a pathname, asks the system to keep track of the connection between your program and the file, and returns a pointer that you can then use in functions that perform other I/O operations.

The pointer is to a structure of type **FILE**, defined in *stdio.h*, that contains information about the file including the location of its buffer, and the current character position in the buffer. Your program must have a declaration such as:

```
FILE *fin;
```

which says that **fin** is a pointer to a **FILE**. The statement:

```
fin = fopen("filename", "r");
```

associates a structure of type **FILE** with *filename*, the pathname of the file to open, and returns a pointer to it. The "r" means that the file is to be opened for reading. This argument is known as the mode. There are modes for reading, writing, and both reading and writing.

In practice, the file open function is often included in an **if** statement:

```
if ((fin = fopen("filename", "r")) == NULL)
    (void)fprintf(stderr, "Cannot open input file %s\n",
        "filename");
```

which uses **fopen()** to return a **NULL** pointer if it cannot open the file. To avoid falling into the immediately following code on failure, you can call **exit()**, which causes your program to quit:

```
if ((fin = fopen("filename", "r")) == NULL) {
    (void)fprintf(stderr, "Cannot open input file %s\n",
        "filename");
    exit(1);
}
```

Once you have opened the file, use the pointer **fin** in functions or macros to refer to the stream associated with the opened file:

```
int c;
c = getc(fin);
```

brings in one character from the stream into an integer variable called **c**. The variable **c** is declared as an integer even though it is reading characters because **getc()** returns an integer. Getting a character is often incorporated in some flow-of-control mechanism such as:

```
while ((c = getc(fin)) != EOF)
    .
    .
    .
```

that reads through the file until **EOF** is returned. **EOF**, **NULL**, and the macro **getc()** are all defined in *stdio.h*. **getc()** and other macros in the standard I/O package keep advancing a pointer through the buffer associated with the stream. The UNIX operating system and the standard I/O functions are responsible for seeing that the buffer is refilled if you are reading the file, or written to the output file if you are producing output, when the pointer reaches the end of the buffer.

Your program may have multiple files open simultaneously, 20 or more depending on system configuration. If your program needs to open more files than it is permitted to have open simultaneously, you can use the standard library function **fclose()** to free up one of the structures of type **FILE** by breaking the connection between your pointer to the structure and the pathname of the file your program has opened. You can now make another call to **fopen()**. For output files, an **fclose()** call makes sure that all output has been sent from the output buffer before disconnecting the file. **exit()** closes all open files for you, but it also gets you completely out of your process, so you should use it only when you are sure you are finished.

Passing command line arguments

As indicated in “Compiling and linking” (page 34), information or control data can be passed to a C program as an argument on the command line. When you execute the program, command line arguments are made available to the function `main()` in two parameters, an argument count, conventionally called `argc`, and an argument vector, conventionally called `argv`. `argc` is the number of arguments with which the program was invoked. `argv` is an array of pointers to characters strings that contain the arguments, one per string. Because the command name itself is considered to be the first argument, or `argv[0]`, the count is always at least.

If you plan to accept run-time parameters in your program, you need to include code to deal with the information. Example 2-3, “Using `argv[1]` to pass a filename” (page 80) and Example 2-4, “Using command line arguments to set flags” (page 81) show program fragments that illustrate two common uses of run-time parameters:

- Example 2-3, “Using `argv[1]` to pass a filename” (page 80) shows how you provide a variable filename to a program, such that a command of the form:

\$ prog filename

causes *prog* to attempt to open the specified file.

- Example 2-4, “Using command line arguments to set flags” (page 81) shows how you set internal flags that control the operation of a program, such that a command of the form:

prog -opr

causes *prog* to set the corresponding variables for each of the options specified. The `getopt()` function used in the example is the most common way to process arguments in UNIX operating system programs. See `getopt(3)` for more information.

Example 2-3 Using argv[1] to pass a filename

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    FILE *fin;
    int ch;

    switch (argc)
    {
    case 2:
        if ((fin = fopen(argv[1], "r")) == NULL)
        {
            /* First string (%s) is program name (argv[0]). */
            /* Second string (%s) is name of file that could */
            /* not be opened (argv[1]). */

            (void)fprintf(stderr, "%s: Cannot open input file %s\n",
                          argv[0], argv[1]);
            return(2);
        }
        break;
    case 1:
        fin = stdin;
        break;

    default:
        (void)fprintf(stderr, "Usage: %s [file]\n", argv[0]);
        return(2);
    }

    while ((ch = getc(fin)) != EOF)
        (void)putchar(ch);

    return (0);
}
```

Example 2-4 Using command line arguments to set flags

```

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

int
main(int argc, char *argv[])
{
    int oflag = 0;
    int pflag = 0;    /* Function flags */
    int rflag = 0;
    int ch;

    while ((ch = getopt(argc, argv, "opr")) != -1)
    {
        /* For options present, set flag to 1.          */
        /* If unknown options present, print error message. */

        switch (ch)
        {
            case 'o':
                oflag = 1;
                break;
            case 'p':
                pflag = 1;
                break;
            case 'r':
                rflag = 1;
                break;
            default:
                (void)fprintf(stderr, "Usage: %s [-opr]\n", argv[0]);
                return(2);
        }
    }
    /* Do other processing controlled by oflag, pflag, rflag. */
    return(0);
}

```


Chapter 3

C language compiler

This chapter is a guide to the SCO C language compiler. It assumes that you have some experience with C, and are familiar with fundamental programming concepts.

The compiler has several compilation modes. The compiler processes a source file into an executable in eight conceptual steps, which are called phases of translation.

The compiler is compatible with the C language described in the American National Standards Institute (ANSI) *American National Standard for Information Systems — Programming Language C* document number X3.159-1989. The standard language is referred to as “ANSI C” in this document. The compiler is also compatible with the C language described in the International Standards Organization (ISO) *Programming Languages — C*, document number ISO/IEC 9899:1990 (E), referred to as “ISO C.” The standards leave a list of the characteristics and extensions to be implementation-defined; these are documented in Appendix A, “ANSI implementation-defined behavior” (page 485).

You can use this chapter either as a quick reference guide, or as a comprehensive summary of the language as implemented by the compilation system. Many topics are grouped according to their place in the ANSI-specified phases of translation. At the end of this chapter is a discussion of portability issues arising from machine dependencies.

Compilation modes

The compilation system has five compilation modes that correspond to degrees of compliance with ANSI C. The modes are:

- Xa Extended ANSI C (the default). This specifies ANSI and ISO C standards conformance except that some required warnings are omitted

and the name space is expanded to include names that are not specified by the standards. All C language constructions behave as specified in the standards. All implemented language extensions beyond the standards are also available.

- Xc Conformant ANSI and ISO C. This is a strict standards conforming C compiler satisfying the language specification in all respects, including issuing all necessary diagnostics for illegal programs. Certain extensions (such as the **asm** keyword) are not available in this dialect.
- Xk The “classic” C, or the dialect compatible with K&R C (See *The C Programming Language*, Kernighan and Ritchie, First Edition).
- Xm A Microsoft C compiler compatibility mode. This is equivalent to option -Xa, plus some (but not all) language extensions provided in the previous release of SCO C compiler, which was derived from Microsoft C compiler version 6. In particular, some Intel® specific keywords like **near** and **far** are supported. This is intended as a transitional aid and may not be supported in future releases.
- Xt A “transitional mode” between K&R and ANSI C. This specifies ANSI and ISO C standards conformance except where the semantics differ from classic C. In addition, warnings are issued about preprocessing phase semantic changes such as new escape sequences (like **\x**) and any trigraph replacements (like **??!**). Certain potential optimizations available in the other -X modes are also disabled.

Global behavior

A program that depends on unsigned-preserving arithmetic conversions will behave differently. This is considered to be the most serious change made by ANSI C to a widespread current practice.

In *The C Programming Language*, Kernighan and Ritchie, (First Edition), **unsigned** specified exactly one type; there were no **unsigned chars**, **unsigned shorts**, or **unsigned longs**, but most C compilers added these very soon thereafter.

In many pre-ANSI C compilers, the “unsigned-preserving” rule is used for promotions: when an **unsigned** type needs to be widened, it is widened to an **unsigned** type; when an **unsigned** type mixes with a **signed** type, the result is an **unsigned** type.

The other rule, specified by ANSI C, came to be called “value-preserving,” in which the result type depends on the relative sizes of the operand types. When an **unsigned char** or **unsigned short** is “widened,” the result type is **int** if an **int** is large enough to represent all the values of the smaller type. Otherwise the result type would be **unsigned int**. The “value-preserving” rule produces the “least surprise” arithmetic result for most expressions.

Only in the transition (-Xt) or classic (-Xk) mode will the compiler use the unsigned-preserving promotions; in the other three modes, -Xc, -Xa and -Xm, the value-preserving promotion rules will be used. No matter what the current mode may be, the compiler can warn about each expression whose behavior might depend on the promotion rules used.

This warning is given if the warning level is 2 (-w2) or 3 (-w2).

Phases of translation

The compiler processes a source file into an executable in eight conceptual steps, which are called phases of translation. While some of these phases may actually be folded together, the compiler behaves as if they occur separately, in sequence.

1. Trigraph sequences are replaced by their single-character equivalents. Trigraph sequences are explained in “Preprocessing” (page 91).
2. Any source lines that end with a backslash and new-line are spliced together with the next line by deleting the backslash and new-line to form logical lines.
3. The source file is partitioned into preprocessing tokens and sequences of white-space characters. Each comment is, in effect, replaced by one space character. Preprocessing tokens are explained in “Preprocessing” (page 91).
4. Preprocessing directives are executed, and macros are expanded. Any files named in **#include** statements are processed from phase 1 through phase 4, recursively.
5. Escape sequences in character constants and string literals are converted to their character equivalents.
6. Adjacent character string literals, and wide character string literals, are concatenated.
7. Each preprocessing token is converted into a token (page 86). The resulting tokens are syntactically and semantically analyzed and translated.
8. All external object and function references are resolved. Libraries are linked to satisfy external references not defined in the current translation unit. All translator output is collected into a program image which contains information needed for execution.

Output from certain phases may be saved and examined by specifying option flags on the cc(CP) command line.

The preprocessing token sequence resulting from Phase 4 can be saved by using the following options:

- P leaves preprocessed output in a file with a *i* extension.
- E sends preprocessed output to the standard output.

Output from Phase 7 can be saved in a file with a *.o* extension by using the **-c** option to **cc**. The output of Phase 8 is the compilation system's final output (*a.out*).

Source files and tokenization

This section summarizes five of the six classes of regular tokens that the C compilation system partitions into from a source file during its third phase of translation. The remaining class, operators, is explained in “Operators” (page 109). Preprocessing tokens are a superset of regular tokens and are summarized in “Preprocessing tokens” (page 92).

Tokens

A token is a series of contiguous characters that the compiler treats as a unit. Translation phase 3 partitions a source file into a sequence of tokens. Tokens fall into six classes:

- Identifiers
- Keywords
- Constants, which can be further classified into numeric constants and character constants
- String literals
- Operators
- Punctuators and other separators

Identifiers

- Identifiers are used to name things such as variables, functions, data types, and macros.
- Identifiers are made up of a combination of letters, digits, or underscore (`_`) characters.
- First character may not be a digit.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

asm	default	for	short	union
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	

The keyword **asm** is reserved in all compilation modes except **-Xc**. The keyword **_asm** is a synonym for **asm** and is available under all compilation modes, although a warning issued when it is used under the **-Xc** mode.

NOTE Under the **-Xm** mode, the keywords:

cdecl near far huge

are reserved but ignored. **cdecl** is silently ignored; the other three will cause a warning to be issued.

Under the **-Xk** mode, the following are considered identifiers and are **not** reserved as keywords:

const signed volatile

Constants

Constants in the C language include:

- Integral constants
- Floating point constants
- Character constants
- Enumeration constants

The first three types, integral constants, floating point constants, and character constants, are explained in this subsection. Enumeration constants are explained in “Enumerations” (page 101).

Integral constants

Integral constants may be:

- Decimal:
 - Digits: “0-9”.
 - The first digit may not be “0” (zero).
- Octal:
 - Digits: “0-7”.
 - The first digit must be “0” (zero).
- Hexadecimal:
 - Digits: “0-9” plus letters “a-f” or “A-F”. Letters correspond to decimal values 10-15.
 - Prefixed by “0x” or “0” (digit zero).
- Suffixes:

All of the above can be suffixed to indicate type, as follows:

Suffix	Type
u or U	unsigned
l or L	long
both	unsigned long

Floating point constants

- Consist of integer part, decimal point, fraction part, an **e** or **E**, an optionally signed integer exponent, and a type suffix, one of **f**, **F**, **l**, or **L**. Each of these elements is optional; however one of the following must be present for the constant to be a floating point constant:
 - A decimal point (preceded or followed by a number).
 - An **e** with an exponent.
 - Any combination of the above. Examples:

```
xxx e exp
xxx.
.xxx
```

Type determined by suffix; **f** or **F** indicates **float**, **l** or **L** indicates **long double**, otherwise type is **double**.

Character constants

- One or more characters enclosed in single quotes, as in 'x'.
- All character constants have type **int**.
- The value of a character constant is the numeric value of the character in the ASCII character set.
- A multiple-character constant that is not an escape sequence (see below) has a value derived from the numeric values of each character. For example, the constant '123' has a value of

0	'1'	'2'	'3'
---	-----	-----	-----

or 0x313233.

- Character constants may not contain the character ' or new-line. To represent these characters, and some others that may not be contained in the source character set, the compiler provides the following escape sequences:

Table 3-1 Escape sequences

new-line	NL (LF)	\n	audible alert	BEL	\a
horizontal tab	HT	\t	question mark	?	\?
vertical tab	VT	\v	double quote	"	\"
backspace	BS	\b	octal escape	ooo	\ooo
carriage return	CR	\r	hexadecimal escape	hh	\xhh
formfeed	FF	\f	backslash	\	\\
single quote	'	\'			

If the character following a backslash is not one of those specified, the compiler issues a warning and treat the backslash-character sequence as the character itself. Thus, '\q' is treated as 'q'. However, if you represent a character this way, you run the risk that the character may be made into an escape sequence in the future, with unpredictable results. An explicit new-line character is invalid in a character constant and causes an error message.

- The octal escape consists of one to three octal digits.
- The hexadecimal escape consists of one or more hexadecimal digits



Wide characters and multibyte characters

- A wide character constant is a character constant prefixed by the letter `L`.
- A wide character has an external encoding as a multibyte character and an internal representation as the integral type `wchar_t`, defined in `stddef.h`.
- A wide character constant has the integral value for the multibyte character between single quote characters, as defined by the locale-dependent mapping function `mbtowc`.

String literals

- One or more characters surrounded by double quotes, as in `"xyz"`.
- Initialized with the characters contained in the double quotes.
- Have **static** storage duration and type “array of characters.”
- Escape sequences (page 89) may also be used in string literals. A double quote within the string must be escaped with a backslash. New-line characters are not valid within a string.
- Adjacent string literals are concatenated into a single string. A null character, `\0`, is appended to the result of the concatenation, if any.
- String literals are also known as “string constants.”

Wide string literals

- A wide-character string literal is a string literal immediately prefixed by the letter `L`.
- Wide-character string literals have type “array of `wchar_t`.”
- Wide string literals may contain escape sequences, and they may be concatenated, like ordinary string literals.

Punctuators

Punctuators can be any one of the following:

[] () { } * , : = ; ...

Depending on context, some of these punctuators can also be an operator (page 109) or part of an operator. The pairs of punctuators `[ ]`, `( )`, and `{ }` must match and appear in pairs after a source file is preprocessed. The punctuator `#` can only be part of preprocessing directive.

Comments

Comments begin with the characters `/*` and end with the next `*/`.

```
/* this is a comment */
```

Except when the option `-Xc` is used, everything following `//` on the same logical line are also treated as comments:

```
// this is also a comment
```

Comments do not nest.

If a comment appears to begin within a string literal or character constant, it will be taken as part of the literal or constant, as specified by the phases of translation.

```
char *p = "/* this is not a comment */"; /* but this is */
```

Preprocessing

- Preprocessing handles macro substitution, conditional compilation, and file inclusion.
- Preprocessing replaces trigraph sequences with their single-character equivalents, partitions source files into preprocessing tokens, and executes preprocessing directives.
- Lines that end with a backslash character `"\"` and new-line are joined with the next line by deleting the backslash and the new-line characters, so that each line in the source file is a logical line. This occurs (in translation phase 2) before input is divided into tokens.
- Lines beginning with `#` are preprocessing directives. Spaces and tabs may appear before and after the `#`.
- Each preprocessing directive must appear on a logical line by itself.

Trigraph sequences

Trigraph sequences are three-character sequences that are replaced by a corresponding single character in Translation Phase 1, as follows:

??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	?!!		??-	~

No other such sequences are recognized. The trigraph sequences provide a way to specify characters that are missing on some terminals, but that the C language uses.

Preprocessing tokens

A token is the basic lexical unit of the language. All source input must be formed into valid tokens by translation phase seven. Preprocessing tokens (pp-tokens) are a superset of regular tokens. Preprocessing tokens allow the source file to contain non-token character sequences that constitute valid preprocessing tokens during translation. There are four categories of preprocessing tokens:

- Header filenames, meant to be taken as a single token.
- Preprocessing numbers (this page)
- All other single characters that are not otherwise (regular) tokens. See the example in “Preprocessing numbers” (this page).
- Identifiers, numeric constants, character constants, string literals, operators, and punctuators.

Preprocessing numbers

- A preprocessing number is made up of a digit, optionally preceded by a period, and may be followed by letters, underscores, digits, periods, and any one of: `e+` `e-` `E+` `E-`.
- Preprocessing numbers include all valid number tokens, plus some that are not valid number tokens. For example, in the macro definition:

```
#define R 2e ## 3
```

the preprocessing number `2e` is not a valid number. However, the preprocessing operator `##` will “paste” it together with the preprocessing number `3` when `R` is replaced, resulting in the preprocessing number `2e3`, which is a valid number.

Preprocessing directives

A preprocessing directive begins with a preprocessing operator, `#`, on a logical line. Only white spaces are allowed to precede `#` on the same line. Preprocessing directives can be divided into the following categories:

- Macro definition and expansion
- File inclusion
- Conditional compilation

- Line control
- Assertions
- Version control
- Pragmas
- Error generation
- Predefined names

Preprocessing operators

The preprocessing operators are evaluated left to right, without any defined precedence.

- #** A macro parameter preceded by the **#** preprocessing operator has its corresponding unexpanded argument tokens converted into a string literal. (Any double quotes and backslashes contained in character constants or part of string literals are escaped by a backslash). The **#** character is sometimes referred to as the “stringizing” operator. This rule applies only within function-like macros.
- ##** If a replacement token sequence contains a **##** operator, the **##** and any surrounding white space are deleted and adjacent tokens are concatenated, creating a new token. This occurs only when the macro is expanded. See “Macro definition and expansion” (this page).

Macro definition and expansion

- An object-like macro is defined with a line of the form:

```
#define identifier token-sequenceopt
```

 where *identifier* will be replaced with *token-sequence* wherever *identifier* appears in regular text.
- A function-like macro is defined with a line of the form:

```
#define identifier (identifier-listopt) token-sequenceopt
```

 where the macro parameters are contained in the comma-separated *identifier-list*. The token-sequence following the identifier list determines the behavior of the macro, and is referred to as the replacement list. There can be no space between the identifier and the “(” character. For example:

```
#define FLM(a,b) a+b
```

 The replacement-list “a+b” determines that the two parameters “a” and “b” will be added.

- A function-like macro is invoked in normal text by using its identifier, followed by a "(" token, a list of token sequences separated by commas, and a ")" token. For example:

```
FLM(3,2)
```

- The arguments in the invocation (comma-separated token sequences) may be expanded, and they then replace the corresponding parameters in the replacement token sequence of the macro definition. Macro arguments in the invocation are not expanded if they are operands of # or ## operators in the replacement string. Otherwise, expansion does take place. For example, assume that **M1** is defined as "3":

```
#define M1 3
```

When the function-like macro **FLM** is used, use of the # or ## operators will affect expansion (and the result), as follows:

Definition	Invocation	Result	Expansion ?
a+b	FLM(M1, 2)	3+2	Yes, Yes
#a	FLM(M1, 2)	"M1"	No
a##b	FLM(M1, 2)	M12	No, No
a+#a	FLM(M1, 2)	3+"M1"	Yes, No

In the last example line, the first "a" in "a+#a" is expanded, but the second "a" is not expanded because it is an operand of the # operator.

- The number of arguments in the invocation must match the number of parameters in the definition.
- A macro's definition, if any, can be eliminated with a line of the form:

```
#undef identifier
```

There is no effect if the definition does not exist.

File inclusion

- A line of the form:

```
#include "filename"
```

causes the entire line to be replaced with the contents of *filename*. The following directories are searched, in order.

- The current directory (of the file containing the **#include** line).
- Any directories named in -I options to the compiler, in order.
- A list of standard places, typically, but not necessarily, */usr/include*.

- A line of the form:

```
#include <filename>
```

causes the entire line to be replaced with contents of *filename*. The angle brackets surrounding *filename* indicate that *filename* is not searched for in the current directory.

- A third form allows an arbitrary number of preprocessing tokens to follow the **#include**, as in:

```
#include preprocessing-tokens
```

The preprocessing tokens are processed the same way as when they are used in normal text. Any defined macro name is replaced with its replacement list of preprocessing tokens. The preprocessing tokens must expand to match one of the first two forms (<...> or "...").

- A filename beginning with a slash "/" indicates the absolute pathname of a file to include, no matter which form of **#include** is used.
- Any **#include** statements found in an included file cause recursive processing of the named file(s).

Conditional compilation

Different segments of a program may be compiled conditionally. Conditional compilation statements must observe the following sequence:

1. One of: **#if** or **#ifdef** or **#ifndef**.
2. Any number of optional **#elif** lines.
3. One optional **#else** line.
4. One **#endif** line.

- **#if *integral-constant-expression***

Is true if *integral-constant-expression* evaluates to non-zero.

If true, tokens following the **#if** line are included.

The *integral-constant-expression* following the **#if** is evaluated by following this sequence of steps:

1. Any preprocessing tokens in the expression are expanded. Any use of the **#define** operator evaluates to "1" or "0" if its operand is, respectively, defined, or not.
2. If any identifiers remain, they evaluate to "0".

3. The remaining integral constant expression is evaluated. The constant expression must be made up of components that evaluate to an integral constant. In the context of a **#if**, the integral constant expression may not contain the **sizeof** operator, casts, or floating point constants.

The following table shows how various types of constant expressions following a **#if** would be evaluated. Assume that *name* is not defined.

Constant expression	Step 1	Step 2	Step 3
!defined(_STDC_)	!1	!1	0
3 <i>name</i>	3 <i>name</i>	3 0	1
2 + <i>name</i>	2 + <i>name</i>	2 + 0	2

- **#ifdef *identifier***

Is true if *identifier* is currently defined by **#define** or by the **-D** option to the **cc** command line.

- **#ifndef *identifier***

Is true if *identifier* is not currently defined by **#define** (or has been undefined).

- **#elif *constant-expression***

Indicates alternate if-condition when all preceding if-conditions are false.

- **#else**

Indicates alternate action when no preceding **if** or **elif** conditions are true. A comment may follow the **else**, but a token may not.

- **#endif**

Terminates the current conditional. A comment may follow the **endif** but a token may not.

Line control

- Useful for programs that generate C programs.
- A line of the form

```
#line constant "filename"
```

causes the compiler to believe, for the purposes of error diagnostics and debugging, that the line number of the next source line is equal to *constant* (which must be a decimal integer) and the current input file is *filename* (enclosed in double quotes). The quoted filename is optional. *constant* must be a decimal integer in the range 1 to **MAXINT**. **MAXINT** is defined in *limits.h*.

Assertions

A line of the form:

```
#assert predicate (token-sequence)
```

associates the *token-sequence* with the predicate in the assertion name space (separate from the space used for macro definitions). The predicate must be an identifier token.

```
#assert predicate
```

asserts that *predicate* exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicates by default:

```
#assert machine ( i386 )
#assert system ( unix )
#assert cpu ( i386 )
```

Any assertion may be removed by using **#unassert**, which uses the same syntax as **#assert**. Using **#unassert** with no argument deletes all assertions on the predicate; specifying an assertion deletes only that assertion.

An assertion may be tested in a **#if** statement with the following syntax:

```
#if #predicate(non-empty token-list)
```

For example, the predefined predicate **system** can be tested with the following line:

```
#if #system(unix)
```

which will evaluate true.

Version control

The **#ident** directive is used to help administer version control information.

```
#ident "version"
```

puts an arbitrary string in the **.comment** section of the object file. The **.comment** section is not loaded into memory when the program is executed.

Pragmas

- Preprocessing lines of the form

```
#pragma pp-tokens
```

specify implementation-defined actions.

- Six **#pragmas** are recognized by the compilation system:

```
#pragma asm [full_optimization|partial_optimization] asm_func
#pragma comment comment(argument)
#pragma ident version
#pragma int_to_unsigned identifier
#pragma pack(n)
#pragma weak identifier [= identifier2]
```

- The compiler ignores unrecognized pragmas.

For more information on pragmas, see “Pragmas” (page 496) in Appendix A, “ANSI implementation-defined behavior”.

Error generation

A preprocessing line consisting of:

```
#error token-sequence
```

causes the compiler to produce a diagnostic message containing the *token-sequence*, and stop.

Predefined names

The following identifiers are predefined as object-like macros:

<code>__LINE__</code>	The current line number as a decimal constant.
<code>__FILE__</code>	A string literal representing the name of the file being compiled.
<code>__DATE__</code>	The date of compilation as a string literal in the form " <i>Mmm dd yyyy</i> ."
<code>__TIME__</code>	The time of compilation, as a string literal in the form " <i>hh:mm:ss</i> ."
<code>__STDC__</code>	The constant 1 under compilation mode -Xc . Under compilation mode -Xk , it is undefined. Under all other compilation modes, it is constant "0".
<code>__USLC__</code>	A positive integer constant; its definition signifies a USL C compilation system.

With the exception of `__STDC__`, these predefined names may not be undefined or redefined. Under compilation mode **-Xt**, `__STDC__` may be undefined (**#undef __STDC__**) to cause a source file to think it is being compiled by a pre-ANSI C compiler.

Declarations and definitions

A declaration describes an identifier in terms of its type and storage duration. The location of a declaration (usually, relative to function blocks) implicitly determines the scope of the identifier. A function definition includes the body of the function after the declaration of the function.

Basic types

The basic types and their sizes are:

- **char** (1 byte)
- **short int** (2 bytes)
- **int** (4 bytes)
- **long int** (4 bytes) Each of **char**, **short**, **int**, and **long** may be prefixed with **signed** or **unsigned**. A type specified with **signed** is the same as the type specified without **signed**.
- **float** (4 bytes)
- **double** (8 bytes)
- **long double** (12 bytes)
- **void**

Integral and floating types are collectively referred to as *arithmetic types*. Arithmetic types and pointer types make up the *scalar types*. For more information see “Pointer declarators” (page 103).

Type qualifiers

- const** The compiler may place an object declared **const** in read-only memory. The program may not change its value and no further assignment may be made to it. An explicit attempt to assign to a **const** object will provoke an error.
- volatile** **volatile** advises the compiler that unexpected, asynchronous events may affect the object so declared, and warns it against making assumptions. An object declared **volatile** is protected from optimization that might otherwise occur.

Structures and unions

A structure is a type that consists of a sequence of named members. The members of a structure may have different object types (as opposed to an array, whose members are all of the same type). To declare a structure is to declare a new type. A declaration of an object of type **struct** reserves enough storage space so that all of the member types can be stored simultaneously.

A structure member may consist of a specified number of bits, called a bit-field. The number of bits (the size of the bit-field) is specified by appending a colon and the size (an integral constant expression, the number of bits) to the declarator that names the bit-field. The declarator name itself is optional; a colon and integer will declare the bit-field. A bit-field must have integral type. The size may be 0, in which case the declaration name must not be specified, and the structure is padded to a boundary of the type specified. For example:

```
int :0
```

means “pad to an **int** boundary before starting next member.”

A named bit-field number that is declared with an explicitly unsigned type holds values in the range:

$0-(2^n-1)$

where n is the number of bits. A bit-field declared with an explicit signed types holds values in the range

$-2^{n-1} - (2^{n-1}-1)$

A bit-field declared neither explicitly signed nor unsigned defaults to signed.

An optional structure tag identifier may follow the keyword **struct**. The tag names the type of structure described, and it and **struct** may then be used as a shorthand name for the declarations that make up the body of the structure. For example:

```
struct t {
    int x;
    float y;
} st1, st2;
```

Here, **st1** and **st2** are structures, each made up of **x**, an **int**, and **y**, a **float**. The tag **t** may be used to declare more structures identical to **st1** and **st2**, as in:

```
struct t st3;
```

A structure may include a pointer to itself as a member; this is known as a *self-referential structure*.

```
struct n {
    int x;
    struct n *left;
    struct n *right;
};
```

A union is an object that may contain one of several different possible member types. A union may have bit-field members. Like a structure, declaring a union declares a new type. Unlike a structure, a union stores the value of only one member at a given time. A union does, however, reserve enough storage to hold its largest member.

Enumerations

An enumeration is a unique type that consists of a set of constants called enumerators. The enumerators are declared as constants of type **int**, and optionally may be initialized by an integral constant expression separated from the identifier by an `=` character.

Enumerations consist of two parts:

- The set of constants.
- An optional tag.

For example:

```
enum color {red, blue=5, yellow};
```

color is the tag for this enumeration type. **red**, **blue**, and **yellow** are its enumeration constants. If the first enumeration constant in the set is not followed by an `=` its value is `"0"`. Each subsequent enumeration constant not followed by an `=` is determined by adding 1 to the value of the previous enumeration constant. Thus **yellow** has the value 6.

```
enum color car_color;
```

declares **car_color** to be an object of type **enum color**.

Scope

The use of an identifier is limited to an area of program text known as the identifier's scope. The four kinds of scope are function, file, block, and function prototype.

- The scope of every identifier (other than label names) is determined by the placement of its declaration (in a declarator or type specifier).
- The scope of structure, union and enumeration tags begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.
- If the declarator or type specifier appears outside a function or parameter list, the identifier has file scope, which terminates at the end of the file (and all included files).
- If the declarator or type specifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which ends at the end of the block (at the `"}`" that closes that block).

- If the declarator or type specifier appears in the list of parameter declarations in a function prototype declaration, the identifier has function prototype scope, which ends at the end of the function declarator (at the “)” that ends the list).
- Label names always have function scope. A label name must be unique within a function.

Storage duration

- Automatic Storage Duration

Storage is reserved for an automatic object, and is available for the object on each entry (by any means) into the block in which the object is declared. On any type of exit from the block, storage is no longer reserved.

- Static Storage Duration

An object declared outside any block, or declared with the keywords **static** or **extern**, has storage reserved for it for the duration of the entire program. The object retains its last-stored value throughout program execution.

Storage class specifiers

- | | |
|-----------------|---|
| auto | An object may be declared auto only within a function. It has block scope and the defined object has automatic storage duration. |
| register | A register declaration is equivalent to an auto declaration. It also advises the compiler that the object will be accessed frequently. |
| static | static gives a declared object static storage duration. For more information see “Storage duration” (this page). The object may be defined inside or outside functions. An identifier declared static with file scope has internal linkage. A function may be declared or defined with static . If a function is defined to be static, the function has internal linkage. A function may be declared with static at block scope; the function should be defined with static as well. |
| extern | extern gives a declared object static storage duration. An object or function declared with extern has the same linkage as any visible declaration of the identifier at file scope. If no file scope declaration is visible the identifier has external linkage. |
| typedef | Using typedef as a storage class specifier does not reserve storage. Instead, typedef defines an identifier that names a type. See the section on derived types for a discussion of typedef . |

Declarators

Declarators include:

- Pointer declarators
- Array declarators
- Function declarators

What follows is a brief summary of the syntax of declarators:

```

declarator:
    pointeropt direct-declarator
direct-declarator:
    identifier
    (declarator)
    direct-declarator [constant-expressionopt]
    direct-declarator (parameter-type-list)
    direct-declarator (identifier-listopt)
pointer:
    * type-qualifier-listopt
    * type-qualifier-listopt pointer
  
```

Pointer declarators

- Pointer to a type:

```
char *p;
```

p is a pointer to type **char**. **p** contains the address of a **char** object.

Care should be taken when pointer declarations are qualified with **const**:

```
const int *pci;
```

declares a pointer to a **const**-qualified “read-only” **int**.

```
int *const cpi;
```

declares a pointer-to-**int** that is itself “read-only”.

- Pointer to a pointer:

```
char **t;
```

t points to a character pointer.

- Pointer to a function:

```
int (*f)();
```

f is a pointer to a function that returns an **int**.

- Pointer to **void**:

```
void *
```

A pointer to **void** may be converted to or from a pointer to any object or

incomplete type, without loss of information. This generic pointer behavior was previously carried out by **char ***; a pointer to **void** has the same representation and alignment requirements as a pointer to a character type.

Array declarators

- One-dimensional array:

```
int ia[10];
```

ia is an array of 10 integers.

- Two-dimensional array:

```
char d[4][10];
```

d is an array of 4 arrays of 10 characters each.

- Array of pointers:

```
char *p[7];
```

p is an array of seven character pointers.

An array type of unknown size is known as an *incomplete type*.

Function declarators

- A function declaration includes the return type of the function, the function identifier, and an optional list of parameters.
- Function prototype declarations include declarations of parameters in the parameter list.
- If the function takes no arguments, the keyword **void** may be substituted for the parameter list in a prototype.
- A parameter type list may end with an ellipsis **"..."** to indicate that the function may take more arguments than the number described. The comma is necessary only if it is preceded by an argument.
- The parameter list may be omitted, which indicates that no parameter information is being provided.

Examples:

```
void srand(unsigned int seed);
```

The function **srand** returns nothing; it has a single parameter which is an **unsigned int**. The name *seed* goes out of scope at the **)** and as such serves solely as documentation.

```
int rand(void);
```

The function **rand** returns an **int**; it has no parameters.

```
int strcmp(const char *, const char *);
```

The function **strcmp** returns an **int**; it has two parameters, both of which are pointers to character strings that **strcmp** does not change.

```
void (*signal(int, void (*)(int)))(int);
```

The function **signal** returns a pointer to a function that itself returns nothing and has an **int** parameter; the function **signal** has two parameters, the first of which has type **int** and the second has the same type as **signal** returns.

```
int fprintf(FILE *stream, const char *format, . . . );
```

The function **fprintf** returns an **int**; **FILE** is a typedef name declared in *stdio.h*; **format** is a **const**

note the use of ellipsis (. . .) to indicate an unknown number of arguments.

Function definitions

A function definition includes the body of the function after the declaration of the function. As with declarations, a function may be defined as a function prototype definition or defined in the old style. The function prototype style includes type declarations for each parameter in the parameter list. This example shows how **main** would be defined in each style:

Function prototype style	Old style
<pre>int main(int argc, char *argv[]) { . . . }</pre>	<pre>int main(argc, argv) { int argc; char *argv[]; . . . }</pre>

Some important rules that govern function definitions:

- An old style definition names its parameters in an identifier list, and their declarations appear between the function declarator and the "{" that begins the function body.
- Under the old style, if the type declaration for a parameter was absent, the type defaulted to **int**. In the new style, all parameters in the parameter list must be type-specified and named. The exception to this rule is the use of ellipsis, explained in "Function declarators" (page 104).
- A function definition serves as a declaration.

- Incomplete types are not allowed in the parameter list or as the return type of a function definition. They are allowed in other function declarations.

Conversions and expressions

An expression (page 108) consists of constants, variables and operators. When constants and variables of different types are mixed in the same expression, you need to cast them into the same type (“explicit conversion”), otherwise the C compiler will convert them to the same type automatically (“implicit conversion”).

Implicit conversions

When constants and variables of different types are mixed in the same expression, C compiler will convert them to the same type automatically (“implicit conversion”) unless they are already casted to the same type. The following topics contain more details about implicit conversions:

- Characters and integers
- Signed and unsigned integers
- Integral and floating
- Float and double
- Usual arithmetic conversions

Characters and integers

Any of the following may be used in an expression where an **int** or **unsigned int** may be used.

- **char**
- **short int**
- A **char**, **short**, or **int** bit-field
- The signed or unsigned varieties of any of the above types
- An object or bit-field that has enumeration type

If an **int** can represent all values of the original type, the value is converted to an **int**; otherwise it is converted to an **unsigned int**. This process is called *integral promotion*.

Compilation mode dependencies that affect unsigned types

- Under compilation mode **-Xk** or **-Xt**, **unsigned char** and **unsigned short** are promoted to **unsigned int**.

- Under compilation mode **-Xa**, **-Xc**, or **-Xm**, **unsigned char** and **unsigned short** are promoted to **int**.

Signed and unsigned integers

- When an integer is converted to another integral type, the value is unchanged if the value can be represented by the new type.
- If a negative signed integer is converted to an unsigned integer with greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer.

Integral and floating

When a floating type is converted to any integral type, any fractional part is discarded.

Float and double

A **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double** without a change in value.

The actual rounding behavior that is used when a floating point value is converted to a smaller floating point value depends on the rounding mode in effect at the time of execution. The default rounding mode is “round to nearest.” See “Floating Point Operations” and the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) for a more complete discussion of rounding modes.

Usual arithmetic conversions

Some binary operators convert the types of their operands in order to yield a common type, which is also the type of the result. These are called the *usual arithmetic conversions*:

- If either operand is type **long double**, the other operand is converted to **long double**.
- Otherwise, if either operand has type **double**, the other operand is converted to **double**.
- Otherwise, if either operand has type **float**, the other operand is converted to **float**.
- Otherwise, the integral promotions are performed on both operands. Then, these rules are applied:
 - If either operand has type **unsigned long int**, the other operand is converted to **unsigned long int**.

- Otherwise, if one operand has type **long int** and the other has type **unsigned int**, both operands are converted to **unsigned long int**.
- Otherwise, if either operand has type **long int**, the other operand is converted to **long int**.
- Otherwise if either operand has type **unsigned int**, the other operand is converted to **unsigned int**.
- Otherwise, both operands have type **int**.

Expressions

Objects and lvalues

An object is a manipulatable region of storage. An lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators that yield lvalues: for example, if *E* is an expression of pointer type, then **E* is an lvalue expression referring to the object to which *E* points.

An lvalue is *modifiable* if it meets all criteria below:

- it does not have array type
- it does not have an incomplete type
- it does not have a **const**-qualified type

In addition, if it is a structure or union, it does not have any member (including, recursively, any member of all contained structures or unions) with a **const**-qualified type.

The name “lvalue” comes from the assignment expression $E1 = E2$ in which the left operand *E1* must be an lvalue expression.

Primary expressions

- Identifiers, constants, string literals, and parenthesized expressions are primary expressions.
- An identifier is a primary expression, provided it has been declared as designating an object (which makes it an lvalue) or a function (which makes it a function designator).
- A constant is a primary expression; its type depends on its form and value.
- A string literal is a primary expression; it is an lvalue.

- A parenthesized expression is a primary expression. Its type and value are identical to those of the un-parenthesized version. It is an lvalue, a function designator, or a void expression, according to the type of the un-parenthesized expression.

Operators

A table of operator associativity and precedence appears in the next section (page 116).

Unary operators

Expressions with unary operators group right to left.

- | | |
|-----------------------|---|
| * <i>e</i> | Indirection operator. Returns the object or function pointed to by its operand. If the type of the expression is “pointer to ...,” the type of the result is “...” |
| & <i>e</i> | Address operator. Returns a pointer to the object or function referred to by the operand. Operand must be an lvalue or function type, and not a bit-field or an object declared register . Where the operand has type “ <i>type</i> ,” the result has type “pointer to <i>type</i> .” |
| - <i>e</i> | Negation operator. The operand must have arithmetic type. Result is the negative of its operand. Integral promotion is performed on the operand, and the result has the promoted type. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the result type. |
| + <i>e</i> | Unary plus operator. The operand must have arithmetic type. Result is the value of its operand. Integral promotion is performed on the operand, and the result has the promoted type. |
| ! <i>e</i> | Logical negation operator. The operand must have arithmetic or pointer type. Result is one if the value of its operand is zero, zero if the value of its operand is non-zero. The type of the result is int . |
| ~ <i>e</i> | The ~ operator yields the one’s complement (all bits inverted) of its operand, which must have integral type. Integral promotion is performed on the operand, and the result has the promoted type. |
| ++ <i>e</i> | The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression ++ <i>x</i> is equivalent to <i>x</i> += 1. The type of the result is the type of the operand. |
| -- <i>e</i> | The modifiable lvalue operand of prefix -- is decremented analogously to the prefix ++ operator. |

- e ++*** When postfix ++ is applied to a modifiable lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue.
- e --*** When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the same manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue.
- sizeof e*** The **sizeof** operator yields the size in bytes of its operand. When applied to an object with array type, the result is the total number of bytes in the array. (The size is determined from the declarations of the objects in the expression.) This expression is semantically an **unsigned** constant (of type **size_t**, a typedef) and may be used anywhere a constant is required (except in a **#if** preprocessing directive line). One major use is in communication with routines like storage allocators and I/O systems.

sizeof (type)

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

Cast operators — explicit conversions

- (type) e*** Placing a parenthesized type name before an expression converts the value of the expression to that type. Both the operand and ***type*** must be pointer type or an arithmetic type.

Multiplicative operators

The multiplicative operators *****, **/**, and **%** group left to right. The usual arithmetic conversions are performed, and that is the type of the result.

- e * e*** Multiplication operator. The ***** operator is commutative.
- e / e*** Division operator. When positive integers are divided, truncation is toward 0. If either operand is negative, the quotient is negative. Operands must be arithmetic types.
- e % e*** Remainder operator. Yields the remainder from the division of the first expression by the second. The operands must have integral type. The sign of the remainder is that of the first operand. It is always true that $(a/b)*b + a\%b$ is equal to a (if a/b is representable).

Additive operators

The additive operators `+` and `-` group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

e + e Result is the sum of the operands. A pointer to an object in an array and an integral value may be added. The latter is always converted to an address offset by multiplying it by the size of the object to which the pointer points. The result is a pointer of the same type as the original pointer that points to another object in the same array, appropriately offset from the original object. Thus if *P* is a pointer to an object in an array, the expression *P*+1 is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The `+` operator is commutative.

The valid operand type combinations for the `+` operator are:

a + a

p + i or *i + p*

where *a* is an arithmetic type, *i* is an integral type, and *p* is a pointer.

e - e Result is the difference of the operands. The operand combinations are the same as for the `+` operator, except that a pointer type may not be subtracted from an integral type.

Also, if two pointers to objects of the same type are subtracted, the result is converted (by division by the size of the object) to an integer that represents the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, because pointers, even to objects of the same type, do not necessarily differ by a multiple of the object size. The result type is **`ptrdiff_t`** (defined in *stddef.h*). **`ptrdiff_t`** is a typedef for **`int`** in this implementation. It should be used "as is" to ensure portability. Valid type combinations are

a - a

p - i

p - p

Bitwise shift operators

The bitwise shift operators << and >> take integral operands.

e1 << *e2* Shifts *e1* left by *e2* bit positions. Vacated bits are filled with zeros.

e1 >> *e2* Shifts *e1* right by *e2* bit positions. Vacated bits are filled with zeros on the 3B2. On the Intel386™ microprocessor, vacated bits are filled with zeros if the promoted type of *e1* is an unsigned type. Otherwise they are filled with copies of the sign bit of the promoted value of *e1*.

The result types of the bitwise shift operators are compilation-mode dependent, as follows:

-Xk, -Xt The result type is **unsigned** if either operand is **unsigned**.

-Xa, -Xc, -Xm

The result type is the promoted type of the left operand. Integral promotion occurs before the shift operation.

Relational operators

a relop a

p relop p

- The relational operators < (less than) > (greater than) <= (less than or equal to) >= (greater than or equal to) yield "1" if the specified relation is true and "0" if it is false.
- The result has type **int**.
- Both operands:
 - have arithmetic type; or
 - are pointers to qualified or unqualified versions of the same object or incomplete types.

Equality operators

a eqop a

p eqop p

p eqop 0

0 eqop p

- The == (equal to) and != (not equal to) operators are analogous to the relational operators; however, they have lower precedence.

Bitwise AND operator $ie1 \& ie2$

- Bitwise “and” of *ie1* and *ie2*.
- Value contains a “1” in each bit position where both *ie1* and *ie2* contain a “1”, and a “0” in every other position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Bitwise exclusive OR operator $ie1 \wedge ie2$

- Bitwise exclusive “or” of *ie1* and *ie2*.
- Value contains a “1” in each position where there is a “1” in either *ie1* or *ie2*, but not both, and a “0” in every other bit position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Bitwise OR operator $ie1 | ie2$

- Bitwise inclusive “or” of *ie1* and *ie2*.
- Value contains a “1” in each bit position where there is a “1” in either *ie1* or *ie2*, and a “0” in every other bit position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Logical AND operator $e1 \&\& e2$

- Logical “and” of *e1* and *e2*.
- *e1* and *e2* must be scalars.
- *e1* is evaluated first, and *e2* is evaluated only if *e1* is non-zero.
- Result is “1” if both *e1* and *e2* are non-zero, otherwise “0”.
- Result type is `int`.

Logical OR operator

$e1 \ || \ e2$

- Logical “or” of $e1$ and $e2$.
- $e1$ and $e2$ must be scalars.
- $e1$ is evaluated first, and $e2$ is evaluated only if $e1$ is zero. Result is 0 only if both $e1$ and $e2$ are false, otherwise 1.
- Result type is **int**.

Conditional operator

$e \ ? \ e1 \ : \ e2$

- If e is non-zero, then $e1$ is evaluated; otherwise $e2$ is evaluated. The value is $e1$ or $e2$.
- The first operand must have scalar type.
- For the second and third operands, one of the following must be true:
 - Both must be arithmetic types. The usual arithmetic conversions are performed to make them a common type and the result has that type.
 - Both must have compatible structure or union type; the result is that type.
 - Both operands have **void** type; the result has **void** type.
 - Both operands are pointers to qualified or unqualified versions of compatible types. The result type is the composite type.
 - One operand is a pointer and the other is a null pointer constant. The result type is the pointer type.
 - One operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**. The result type is a pointer to **void**.

For the pointer cases (the last three), the result is a pointer to a type qualified by all the qualifiers of the types pointed to by the operands.

Assignment expressions

- Assignment operators are: `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `|=` `^=`
- An expression of the form `e1 op= e2` is equivalent to `e1 = e1 op (e2)` except that `e1` is evaluated only once.
- The left operand:
 - must be a modifiable lvalue.
 - must have arithmetic type, or, for `+=` and `-=`, must be a pointer to an object type and the right operand must have integral type.
 - of an `=` operator, if the operand is a structure or union, must not have any member or submember qualified with **const**.
- Result type is the type of the (unpromoted) left operand.

Comma operator

`e1 , e2`

- `e1` is evaluated first, then `e2`.
- The result has the type and value of `e2` and is not an lvalue.

Structure operators

`su . mem`

Indicates member *mem* of structure or union **su**.

`sup -> mem`

Indicates member *mem* of structure or union pointed to by *sup*. Equivalent to `(*sup).mem`.

Associativity and precedence of operators

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
= = !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	right to left
= + = - = * = / = % = & = ^ = = < <= > >=	right to left
,	left to right

Unary +, -, and * have higher precedence than their binary versions.

Prefix ++ and -- have higher precedence than their postfix versions.

Constant expressions

- A constant expression is evaluated during compilation (rather than at run time). As a result, a constant expression may be used any place that a constant is required.
- Constant expressions must not contain assignment, ++, --, function-call, or comma operators, except when they appear within the operand of a **sizeof** operator.

Initialization

- Scalars (all arithmetic types and pointers):

Scalar types with static or automatic storage duration are initialized with a single expression, optionally enclosed in braces. Example:

```
int i = 1;
```

Additionally, scalar types (with automatic storage duration only) may be initialized with a nonconstant expression.

- Unions:

An initializer for a union with static storage duration must be enclosed in braces, and it initializes the first member in the declaration list of the union. The initializer must have a type that can be converted to the type of the first union member. Example:

```
union {
    int i;
    float f;
} u = {1};    /* initialize u.i */
```

For a union with automatic storage duration, if the initializer is enclosed in braces, it must consist of constant expressions that initialize the first member of the union. If the initializer is not enclosed in braces, it must be an expression that has the matching union type.

- Structures:

The members of a structure may be initialized by initializers that can be converted to the type of the corresponding member.

```
struct s {
    int i;
    char c;
    char *s;
} st = { 3, 'a', "abc" };
```

This example illustrates initialization of all three members of the structure. If initialization values are missing, as in `struct s st2 = {5};` then the first member is initialized (in this case, member `i` is initialized with a value of “5”), and any uninitialized member is initialized with “0” for arithmetic types and a null pointer constant for pointer types.

For a structure with automatic storage duration, if the initializer is enclosed in braces, it must consist of constant expressions that initialize the respective members of the structure. If the initializer is not enclosed in braces, it must be an expression that has the matching structure type.

- Arrays:

The number of initializers for an array must not exceed the dimension (the declared number of elements), but there may be fewer initializers than the number of elements. When the number of initializers is less than the size of the array, the first array elements are initialized with the values given, until the supply of initializers is exhausted. Any remaining array elements are initialized with the value 0 or a null pointer constant, as explained above in the discussion of structures. Example:

```
int ia[5] = { 1, 2 };
```


In this example, an array of five `ints` is declared, but only the first two members are initialized explicitly. The first member, `ia[0]`, is initialized with a value of “1”; the second member, `ia[1]`, is initialized with a value of “2”. The remaining members are initialized with a value of “0”.

When no dimensions are given, the array is sized to hold exactly the number of initializers supplied.

A character array may be initialized with a string literal, as in:

```
char ca[] = { "abc" }; /*curly braces are optional*/
```

where the size of the array is four (three characters with a null byte appended). The following:

```
char cb[3] = "abc";
```

is valid; however, in this case the null byte is discarded. But:

```
char cc[2] = "abc";
```

is erroneous because there are more initializers than the array can hold.

Arrays may be initialized similarly with wide characters:

```
wchar_t wc[] = L"abc";
```

Initializing subaggregates (for example, arrays of arrays) requires the proper placement of braces. For example:

```
int ia [4][2] =
{
    1,
    2,
    3,
    4
};
```

initializes the first two rows of `ia` (`ia[0][0]`, `ia[0][1]`, `ia[1][0]`, and `ia[1][1]`), and initializes the rest to “0”. This is a “minimally bracketed” initialization.

Note that a similar “fully bracketed” initialization yields a different result:

```
int ia [4][2] =
{
    {1},
    {2},
    {3},
    {4},
};
```

initializes the first column of `ia` (`ia[0][0]`, `ia[1][0]`, `ia[2][0]`, and `ia[3][0]`), and initializes the rest to “0”.

Mixing the fully and minimally bracketed styles may lead to unexpected results. Use one style or the other consistently.

Statements

A statement specifies a task to be performed. There are five types of statements:

- Expression statement
- Compound statement
- Selection statements
- Iteration statements
- Jump statements

Expression statement

expression ;

The *expression* is executed for its side effects, if any (such as assignment or function call).

Compound statement

```
{
    declaration-listopt
    statement-listopt
}
```

- Delimited by { and }.
- May have a list of *declarations*.
- May have a list of *statements*.
- May be used wherever *statement* appears below.

Selection statements

if

```
if (expression)
    statement
```

- If *expression* evaluates to non-zero (true), *statement* is executed.
- If *expression* evaluates to 0 (false), control passes to the statement following *statement*.
- The *expression* must have scalar type.

else

```

    if (expression1)
        statement1
    else if (expression2)
        statement2
    else
        statement3

```

- If *expression1* is true, *statement1* is executed, and control passes to the statement following *statement3*. Otherwise, *expression2* is evaluated.
- If *expression2* is true, *statement2* is executed, and control passes to the statement following *statement3*. Otherwise, *statement3* is executed, and control passes to the statement following *statement3*.
- An **else** is associated with the lexically nearest **if** that has no **else** and that is at the same block level.

switch

```

    switch (expression)
        statement

```

- Control jumps to or past *statement* depending on the value of *expression*.
- *expression* must have integral type.
- Any optional **case** is labeled by an integral constant expression.
- If a **default** case is present, it is executed if no other case match is found.
- If no **case** matches, including **default**, control goes to the statement following *statement*.
- If the code associated with a **case** is executed, control falls through to the next **case** unless a **break** statement is included.
- Each case of a switch must have a unique constant value after conversion to the type of the controlling expression.

In practice, *statement* is usually a compound statement with multiple cases, and possibly a **default**; the description above shows the minimum usage. In the following example, **flag** gets set to 1 if **i** is 1 or 3, and to 0 otherwise:

```

switch (i) {
case 1:
case 3:
    flag = 1;
    break;
default:
    flag = 0;
}

```

Iteration statements

while

```

while (expression)
    statement

```

This sequence is followed repetitively:

- *expression* is evaluated.
- If *expression* is non-zero, *statement* is executed.
- If *expression* is zero, *statement* is not executed, and the repetition stops.

expression must have scalar type.

do-while

```

do
    statement
while (expression);

```

This sequence is followed repetitively:

- *statement* is executed.
- *expression* is evaluated.
- If *expression* is zero, repetition stops.

(**do-while** tests loop at the bottom; **while** tests loop at the top.)

for

```

for (expression1; expression2; expression3)
    statement

```

- *expression1* initializes the loop.
- *expression2* is tested before each iteration.

- If *expression2* is true:
 - *statement* is executed.
 - *expression3* is evaluated.
 - Loop until *expression2* is false (zero).
- Any of *expression1*, *expression2*, or *expression3* may be omitted, but not the semicolons.
- *expression1* and *expression3* may have any type; *expression2* must have scalar type.

Jump statements

goto

```
goto identifier;
```

- Goes unconditionally to statement labeled with *identifier*.
- Statement is labeled with an identifier followed by a colon, as in:

```
A2: x = 5;
```

- Useful to break out of nested control flow statements.
- Can only jump within the current function.

break

Terminates nearest enclosing **switch**, **while**, **do**, or **for** statement. Passes control to the statement following the terminated statement. Example:

```
for (i=0; i<n; i++) {  
    if ((a[i] = b[i]) == 0)  
        break;    /* exit for */  
}
```

continue

Goes to top of smallest enclosing **while**, **do**, or **for** statement, causing it to reevaluate the controlling expression. A **for** loop's *expression3* is evaluated before the controlling expression. It can be thought of as the opposite of the **break** statement. Example:

```

for (i=0; i<n; i++) {
    if (a[i] != 0)
        continue;
    a[i] = b[i];
    k++;
}

```

return

```

return;
return expression;

```

- **return** by itself exits a function.
- **return *expression*** exits a function and returns the value of *expression*. For example:

```
return a + b;
```

Portability considerations

Certain parts of C are inherently machine dependent. The following potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of variables declared with **register** that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified. For example, in the expression:

```
a[i] = b[i++]
```

the value of **i** could be incremented after **b[i]** is fetched, but before **a[i]** is evaluated and assigned to, or it could be incremented after the assignment.

The value of a multi-character character constant may be different for different machines.

Fields are assigned to words, and characters to integers, right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (for example, by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally imposed storage layouts.

The **lint** program checking tool is useful for finding program bugs and non-portable constructs. For information on how to use **lint**(CP), see Chapter 5, “lint analyzer” (page 153).

Chapter 4

COFF link editor

This chapter describes the link editor, **ld**(CP), and the command language used to direct operations of the COFF link editor.

ld creates a program by combining object files, performing relocations, and resolving external symbols. The input files are normally relocatable object files, or library files containing relocatable object files which have been created by **cc**(CP) or by previous invocations of **ld**. The output file created by **ld** may be a relocatable object file or an absolute (executable) object file.

Two object file formats, COFF (Common Object File Format) and ELF (Executable and Linking Format), are supported by this release of the link editor. The command language supported by the COFF version of **ld** allows the user a greater degree of control over object files and their memory locations. A similar facility, the **mapfile** option, is provided for the ELF linker. See Chapter 10, “**Mapfile option**” in *Developer’s Topics* for details on how to change the default mapping provided by the ELF linker.

If a particular file format is not explicitly specified through the **-b** option, **ld** scans all the user specified object files and user provided libraries (if any) that are to be linked. It determines whether an object file is in COFF or ELF format and executes the appropriate binaries in each case. See the **ld**(CP) man page for more information.

The discussion that follows relies heavily on the reader’s knowledge of the COFF file format. In particular, familiarity with the concept of a **section** is essential to understanding this material. See Chapter 7, “**Common Object File Format (COFF)**” in *Developer’s Topics* for a general discussion of the file format.

NOTE Although the basic concepts are similar, the ELF file format is different from the COFF file format. The command language for the COFF linker is also different from the mapfile directives used for the ELF linker. Most of the examples provided here do not apply to ELF binaries. See Chapter 8, “ELF object files” and Chapter 10, “Mapfile option” in *Developer’s Topics* for general discussion of the ELF file format and how to write mapfiles for the ELF link editor.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections in an **ld** input object file are called *input sections*; sections in a file created by **ld** are called *output sections*. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. The link editor performs allocation, that is, the setting aside of some portion of the virtual address space for a section. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Memory configuration

The virtual memory of the target machine is, for allocation purposes, partitioned into configured and unconfigured memory. By default, **ld** treats all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. In this case, addresses in the range 3K to 20K-1 are not configured. Unconfigured memory is treated as reserved or unusable by **ld**.

Unless otherwise specified, all discussions of memory, addresses, and so forth, refer to the configured sections of the address space.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific address. Specifying this starting address is called binding, and the section in question is said to be “bound to” or “bound at” the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with a command language assignment statement.

Object files

Object files are produced both by the assembler **as**(CP) (may be called by the compiler internally) and by **ld**(CP). **ld** accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the object files given to **ld** can also be executable files.

Files produced by compilation may contain sections called `.text`, `.data`, `.bss`, `.init` and `.fini`. The `.text` section contains the instruction text (executable instructions), `.data` contains initialized data variables, and `.bss` contains uninitialized data variables. Consider the following fragment of C code:

```
1      char abc[200];
2      int i = 100;
3
4      i = 0;
```

The uninitialized variable **abc** (line 1) is located in `.bss`, the initialized variable **i** (line 2) is located in `.data`, compiled code from the C assignment (line 4) is stored in `.text`. The `.init` section, if it exists, contains initialization code used in static shared libraries. The `.fini` section, if it exists, contains finalization code used in static shared libraries. Sections with names other than these may be present in object files created by **ld**.

Link editor command language

This section gives information on the command language for the COFF linker. The command language enables you to:

- specify the memory configuration of the target machine
- direct how **ld** combines the sections of an object file
- bind sections to specific addresses or within specific portions of memory
- define or redefine global symbols

Under normal circumstances there is no need for such tight control over object files and their memory locations. When you do need to have precise control over link editor output, you do it with the command language. You

must be very careful when using the command language, because it is easy to create an executable that will not run, or will crash the system.

The link editor is called using the command:

ld [options] filename1 filename2 ...

Link editor command language directives are passed in a file named on the command line. Such a file is often called an **ifile** or **mapfile**. Any file named on the command line that is not identifiable (from the magic number) as an object module or an archive library is assumed to contain link editor directives. However, to use such an **ifile**, you must link the object files statically.

Expressions

Expressions in the COFF link editor command language may contain global symbols, constants, and most of the basic C language operators. (See “Syntax diagram for input directives” (page 149).) Constants are the same as in C, with a number recognized as hexadecimal if it starts with ‘0x’, as octal if it starts with ‘0’, and as decimal otherwise. All numbers are treated as long integers. Symbol names may contain uppercase or lowercase letters, digits, and the underscore character (_). Symbols within an expression have the value of the *address* of the symbol only. **ld** does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, and so on.

Certain names are reserved and hence unavailable as symbol or section names. This impacts your C source program. A common error is to use **ifiles** when your C program contains any of the reserved names (for example, **length**). The reserved names include:

ADDR	BLOCK	GROUP	NEXT	RANGE	SPARE
ALIGN	COMMON	INFO	NOLOAD	REGIONS	PHY
ASSIGN	COPY	LENGTH	ORIGIN	SECTIONS	TV
BIND	DSECT	MEMORY	OVERLAY	SIZEOF	
addr	block	length	origin	sizeof	
align	group	next	phy	spare	
assign	l	o	range		
bind	len	org	s		

The supported operators are shown in the following table, in order of precedence from highest to lowest:

Table 4-1 Operator Symbols

symbol
! ~ - (UNARY Minus)
* / %
+ - (BINARY Minus)
>> <<
== != > < <= >=
&
&&
= += -= *= /=

The operators have the same meaning as in the C language. Operators on the same line in the table have the same precedence.

Assignment statements

External symbols are globally visible functions or data items. They may be defined and assigned addresses by an assignment statement in a file of COFF **ld** directives.

Syntax

The syntax of the assignment statement is:

symbol op expression;

op may be one of the operators "=", "+=", "-=", "*=", or "/=". Assignment statements must be terminated by a semicolon.

Interaction with source code

Assignments statements in an **ifile** may refer to symbols defined in the object files being linked. These statements cannot directly affect the value of the symbols in the object file, but they can affect the addresses of these symbols.

The following is an example of how assignment statements interact with the source code. Consider the C source file:

```
int i = 100;
int k = 22;
int j;
main() {
    printf("Value of j is %d\n");
}
```

Suppose that the object file corresponding to this program is linked with the following **ifile**:

```
j=i+4;
```

This directive causes the external symbol **j** to be assigned the address of **i** plus 4 bytes. This is the same address as external symbol **k**. The value found at this address will be the integer 22. The action of the C program will be to print '22'.

All assignment statements (with one exception, described in the next paragraph) are evaluated after allocation has been performed. This occurs after all symbols defined in the input file are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbols address in the output object file. References within text and data to symbols given a value through an assignment statement access this latest assigned value. Assignment statements are processed in the order in which they are input to **ld**.

The location counter symbol

Assignment statements are normally placed outside the scope of section definition directives (see "Section definition directives" (page 132)). However, there is a special symbol, the dot (**.**), that can occur only within a section definition directive. This symbol refers to the current address of **ld**'s location counter (a pointer to the next memory address which may be allocated). Assignment expressions involving **"."** are evaluated during the allocation phase of **ld**. Assigning a value to **"."** within a section definition directive can increment (but not decrement) the location counter and create holes within the section, as described in "Section definition directives" (page 132). Assigning the value of **"."** to a symbol permits the final allocated address of a section to be saved.

Pseudo-functions

The pseudo-function **ALIGN** is used to increment the location counter so as to align a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. This is done with the expression:

```
ALIGN(n)
```

This function call is equivalent to:

```
. = (. + n - 1) & ~(n - 1)
```

SIZEOF and **ADDR** are pseudo-functions that, given the name of a section, return the size or address of the section, respectively. These may be used in assignment statements.

Types of expression values

Link editor expressions may have either an absolute or a relocatable value. When **ld** creates a symbol through an assignment statement, the symbol's value is the same type absolute or relocatable, as the expression. That type depends on the following rules:

- an expression with a single relocatable symbol and zero or more constants or absolute symbols is relocatable
- the difference of two relocatable symbols from the same section is absolute
- all other expressions are combinations of the above

Specifying a memory configuration

MEMORY directives are used to specify:

- the total size of the virtual space of the target machine
- the configured and unconfigured areas of the virtual space

If no directives are supplied, **ld** assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

MEMORY directives are used to assign a name to a virtual address range. This range is described by specifying its origin and length. Output sections can then be bound to virtual addresses within specifically named memory areas. Memory names can be up to 8 characters long and may contain upper-case or lowercase letters, digits, and the characters "\$", ".", and "_". Names of memory ranges are used only by **ld** and are not carried in the output file symbol table or headers.

The syntax of the **MEMORY** directive is as follows:

```
MEMORY
{
    name1 (attr) : origin = n1, length = n2
    name2 (attr) : origin = n3, length = n4
    .
    .
    .
}
```

For example:

```
MEMORY
{
    mem1:          origin=0x000000,   length=0x10000
    mem2 (RW):     origin=0x020000,   length=0x40000
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above example. The **origin** operand refers to the virtual address of the memory range. The origin and length are entered as long integer constants in either decimal,

octal, or hexadecimal. The **origin** and **length** specifications, as well as individual **MEMORY** directives, may be separated by white space or a comma.

Attributes may be associated with a named memory area. These attributes follow the memory area name and are enclosed in parentheses. The attributes that may be specified are:

- R - readable memory
- W - writable memory
- X - executable (instructions may reside in this memory)
- I - initializable (stack areas are typically not initialized)

If no attributes are specified on a **MEMORY** directive or if no **MEMORY** directives are supplied, memory areas assume all of the attributes R, W, X, and I.

By way of **MEMORY** directives, **ld**(CP) can be told that memory is configured in some manner other than the default. If **MEMORY** directives are used, all virtual memory not described in some **MEMORY** directive is considered to be unconfigured. Unconfigured memory is not used in **ld**'s allocation process: nothing except **DSECT** sections (discussed later) can be link edited or bound to an address within unconfigured memory. For example, it may be necessary to prevent anything from being linked to the first 0x10000 words of memory. This can be accomplished by using **MEMORY** directives that do not mention that portion of memory.

Section definition directives

The purpose of the **SECTIONS** directive is to describe how input sections in COFF files are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no **SECTIONS** directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections **s1** and **s2** and the other containing sections **s2** and **s3**, the output object file contains three output sections **s1**, **s2**, and **s3**. The input sections **s1** and **s3** appear in output sections of the same name; the two input sections named **s2** appear in an output section named **s2**. The order of these output sections depends on the order in which the link editor sees the input files.

The basic syntax of the **SECTIONS** directive is:

```
SECTIONS
{
    secname1 :
    {
        file_specifications,
        assignment_statements
    }
    secname2 :
    {
        file_specifications,
        assignment_statements
    }
    .
    .
    .
}
```

File specifications

Within a section definition (a **SECTIONS** directive), the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified in a statement of the form:

```
filename ( secname )
```

or:

```
filename ( secnam1 secnam2 . . . )
```

White space or commas are used to separate file specifications and to separate input section names within file specifications.

The following is an example of a **SECTIONS** directive:

```
SECTIONS
{
    outsec1:
    {
        file1.o (sec1)
        file2.o
        file3.o (sec1, sec2)
    }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be:

1. section **sec1** from file *file1.o*
2. all sections from *file2.o*, in the order they appear in the input file
3. section **sec1** from file *file3.o*, and then section **sec2** from file *file3.o*

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in *file1.o* or *file3.o*, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

To refer to all the uninitialized, unallocated global symbols in a file, the following statement may be used in a file specification:

```
filename [COMMON]
```

If a filename appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section.

The following code may be used in a file specification to refer to all previously unallocated input sections of the given name, regardless of what input file they are contained in:

```
*(secname)
```

Loading a section at a specified address

To bind an output section to a specific virtual address, use a **SECTIONS** directive of the following form:

```
SECTIONS
{
    outsec addr:
    {
        . . .
    }
    .
    .
    .
}
```

The value *addr* is a C constant which specifies the binding address. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld** issues an appropriate error message. *addr* may also be the word **BIND**, followed by a parenthesized expression. The expression may use the pseudo-functions **SIZEOF**, **ADDR**, or **NEXT**. **NEXT** accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; **SIZEOF** and **ADDR** accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The **SECTIONS** directives defining output sections need not be given to **ld** in any particular order, unless **SIZEOF** or **ADDR** is used.

ld does not ensure that the size of each section consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4.

The **ld** directives can be used to force a section to start on an odd byte boundary, although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld** issues a warning message.

Aligning an output section

An output section may be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. This may be done in order to take advantage of the underlying architecture. For example, it may be possible to reduce the number of instructions necessary to address a data object by aligning the data object on a word boundary. This is performed using an **ALIGN** in a **SECTIONS** directive. For example:

```
SECTIONS
{
    outsec ALIGN(0x20000) :
    {
        . . .
    }
    .
    .
    .
}
```

The output section **outsec** is not bound to any specific address but is placed at some virtual address that is a multiple of 0x20000.

Grouping sections together

The default allocation algorithm for COFF **ld** does the following:

- Links all input **.init** sections followed by **.text** sections into one output section. This output section is called **.text** and is bound to the address 0x0 plus the size of all headers in the output file.
- Links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine-dependent constant plus a number dependent on the size of headers and text.
- Links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**.

If any **SECTIONS** directives are specified, they replace the default allocation algorithm. Rather than relying on the **ld** default algorithm when manipulating COFF files, the one certain way to determine address and order information is to take it from the file and section headers. The default allocation of **ld** is equivalent to the following directive, where *align_value* and *sizeof_headers* are machine-dependent constants:

```
SECTIONS
{
    .text sizeof_headers : { *(.init) *(.text) *(.fini)}
    GROUP BIND( NEXT(align_value) +
    ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
    {
        .data      : { }
        .bss       : { }
    }
}
```

The **GROUP** command ensures that the two output sections **.data** and **.bss** are allocated together. Binding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

For compatibility with UNIX System V Release 2, the addresses of these sections cannot change. Unfortunately, **.init** sections in the algorithm above will interfere with the placement of the signal recovery routines. Hence the **.text** sections are linked into the *a.out* **.text** section first. The **.init** sections (for shared libraries) and the **.fini** sections follow all of the **.text** sections. Routines in *crt1.o* (a C runtime startup routine) branch to the **.init** sections before calling the **main()** function of the program.

The following **SECTIONS** directive may be used to place **.text**, **.data**, and **.bss** in the same segment of memory,

```
SECTIONS
{
    GROUP
    {
        .text      : { }
        .data      : { }
        .bss       : { }
    }
}
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive in the above example. To bind the group to 0xC0000, add the address after the GROUP keyword:

```
GROUP 0xC0000 : {
    ...
}
```

This change causes the output section `.text` to be bound at 0xC0000, followed by the remaining members of the group in order of their appearance. To align the group to 0x10000, add an ALIGN after the GROUP keyword:

```
GROUP ALIGN(0x10000) : {
    ....
}
```

This change will causes the output section `.text` to be aligned to 0x10000, followed by the remaining members of the group.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
    .text    : { }
    .data ALIGN(0x400000) : { }
    .bss     : { }
}
```

In this example, the `.text` section starts at virtual address 0x0 (provided that address is in configured memory) and the `.data` section starts at a virtual address aligned to 0x400000. The `.bss` section immediately follows the `.text` section if there is enough space. If not, it follows the `.data` section. The order in which output sections are defined to `ld` cannot be used to force a certain allocation order in the output file.

Creating holes within output sections

The special symbol dot ("`.`"), representing `ld`'s location counter, appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, "`.`" causes the location counter to be reset, leaving a hole in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character. The default fill character is 0x00; alternately, the user may supply a fill character. See the discussion of filling holes in "Initialized section holes or `.bss` sections" (page 140).

Consider the following section definition:

```

1 outsec:
2 {
3     . += 0x1000;
4     f1.o (.text)
5     . += 0x100;
6     f2.o (.text)
7     . = align (4);
8     f3.o (.text)
9 }
```

The effect of this command is as follows:

- line 3 increments the location counter by 0x1000, thereby leaving a 0x1000 byte hole, filled with the default fill character, at the beginning of the section
- in line 4, the **.text** section of input file *f1.o* is linked after the hole that was just left
- in line 5, the location counter is incremented by 0x100, leaving a 0x100 byte hole filled with the default fill character
- in line 6, the **.text** section of input file *f2.o* is linked following the second hole; this section begins 0x100 bytes from the end of **f1.o (.text)**.
- line 7 causes the location counter to be aligned with the next 4-byte boundary (that is, the next double-word boundary)
- in line 8, the **.text** section of **f3.o** is linked; the effect of lines 7 and 8 is to cause this section to start at the next full word boundary following the **.text** section of *f2.o*. The boundary is determined relative to the beginning of the output section **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld** treats the output section as if it began at address zero. As a result, in the above example, if **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address, even though **f3.o (.text)** is aligned to a full word boundary. This may be prevented by specifying an alignment for the entire output section, as follows:

```

outsec ALIGN(4) : {
    ...
}
```

Expressions that decrement “.” are illegal. Subtracting a value from the location counter is not allowed, since this can cause memory to be overwritten.

Creating and defining symbols at link-edit time

Assignment statements can be used to give symbols a value that is link-edit dependent. For example, we just saw that the “.” symbol can be used to adjust the location counter during allocation. It is possible to assign allocation-dependent values to other symbols. These can be symbols that

were defined in an object file that is being linked, or they may be symbols that are used only in the **ifile**. This provides a way to assign to symbols addresses known only after allocation. For example:

```
SECTIONS
{
    outsc1: {...}
    outsc2:
    {
        file1.o (s1)
        s2_start = . ;
        file2.o (s2)
        s2_end = . - 1;
    }
}
```

The symbol **s2_start** is defined to be the address of **file2.o(s2)**, and **s2_end** is the address of the last byte of **file2.o(s2)**. Consider the following example:

```
SECTIONS
{
    outsc1:
    {
        file1.o (.data)
        mark = .;
        . += 4;
        file2.o (.data)
    }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of *file1.o*'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving **"."** must appear within **SECTIONS** definitions since they are evaluated during allocation. Assignment instructions that do not involve **"."** can appear within **SECTIONS** definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. **ld** issues warning messages for each defined symbol that is being redefined within an **ifile**. However, it is safe to assign of absolute values to new symbols because there are no references or initialized data associated with these symbols.

Allocating a section into named memory

It is possible to specify that a section be linked somewhere within a named memory range, previously defined on a MEMORY directive.

For example:

```
MEMORY
{
    mem1:          o=0x000000    l=0x10000
    mem2 (RW):     o=0x020000    l=0x40000
    mem3 (RW):     o=0x070000    l=0x40000
    mem1:          o=0x120000    l=0x04000
}

SECTIONS
{
    outsec1: { f1.o(.data) } > mem1
    outsec2: { f2.o(.data) } > mem3
}
```

The '**>**' operator (analogous to the UNIX system redirection operator) directs **ld** to place **outsec1** anywhere within the memory area named **mem1** (that is, somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The output section **outsec2** is to be placed somewhere in the area named **mem3**, that is, the address range 0x70000-0xAFFFF.

Initialized section holes or .bss sections

When holes are created within a section, **ld** normally fills them with bytes of zero (0x00). By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor.

SECTIONS directives may be used to initialize such holes or **.bss** output sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. For example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file, or a hole in the text area to be filled with a transfer to an error routine.

An entire output section may be initialized, or specific areas within an output section. However, since no text is generated for an uninitialized **.bss** section,

if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will apply:

- Explicit initialization options may be used to initialize all **.bss** sections in the output section.
- **ld** will use the default fill value to initialize all **.bss** sections in the output section.

Holes are filled using a statement of the form:

```
section_name:
{
    ....
} = long_int
```

or:

```
file_specification = long_int
```

Consider the following **ld** file:

```
SECTIONS
{
    sec1:
    {
        f1.o
        . += 0x200;
        f2.o (.text)
    } = 0xDFFF
    sec2:
    {
        f1.o (.bss)
        f2.o (.bss) = 0x1234
    }
    sec3:
    {
        f3.o (.bss)
        . . .
    } = 0xFFFF
    sec4: { f4.o (.bss) }
```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within **sec3** as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section. When unconfigured areas exist in the virtual memory, each application must assume responsibility for

forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
    mem1:      o = 0x00000      l = 0x02000
    mem2:      o = 0x20000      l = 0x10000
    mem3:      o = 0x40000      l = 0x05000
}
```

Suppose that the files *f1.o*, *f2.o*, . . . *fn.o* each contain three sections **.text**, **.data**, and **.bss**, with the combined **.text** section length being 0x12000 bytes. There is no configured area of memory in which this section can be placed, because the longest memory range that has been defined is only 0x10000 bytes long. Appropriate directives must be supplied to break up the **.text** output section so **ld** may do the allocation. The following set of directives group the **.text** sections from the input files into a number of output sections **txt1**, **txt2**, and so on.

```
SECTIONS
{
    txt1:
    {
        f1.o (.text)
        f2.o (.text)
        f3.o (.text)
    }
    txt2:
    {
        f4.o (.text)
        f5.o (.text)
        f6.o (.text)
    }
    .
    .
    .
}
```

Changing the entry point

The UNIX system *a.out* optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules: the rule used is the first one (from the order given) that applies:

1. The value of the symbol specified with the **-e** option to **ld** is used if present
2. The value of the symbol **_start** is used if present
3. The value of the symbol **main** (defined somewhere in the user program) is used if present
4. The value zero is used

The **start** symbol (the C language startup routine) is usually defined in the *crt*

file found in */lib*. It may be defined in an **ifile** using an assignment statement of the form:

```
_start = expression;
```

If **ld** is called through **cc**, a startup routine is automatically linked in. When the program is executed, this startup routine causes **exit()** to be called after **main()** finishes to close file descriptors and do other cleanup. When calling **ld** directly or when changing the entry point, the user must supply the start-up routine or make sure that the program always calls **exit** rather than falling through the end. Otherwise, the program will dump core.

Using archive libraries

Each member of an archive library is a complete object file. Archive libraries are created with the **ar(CP)** command from object files generated by **cc** or **as**.

Libraries may be searched by the link editor to find definitions for undefined symbols. For a member to be extracted from the library it either must satisfy a reference that is known to be unresolved at the time the library is searched, or has been designated as an undefined symbol through the **-u** option of **ld**. Only those members that are required to resolve existing undefined-symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. When a library member is included by searching the library inside a **SECTIONS** directive, all input sections from the library member are included in the output section being defined by that directive. When a library member is included by searching the library outside of a **SECTIONS** directive, all input sections from the library member are included in an output section with the same name as that input section. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, the following:

- specific members of a library cannot be referenced explicitly in an **ld** directive
- the default rules for the placement of members and sections cannot be overridden when they apply to archive library members

The **-l** option

The **-l** option to **ld** is used to link with an input file coming from a predefined set of directories and having a predefined name. Such files are usually archive libraries, however, they don't have to be. Furthermore, archive libraries can be specified without using the **-l** option by simply giving the full or relative UNIX system file path.

An option of the form **-lx**, where *x* is a string of up to nine characters, will cause a library with filename *libx.a*, located in the directory */lib*, to be searched. The effect of this option is the same as if the library file name was substituted for the option on the command line.

The ordering of archive libraries is important since the library must be searched after the unresolved symbol reference is encountered. Archive libraries can be specified more than once, and they are searched every time they are encountered. Archive files have a symbol table at the beginning. If the definition that the library provides for a symbol gives rise to more undefined symbols, **ld** will cycle through the symbol table until it has determined that it cannot resolve any more references from the current library.

Consider the following example involving two input files, *file1.o* and *file2.o*:

- The input files *file1.o* and *file2.o* each contain a reference to the external function **FCN**.
- Input *file1.o* contains a reference to symbol **ABC**.
- Input *file2.o* contains a reference to symbol **XYZ**.
- Library *liba.a*, member 0, contains a definition of **XYZ**.
- Library *libc.a*, member 0, contains a definition of **ABC**.
- Both libraries have a member 1 that defines **FCN**.

The following **ld** command line will cause the **FCN** references to be satisfied by *liba.a*, member 1, **ABC** to be obtained from *libc.a*, member 0, and **XYZ** to remain undefined:

```
ld file1.o -la file2.o -lc
```

When *file1.o* is processed by **ld**, references to **ABC** and **FCN** are encountered. The **-la** option causes *liba.a* to be searched, and the unresolved references are satisfied by definitions in that library. When *file2.o* is processed, references to **FCN** and **XYZ** are encountered. A definition for **FCN** has already been found, and hence this definition (from *liba.a*) resolves the reference. The **-lc** option causes *libc.a* to be searched. The reference to **XYZ** cannot be resolved from *libc.a* because there is no definition of that symbol in the library. There is a definition of **XYZ** in *liba.a* but that definition cannot be obtained at this point, because **ld** did not know at the time it was searching *liba.a* that there would be an unresolved reference to **XYZ**.

The following command line will allow the reference to **XYZ** to be resolved by the definition of that symbol in member 0 of the library *liba.a*. This is possible because the **-la** option appears to the right of the filename *file2.o*. **ld** processes the file *file2.o* and encounters the unresolved reference to **XYZ**; when it

searches the library *liba.a* it knows that it must resolve that reference and hence obtains the definition from the library. The other references are resolved as in the previous example.

```
ld file1.o file2.o -la -lc
```

With the following command line, the FCN references are satisfied by *libc.a*, member 1, ABC is obtained from *libc.a*, member 0, and XYZ is obtained from *liba.a*, member 0.

```
ld file1.o file2.o -lc -la
```

The -u option

The **-u** option is used to force the linking of a library member that contains a definition of a symbol even if none of the source files contain an unresolved reference to that symbol. As is done in the next example, this option may also be used without any source files to extract a member from a library.

The following command line creates an undefined symbol called **rout1** in **ld**'s global symbol table. If any member of library *liba.a* defines this symbol, that member is extracted. Without the **-u** option, there would have been no unresolved references or undefined symbols to cause **ld** to search the archive library. Note that the **-u** option must appear on the command line before the library that you want to link a member from.

```
ld -u rout1 -la
```

Allocation algorithm

An output section is formed as a result of a **SECTIONS** directive, by combining input sections of the same name, or by combining **.text** and **.init** into **.text**. An output section can consist of zero or more input sections. After the composition of an output section is determined, it must then be allocated into configured virtual memory. The COFF link editor uses an algorithm that attempts to minimize fragmentation of memory, and hence increase the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit binding addresses are specified are allocated
2. Any output sections to be included in a specific named memory range are allocated. Each output section is placed into the first available space within *named* memory with any alignment taken into consideration.
3. Output sections not handled by one of the above steps are allocated. Each output section is placed into the first available space within memory with any alignment taken into consideration.

If all memory is contiguous and configured (the default case), and no **SECTIONS** directives are given, then output sections are allocated in the order they appear to **ld**.

Incremental link editing

As previously mentioned, the output of **ld** can be used as an input file to subsequent **ld** runs provided that the relocation information is retained in the initial run by using the **-r** option. With large applications, it may be desirable to partition the C programs into subsystems, link each subsystem independently, and then link edit the entire application.

The first two steps in the following example are described by giving an **ld** command line followed by the file of COFF **ld** directives that was used in that command. The third step is the final link edit.

ld -r -o outfile1 ifile1 infile1.o

```
/* ifile1 */
SECTIONS
{
    ss1:
    {
        f1.o
        f2.o
        ...
        fn.o
    }
}
```

Step 2:

ld -r -o outfile2 ifile2 infile2.o

```
/* ifile2 */
SECTIONS
{
    ss2:
    {
        g1.o
        g2.o
        ...
        gn.o
    }
}
```

Step 3:

ld -o final.out outfile1 outfile2

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

There are two simple rules to follow when applying this technique to incrementally link COFF files:

- Intermediate link edits should have only **SECTIONS** declarations in the link edit command language file, and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.
- All allocation and memory directives, as well as any assignment statements, are included only in the final **ld** run.

DSECT, COPY, NOLOAD, INFO, and OVERLAY sections

Sections in a COFF file may be given a *type* in a section definition, as shown in the following example:

```
SECTIONS
{
    name1 0x200000 (DSECT)      : { file1.o }
    name2 0x400000 (COPY)      : { file2.o }
    name3 0x600000 (NOLOAD)    : { file3.o }
    name4          (INFO)      : { file4.o }
    name5 0x900000 (OVERLAY)   : { file5.o }
}
```

The **DSECT** option creates a *dummy section*. A dummy section has the following properties:

- It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld**.
- It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
- The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. Symbols defined in a DSECT may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched: any members which define such symbols are link edited normally.
- None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from *file1.o* are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the **COPY** option is similar to a dummy section. The difference between a COPY section and a dummy section is that the contents of a COPY section and all associated information is written to the output file.

An **INFO** section is similar to a COPY section: instead of containing valid text and data, as a COPY section does, its purpose is to carry information about the object file. INFO sections are usually used to contain file version identification information.

A section with type **NOLOAD** differs in only one respect from a normal output section: its text or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, and so on.

An **OVERLAY** section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output file blocking

The **BLOCK** option, applied to any output section or **GROUP** directive, is used to direct **ld** to align a section at a specified byte offset in the *output file*. Note that this has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
SECTIONS
{
    .text BLOCK(0x200) : { }
    .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

With this **SECTIONS** directive, **ld** assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of 0x200 (at an offset of 0, 0x200, 0x400, and so forth, in the file).

Nonrelocatable input files

If a file produced by **ld** is intended to be used in a subsequent **ld** run, the first **ld** run should have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an **ld** input file does not have relocation or symbol table information (perhaps from the action of a **strip**(CP) command, or from being link edited without a **-r** option, or with a **-s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (that is, to correctly link all input files, relocate all symbols, resolve unresolved references, and so on), two conditions on the nonrelocatable input files must be met:

- Each input file must have no unresolved external references
- Each input file must be bound to the same virtual address as it was bound to in the **ld** run that created it

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld**.

Syntax diagram for input directives

NOTE Square brackets and curly braces have two meanings in this diagram. Where the actual symbols, **[]** and **{ }**, are used, they are part of the syntax and must be present when the directive is specified.

Where symbols **[** and **]** (larger and in bold) appear, the material enclosed is optional.

Where the symbols **{** and **}** (larger and in bold) appear, multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
<ifile>	{<cmd>}
<cmd>	<memory> <sections> <assignment> <filename> <flags>
<memory>	MEMORY { <memory_spec> { [[] <memory_spec> } }
<memory_spec>	<name> [<attributes>] : <origin_spec> [[] <length_spec>
<attributes>	({ R W X I })
<origin_spec>	<origin> = <long>

(Continued on next page)

(Continued)

Directives	Expanded Directives
<length_spec>	<length> = <long>
<origin>	ORIGIN o org origin
<length>	LENGTH l len length
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> <group> <library>
<group>	GROUP <group_options> : { <section_list> } [<mem_spec>]
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> : { <statement> } [<fill>] [<mem_spec>]
<group_options>	[<addr>] [<align_option>] [<block_option>]
<sec_options>	[<addr>] [<align_option>] [<block_option>] [<type_option>]
<addr>	<long> <bind> (<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN align
<block_option>	<block> (<long>)
<block>	BLOCK block
<type_option>	(DSECT) (NOLOAD) (COPY) (INFO) (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name> > <attributes>
<statement>	<filename> <filename> (<name_list>) [COMMON] * (<name_list>) [COMMON] <assignment> <library> null
<name_list>	<section_name> [,] { <section_name> }
<library>	-l<name>
<bind>	BIND bind

(Continued on next page)

(Continued)

Directives	Expanded Directives
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> .
<assign_op>	= += -= *= /=
<end>	; ,
<expr>	<expr> <binary_op> <expr> <term>
<binary_op>	* / % + - >> << == != > < <= >= & &&
<term>	<long> <name> <align> (<term>) (<expr>) <unary_op> <term> <phy> (<lside>) <sizeof>(<sectionname>) <next>(<long>) <addr>(<sectionname>)
<unary_op>	! -
<phy>	PHY phy
<sizeof>	SIZEOF sizeof
<next>	NEXT next
<addr>	ADDR addr
<flags>	-e<wht_space><name> -f<wht_space><long> -h<wht_space><long> -l<name> -m

(Continued on next page)

(Continued)

Directives	Expanded Directives
	-o<wht_space><filename>
	-r
	-s
	-t
	-u<wht_space><name>
	-z
	-H
	-L<path_name>
	-M
	-N
	-S
	-V
	-VS<wht_space><long>
	-a
	-x
<name>	Any valid symbol name
<long>	Any valid long integer constant
<wht_space>	Blanks, tabs, and newlines
<filename>	Any valid UNIX Operating System file name. This may include a full or partial path name.
<sectionname>	Any valid section name, up to 8 characters
<path_name>	Any valid UNIX Operating System path name (full or partial)

Chapter 5

lint analyzer

The **lint**(CP) analyzer checks for code constructs that may cause your C program not to compile, or to execute with unexpected results. **lint** issues the error and warning messages produced by the C compiler. It also issues “lint-specific” warnings about potential bugs and portability problems.

In particular, lint compensates for separate and independent compilation of files in C by flagging inconsistencies in definition and use across files, including any libraries you have used. In a large project environment especially, where the same function may be used by different programmers in hundreds of separate modules of code, lint can help discover bugs that otherwise might be difficult to find. A function called with one less argument than expected, for example, looks at the stack for a value the call has never pushed, with results correct in one condition, incorrect in another, depending on whatever happens to be in memory at that stack location. By identifying dependencies like this one, and dependencies on machine architecture as well, lint can improve the reliability of code run on your machine or someone else’s.

Options and directives

lint is a static analyzer, which means that it cannot evaluate the run-time consequences of the dependencies it detects. Certain programs may contain hundreds of unreachable **break** statements, and lint will give a warning for each of them. The sheer number of lint messages issued can be distracting. lint, however, provides command line options and directives to help suppress warnings you consider to be spurious.

NOTE Directives are special comments embedded in the source text.

For the example we've cited here,

- You can invoke lint with the **-b** option to suppress all complaints about unreachable **break** statements.
- For a finer-grained control, you can precede any unreachable statement with the comment `/* NOTREACHED */` to suppress the diagnostic for that statement.

"Using lint" (page 159) details options and directives and introduces the lint filter technique, which lets you tailor lint's behavior even more finely to your project's needs. It also shows you how to use lint libraries to check your program for compatibility with the library functions you have called in it.

lint and the compiler

Nearly five hundred diagnostic messages are issued by lint. However, this chapter only describes those lint-specific warnings that are not issued by the compiler. Additionally, this chapter lists diagnostics issued both by lint and the compiler that are capable of being suppressed only by lint options. For the text and examples of all messages issued exclusively by lint or subject exclusively to its options, refer to "Understanding lint-specific messages" (page 165). For the messages also issued by the compiler, consult Appendix B, "C compiler diagnostics" (page 515).

Message formats

Most of lint's messages are simple, one-line statements printed for each occurrence of the problem they diagnose. Errors detected in included files are reported multiple times by the compiler but only once by lint, no matter how many times the file is included in other source files. Compound messages are issued for inconsistencies across files and, in a few cases, for problems within them as well. A single message describes every occurrence of the problem in the file or files being checked. When use of a lint filter requires that a message be printed for each occurrence, compound diagnostics can be converted to the simple type by invoking lint with the **-s** option.

NOTE For more information see "Using lint" (page 159).

What lint does

lint-specific diagnostics are issued for three broad categories of conditions: inconsistent use, nonportable code, and suspicious constructs. In this section, we'll review examples of lint's behavior in each of these areas, and suggest possible responses to the issues they raise.

Consistency checks

Inconsistent use of variables, arguments, and functions is checked within files as well as across them. Generally speaking, the same checks are performed for prototype uses, declarations, and parameters as for old-style functions. (If your program does not use function prototypes, lint will check the number and types of parameters in each call to a function more strictly than the compiler.) lint also identifies mismatches of conversion specifications and arguments in `[fs]printf()` and `[fs]scanf()` control strings. Examples:

- Within files, lint flags non **void** functions that “fall off the bottom” without returning a value to the invoking function. In the past, programmers often indicated that a function was not meant to return a value by omitting the return type: `fun() {}`. That convention means nothing to the compiler, which regards `fun()` as having the return type **int**. Declare the function with the return type **void** to eliminate the problem.
- Across files, lint detects cases where a non **void** function does not return a value, yet is used for its value in an expression, and the opposite problem, a function returning a value that is sometimes or always ignored in subsequent calls. When the value is always ignored, it may indicate an inefficiency in the function definition. When it is sometimes ignored, it's probably bad style (typically, not testing for error conditions). If you do not need to check the return values of string functions like `strcat()`, `strcpy()`, and `sprintf()`, or output functions like `printf()` and `putchar()`, cast the offending call(s) to **void**.
- lint identifies variables or functions that are declared but not used or defined; used but not defined; or defined but not used. That means that when lint is applied to some, but not all files of a collection to be loaded together, it will complain about functions and variables declared in those files but defined or used elsewhere; used there but defined elsewhere; or defined there and used elsewhere. Invoke the `-x` option to suppress the former complaint, `-u` to suppress the latter two.

Portability checks

Some nonportable code is flagged by lint in its default behavior, and a few more cases are diagnosed when lint is invoked with **-p** and/or **-Xc**. The latter tells lint to check for constructs that do not conform to the ANSI C standard. For the messages issued under **-p** and **-Xc**, check “Using lint” (page 159).

Examples:

- In some C language implementations, character variables that are not explicitly declared **signed** or **unsigned** are treated as signed quantities with a range typically from -128 to 127. In other implementations, they are treated as nonnegative quantities with a range typically from 0 to 255. So the test

```
char c;
```

```
c = getchar();
if (c == EOF) . . .
```

where **EOF** has the value -1, will always fail on machines where character variables take on nonnegative values. One of lint’s **-p** checks will flag any comparison that implies a “plain” **char** may have a negative value. Note, however, that declaring **c** a **signed char** in the above example eliminates the diagnostic, not the problem. That’s because **getchar()** must return all possible characters and a distinct **EOF** value, so a **char** cannot store its value. This example, which is perhaps the most common one arising from implementation-defined sign-extension, shows how a thoughtful application of lint’s portability option can help you discover bugs not related to portability. In any case, declare **c** as an **int**.

- A similar issue arises with bit-fields. When constant values are assigned to bit-fields, the field may be too small to hold the value. On a machine that treats bit-fields of type **int** as unsigned quantities, the values allowed for **int x:3** range from 0 to 7, whereas on machines that treat them as signed quantities they range from -4 to 3. However unintuitive it may seem, a three-bit field declared type **int** cannot hold the value 4 on the latter machines. lint invoked with **-p** flags all bit-field types other than **unsigned int** or **signed int**. Note that these are the only portable bit-field types. The compilation system supports **int**, **char**, **short**, and **long** bit-field types that may be **unsigned**, **signed**, or “plain.” It also supports the **enum** bit-field type.

- Bugs can arise when a larger-sized type is assigned to a smaller-sized type. If significant bits are truncated, accuracy is lost:

```
short s;
long l;
s = l;
```

lint flags all such assignments by default; the diagnostic can be suppressed by invoking the **-a** option. Bear in mind that you may be suppressing other diagnostics when you invoke lint with this or any other option. Check the list in “Using lint” (page 159) for the options that suppress more than one diagnostic.

- A cast of a pointer to one object type to a pointer to an object type with stricter alignment requirements may not be portable. lint flags

```
int *fun(y)
    char *y;
{
    return(int *)y;
}
```

because, on most machines, an **int** cannot start on an arbitrary byte boundary, whereas a **char** can. If you suppress the diagnostic by invoking lint with **-h**, you may be disabling other messages. You can eliminate the problem by using the generic pointer **void ***.

- ANSI C leaves the order of evaluation of complicated expressions undefined. What this means is that when function calls, nested assignment statements, or the increment and decrement operators cause side effects — when a variable is changed as a byproduct of the evaluation of an expression — the order in which the side effects take place is highly machine dependent. By default, lint flags any variable changed by a side effect and used elsewhere in the same expression:

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

Note that in this example the value of **a[1]** may be 1 if one compiler is used, 2 if another. The bitwise logical operator **&** can also give rise to this diagnostic when it is mistakenly used in place of the logical operator **&&**:

```
if ((c = getchar()) != EOF & c != '0')
```


Suspicious constructs

lint flags a number of valid constructs that may not represent what the programmer intended. Examples:

- An **unsigned** variable always has a nonnegative value. So the test

```
unsigned x;
if (x < 0) . . .
```

will always fail. Whereas the test

```
unsigned x;
if (x > 0) . . .
```

is equivalent to

```
if (x != 0) . . .
```

which may not be the intended action. lint flags suspicious comparisons of **unsigned** variables with negative constants or 0. To compare an **unsigned** variable to the bit pattern of a negative number, cast it to **unsigned**:

```
if (u == (unsigned) -1) . . .
```

Or use the **U** suffix:

```
if (u == -1U) . . .
```

- lint flags expressions without side effects that are used in a context where side effects are expected, where the expression may not represent what the programmer intended. It issues an additional warning whenever the equality operator is found where the assignment operator was expected, in other words, where a side effect was expected:

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- lint cautions you to parenthesize expressions that mix both the logical and bitwise operators (specifically, **&**, **|**, **^**, **<<**, **>>**), where misunderstanding of operator precedence may lead to incorrect results. Because the precedence of bitwise **&**, for example, falls below logical **==**, the expression

```
if (x & a == 0) . . .
```

will be evaluated as

```
if (x & (a == 0)) . . .
```

which is most likely not what you intended. Invoking lint with **-h** disables the diagnostic.

Using lint

You invoke lint with a command of the form

```
$ lint file.c file.c
```

lint examines code in two passes. In the first, it checks for error conditions local to C source files, in the second for inconsistencies across them. This process is invisible to the user unless lint is invoked with `-c`:

```
$ lint -c file1.c file2.c
```

That command directs lint to execute the first pass only and collect information relevant to the second — about inconsistencies in definition and use across *file1.c* and *file2.c* — in intermediate files named *file1.ln* and *file2.ln*:

```
$ ls -l
file1.c
file1.ln
file2.c
file2.ln
```

In this way, the `-c` option to lint is analogous to the `-c` option to `cc`, which suppresses the link editing phase of compilation. Generally speaking, lint's command line syntax closely follows `cc`'s.

When the *.ln* files are linted

```
$ lint file1.ln file2.ln
```

the second pass is executed. lint processes any number of *.c* or *.ln* files in their command line order. So

```
$ lint file1.ln file2.ln file3.c
```

directs lint to check *file3.c* for errors internal to it and all three files for consistency.

lint searches directories for included header files in the same order as `cc`

NOTE For further information, see “Searching for a header file” (page 40).

Use the **-I** option to lint as you would the **-I** option to **cc**. If you want lint to check an included header file that is stored in a directory other than your current directory or the standard place, specify the path of the directory with **-I** as follows:

```
$ lint -Idir file1.c file2.c
```

You can specify **-I** more than once on the lint command line. Directories are searched in the order they appear on the command line. Of course, you can specify multiple options to lint on the same command line. Options may be concatenated unless one of the options takes an argument:

```
$ lint -cp -Idir -Idir file1.c file2.c
```

That command directs lint to

- execute the first pass only
- perform additional portability checks
- search the specified directories for included header files

lint libraries

You can use lint libraries to check your program for compatibility with the library functions you have called in it: the declaration of the function return type, the number and types of arguments the function expects, and so on. The standard lint libraries correspond to libraries supplied by the C compilation system, and generally are stored in the standard place on your system, the directory */usr/ccs/lib*. By convention, lint libraries have names of the form *llib-lx.ln*.

The lint standard C libraries, *llib-lgen* and *llib-crypt*, are appended to the lint command line by default; checks for compatibility with it can be suppressed by invoking the **-n** option. Other lint libraries are accessed as arguments to **-l**.

```
$ lint -lx file1.c file2.c
```

directs lint to check the usage of functions and variables in *file1.c* and *file2.c* for compatibility with the lint library *llib-lx.ln*. The library file, which consists only of definitions, is processed exactly as are ordinary source files and ordinary *.ln* files, except that functions and variables used inconsistently in the library file, or defined in the library file but not used in the source files, elicit no complaints.

To create your own lint library, insert the directive `/* LINTLIBRARY */` at the head of a C source file, then invoke lint for that file with the **-o** option and the library name that will be given to **-l**:

```
$ lint -ox files
headed by /* LINTLIBRARY */
```

causes only definitions in the source files headed by `/* LINTLIBRARY */` to be written to the file `llib-lx.ln`. (Note the analogy of **lint -o** to **cc -o**.) A library can be created from a file of function prototype declarations in the same way, except that both `/* LINTLIBRARY */` and `/* PROTOLIBn */` must be inserted at the head of the declarations file. If *n* is 1, prototype declarations will be written to a library `.ln` file just as are old-style definitions. If *n* is 0, the default, the process is cancelled. Invoking **lint** with **-y** is another way of creating a lint library:

```
$ lint -y -ox file1.c file2.c
```

causes each source file named on the command line to be treated as if it began with `/* LINTLIBRARY */` and only its definitions to be written to `llib-lx.ln`.

By default, **lint** searches for lint libraries in the standard place. To direct **lint** to search for a lint library in a directory other than the standard place, specify the path of the directory with the **-L** option:

```
$ lint -Ldir -lx file1.c file2.c
```

The specified directory is searched before the standard place.

lint filters

A lint filter is a project-specific post-processor that typically uses an **awk** script or similar program to read the output of **lint** and discard messages that your project has decided do not identify real problems — string functions, for instance, returning values that are sometimes or always ignored. It enables you to generate customized diagnostic reports when **lint** options and directives do not provide sufficient control over output.

Two options to **lint** are particularly useful in developing a filter. Invoking **lint** with **-s** causes compound diagnostics to be converted into simple, one-line messages issued for each occurrence of the problem diagnosed. The easily parsed message format is suitable for analysis by an **awk** script.

Invoking **lint** with **-k** causes certain comments you have written in the source file to be printed in output, and can be useful both in documenting project decisions and specifying the post-processor's behavior. In the latter instance, if the comment identified an expected lint message, and the reported message was the same, the message might be filtered out. To use **-k**, insert on the line preceding the code you want to comment the `/* LINTED [msg] */` directive, where *msg* refers to the comment to be printed when **lint** is invoked with **-k**. (Refer to the list of directives below for what **lint** does when **-k** is not invoked for a file containing `/* LINTED [msg] */`.)

Options and directives listed

These options suppress specific messages:

- a Suppress:
 - assignment causes implicit narrowing conversion
 - conversion to larger integral type may sign-extend incorrectly
- b For unreachable **break** and empty statements, suppress:
 - statement not reached
- h Suppress:
 - assignment operator "=" found where equality operator "==" was expected
 - constant operand to op: "!"
 - fallthrough on case statement
 - pointer cast may result in improper alignment
 - precedence confusion possible; parenthesize
 - statement has no consequent: if
 - statement has no consequent: else
- m Suppress:
 - declared global, could be static
- u Suppress:
 - name defined but never used
 - name used but not defined
- v Suppress:
 - argument unused in function
- x Suppress:
 - name declared but never used or defined

These options enable specific messages:

- p Enable:
 - conversion to larger integral type may sign-extend incorrectly
 - may be indistinguishable due to truncation or case
 - pointer casts may be troublesome

- nonportable bit-field type
 - suspicious comparison of char with *value*: op "*op*"
- Xc** Enable:
- bitwise operation on signed value nonportable
 - function must return int: main()
 - may be indistinguishable due to truncation or case
 - only 0 or 2 parameters allowed: main()
 - nonportable character constant

Other options:

- c** Create a *.ln* file consisting of information relevant to lint's second pass for every *.c* file named on the command line. The second pass is not executed.
- F** When referring to the *.c* files named on the command line, print their path names as supplied on the command line rather than only their base names.
- Idir** Search the directory *dir* for included header files.
- k** When used with the directive `/* LINTED [msg] */`, print info: *msg*.
- lx** Access the lint library *llib-lx.ln*.
- Ldir** When used with **-l**, search for a lint library in the directory *dir*.
- n** Suppress checks for compatibility with the default lint standard C libraries.
- ox** Create the file *llib-lx.ln*, consisting of information relevant to lint's second pass, from the *.c* files named on the command line. Generally used with **-y** or `/* LINTLIBRARY */` to create lint libraries.
- s** Convert compound messages into simple ones.
- y** Treat every *.c* file named on the command line as if it began with the directive `/* LINTLIBRARY */`.
- V** Write the product name and release to standard error.

Directives:

```
/* ARGUSEDn */
```

Suppress:

- argument unused in function
- for every argument but the first *n* in the function definition it precedes. Default is 0.

```
/* CONSTCOND */
    Suppress:
        • "constant in conditional context"
        • "constant operand to op: "!"
        • "logical expression always false: op "&&"
        • "logical expression always true: op "||"
    for the constructs it precedes. Also /* CONSTANTCONDITION */.

/* EMPTY */
    Suppress:
        • "statement has no consequent: else"
        when inserted between the else and semicolon;
        • "statement has no consequent: if"
        when inserted between the controlling expression of the if and semicolon.

/* FALLTHRU */
    Suppress:
        • "fallthrough on case statement"
    for the case statement it precedes. Also /* FALLTHROUGH */.

/* LINTED [msg] */
    When -k is not invoked, suppress every warning pertaining to an intrafile
    problem except:
        • "argument unused in function"
        • "declaration unused in block"
        • "set but not used in function"
        • "static unused"
        • "variable unused in function"
    for the line of code it precedes. msg is ignored.

/* LINTLIBRARY */
    When -o is invoked, write to a library .ln file only definitions in the .c file it
    heads.

/* NOTREACHED */
    Suppress:
        • "statement not reached"
        for the unreachable statements it precedes;
        • "fallthrough on case statement"
        for the case it precedes that cannot be reached from the preceding case;
        • "function falls off bottom without returning value"
        for the closing curly brace it precedes at the end of the function.
```

```
/* PRINTFLIKE $n$  */
```

Treat the n th argument of the function definition it precedes as a **[fs]printf()** format string and issue:

- "malformed format string"
- "function argument type inconsistent with format"
- "too few arguments for format"
- "too many arguments for format"

for mismatches between the remaining arguments and the conversion specifications. lint issues these warnings by default for errors in calls to **[fs]printf()** functions provided by the standard C libraries.

```
/* PROTOLIB $n$  */
```

When n is 1 and `/* LINTLIBRARY */` is used, write to a library `.ln` file only function prototype declarations in the `.c` file it heads. Default is 0, canceling the process.

```
/* SCANFLIKE $n$  */
```

Same as `/* PRINTFLIKE $n$  */` except that the n th argument of the function definition is treated as a **[fs]scanf()** format string. By default, lint issues warnings for errors in calls to **[fs]scanf()** functions provided by the standard C libraries.

```
/* VARARGS $n$  */
```

For the function whose definition it precedes, suppress:

- "function called with variable number of arguments"
- for calls to the function with n or more arguments.

Understanding lint-specific messages

This section lists alphabetically the warning messages issued exclusively by lint or subject exclusively to its options. The code examples illustrate conditions in which the messages are elicited. Note that some of the examples would elicit messages in addition to the one stated. For the remaining lint messages, consult Appendix B, "C compiler diagnostics" (page 515).

argument unused in function

Format: Compound

A function argument was not used. Preceding the function definition with `/* ARGSUSEDn */` suppresses the message for all but the first *n* arguments; invoking lint with `-v` suppresses it for every argument.

```

1  int fun(int x, int y)
2  {
3      return x;
4  }
5  /* ARGSUSED1 */
6  int fun2(int x, int y)
7  {
8      return x;
9  }
=====
argument unused in function
      (1) y in fun

```

array subscript cannot be > *value*: *value*

Format: Simple

The value of an array element's subscript exceeded the upper array bound.

```

1  int fun()
2  {
3      int a[10];
4      int *p = a;
5      while (p != &a[10]) /* using address is ok */
6          p++;
7      return a[5 + 6];
8  }
=====
(7) warning: array subscript cannot be > 9: 11

```

array subscript cannot be negative: *value*

Format: Simple

The constant expression that represents the subscript of a true array (as opposed to a pointer) had a negative value.

```

1  int f()
2  {
3      int a[10];
4      return a[5 * 2 / 10 - 2];
5  }
=====
(4) warning: array subscript cannot be negative: -1

```

assignment causes implicit narrowing conversion

Format: Compound

An object was assigned to one of a smaller type. Invoking lint with **-a** suppresses the message. So does an explicit cast to the smaller type.

```

1 void fun()
2 {
3     short s;
4     long l = 0;
5     s = l;
6 }
=====
assignment causes implicit narrowing conversion
(5)

```

assignment of negative constant to unsigned type

Format: Simple

A negative constant was assigned to a variable of **unsigned** type. Use a cast or the **U** suffix.

```

1 void fun()
2 {
3     unsigned i;
4     i = -1;
5     i = -1U;
6     i = (unsigned) (-4 + 3);
7 }
=====
(4) warning: assignment of negative constant to unsigned type

```

assignment operator "=" found where "==" was expected

Format: Simple

An assignment operator was found where a conditional expression was expected. The message is not issued when an assignment is made to a variable using the value of a function call or in the case of string copying (see the example below). The warning is suppressed when lint is invoked with **-h**.

```

1  void fun()
2  {
3      char *p, *q;
4      int a = 0, b = 0, c = 0, d = 0, i;
5      i = (a = b) && (c == d);
6      i = (c == d) && (a = b);
7      if (a = b)
8          i = 1;
9      while (*p++ = *q++);
10     while (a = b);
11     while ((a = getchar()) == b);
12     if (a = foo()) return;
13 }
=====
(5) warning: assignment operator "=" found where "=="
      was expected
(7) warning: assignment operator "=" found where "=="
      was expected
(10) warning: assignment operator "=" found where "=="
      was expected

```

bitwise operation on signed value nonportable

Format: Compound

The operand of a bitwise operator was a variable of signed integral type, as defined by ANSI C. Because these operators return values that depend on the internal representations of integers, their behavior is implementation-defined for operands of that type. The message is issued only when lint is invoked with **-Xc**.

```

1  fun()
2  {
3      int i;
4      signed int j;
5      unsigned int k;
6      i = i & 055;
7      j = j | 022;
8      k = k >> 4;
9  }
=====
warning: bitwise operation on signed value nonportable
      (6)      (7)

```

constant in conditional context

Format: Simple

The controlling expression of an **if**, **while**, or **for** statement was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message.

```

1 void fun()
2 {
3     if (! 1) return;
4     while (1) foo();
5     for (;1;);
6     for (;;);
7     /* CONSTCOND */
8     while (1);
9 }
=====
(3) warning: constant in conditional context
(4) warning: constant in conditional context
(5) warning: constant in conditional context

```

constant operand to op: "!"

Format: Simple

The operand of the NOT operator was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message for that statement; invoking lint with `-h` suppresses it for every statement.

```

1 void fun()
2 {
3     if (! 0) return;
4     /* CONSTCOND */
5     if (! 0) return;
6 }
=====
(3) warning: constant operand to op: "!"

```

constant truncated by assignment

Format: Simple

An integral constant expression was assigned or returned to an object of an integral type that cannot hold the value without truncation.

```

1 unsigned char f()
2 {
3     unsigned char i;
4     i = 255;
5     i = 256;
6     return 256;
7 }
=====
(5) warning: constant truncated by assignment
(6) warning: constant truncated by assignment

```

conversion of pointer loses bits

Format: Simple

A pointer was assigned to an object of an integral type that is smaller than the pointer.

```
1 void fun()
2 {
3     char c;
4     int *i;
5     c = i;
6 }
```

=====

(5) warning: conversion of pointer loses bits

conversion to larger integral type may sign-extend incorrectly

Format: Compound

A variable of type “plain” **char** was assigned to a variable of a larger integral type. Whether a “plain” **char** is treated as signed or unsigned is implementation-defined. The message is issued only when lint is invoked with **-p**, and is suppressed when it is invoked with **-a**.

```
1 void fun()
2 {
3     char c = 0;
4     short s = 0;
5     long l;
6     l = c;
7     l = s;
8 }
```

=====

conversion to larger integral type may sign-extend incorrectly

(6)

declaration unused in block

Format: Compound

An external variable or function was declared but not used in an inner block.

```

1  int fun()
2  {
3      int foo();
4      int bar();
5      return foo();
6  }
=====
declaration unused in block
      (4) bar

```

declared global, could be static

Format: Compound

An external variable or function was declared global, instead of **static**, but was referenced only in the file in which it was defined. The message is suppressed when lint is invoked with **-m**.

file: f1.c

```

1  int i;
2  int foo() {return i;}
3  int fun() {return i;}
4  static int stfun() {return fun();}

```

file: f2.c

```

1  main()
2  {
3      int a;
4      a = foo();
5  }

```

=====

declared global, could be static

```

      fun          f1.c(3)
      i            f1.c(1)

```

equality operator "==" found where "=" was expected

Format: Simple

An equality operator was found where a side effect was expected.

```
1 void fun(a, b)
2 int a, b;
3 {
4     a == b;
5     for (a == b; a < 10; a++);
6 }
```

=====

(4) warning: equality operator "==" found where "=" was expected

(5) warning: equality operator "==" found where "=" was expected

evaluation order undefined: *name*

Format: Simple

A variable was changed by a side effect and used elsewhere in the same expression.

```
1 int a[10];
2 main()
3 {
4     int i = 1;
5     a[i++] = i;
6 }
```

=====

(5) warning: evaluation order undefined: i

fallthrough on case statement

Format: Simple

Execution fell through one case to another without a **break** or **return**. Preceding a **case** statement with `/* FALLTHRU */`, or `/* NOTREACHED */` when the case cannot be reached from the preceding case (see below), suppresses the message for that statement; invoking lint with **-h** suppresses it for every statement.

```

1 void fun(i)
2 {
3     switch (i) {
4         case 10:
5             i = 0;
6         case 12:
7             return;
8         case 14:
9             break;
10        case 15:
11        case 16:
12            break;
13        case 18:
14            i = 0;
15            /* FALLTHRU */
16        case 20:
17            error("bad number");
18            /* NOTREACHED */
19        case 22:
20            return;
21    }
22 }

```

=====

(6) warning: fallthrough on case statement

function argument (number) declared inconsistently

Format: Compound

The parameter types in a function prototype declaration or definition differed from their types in another declaration or definition. The message described after this one is issued for uses (not declarations or definitions) of a prototype with the wrong parameter types.

file: i3a.c

```
1  int fun1(int);
2  int fun2(int);
3  int fun3(int);
```

file: i3b.c

```
1  int fun1(int *i);
2  int fun2(int *i) {}
3  void foo()
4  {
5      int *i;
6      fun3(i);
7  }
```

=====

```
function argument ( number ) declared inconsistently
    fun2 (arg 1)          i3b.c(2) int * :: i3a.c(2) int
    fun1 (arg 1)          i3a.c(1) int :: i3b.c(1) int *
function argument ( number ) used inconsistently
    fun3 (arg 1)          i3a.c(3) int :: i3b.c(6) int *
```

function argument (number) used inconsistently

Format: Compound

The argument types in a function call did not match the types of the formal parameters in the function definition. (And see the discussion of the preceding message.)

file: f1.c

```
1  int fun(int x, int y)
2  {
3      return x + y;
4  }
```

file: f2.c

```
1  int main()
2  {
3      int *x;
4      extern int fun();
5      return fun(1, x);
6  }
```

=====

```
function argument ( number ) used inconsistently
    fun( arg 2 )          f1.c(2) int :: f2.c(5) int *
```

function argument type inconsistent with format

Format: Compound

An argument was inconsistent with the corresponding conversion specification in the control string of a **[fs]printf()** or **[fs]scanf()** function call. (See also `/* PRINTFLIKE $n$  */` and `/* SCANFLIKE $n$  */` in the list of directives in “Using lint” (page 159).)

```
1  #include <stdio.h>
2  main()
3  {
4      int i;
5      printf("%s", i);
6  }
```

=====

```
function argument type inconsistent with format
printf(arg 2) int :: (format) char * test.c(5)
```

function called with variable number of arguments

Format: Compound

A function was called with the wrong number of arguments. Preceding a function definition with `/* VARARGS $n$  */` suppresses the message for calls with n or more arguments; defining and declaring a function with the ANSI C notation “...” suppresses it for every argument.

NOTE See “function declared with variable number of arguments” for more information.

file: f1.c

```

1  int fun(int x, int y, int z)
2  {
3      return x + y + z;
4  }
5  int fun2(int x, . . .)
6  {
7      return x;
8  }
10 /* VARARGS1 */
11 int fun3(int x, int y, int z)
12 {
13     return x;
14 }

```

file: f2.c

```

1  int main()
2  {
3      extern int fun(), fun3(), fun2(int x, . . .);
4      return fun(1, 2);
5      return fun2(1, 2, 3, 4);
6      return fun3(1, 2, 3, 4, 5);
7  }

```

```

=====

```

```

function called with variable number of arguments
    fun          f1.c(2) :: f2.c(4)

```

function declared with variable number of arguments

Format: Compound

The number of parameters in a function prototype declaration or definition differed from their number in another declaration or definition. Declaring and defining the prototype with the ANSI C notation `". . ."` suppresses the warning if all declarations have the same number of arguments. The message immediately preceding this one is issued for uses (not declarations or definitions) of a prototype with the wrong number of arguments.

file: i3a.c

```
1  int fun1(int);
2  int fun2(int);
3  int fun3(int);
```

file: i3b.c

```
1  int fun1(int, int);
2  int fun2(int a, int b) {}
3  void foo()
4  {
5      int i, j, k;
6      i = fun3(j, k);
7  }
```

=====

function declared with variable number of arguments

```
    fun2          i3a.c(2) :: i3b.c(2)
```

```
    fun1          i3a.c(1) :: i3b.c(1)
```

function called with variable number of arguments

```
    fun3          i3a.c(3) :: i3b.c(6)
```

function falls off bottom without returning value

Format: Compound

A non **void** function did not return a value to the invoking function. If the closing curly brace is truly not reached, preceding it with `/* NOTREACHED */` suppresses the message.

```
1  fun()
2  {}
3  void fun2()
4  {}
5  foo()
6  {
7      exit(1);
8      /* NOTREACHED */
9  }
```

=====

function falls off bottom without returning value

```
(2) fun
```

function must return int: main()

Format: Simple

The program's **main()** function does not return **int**, in violation of ANSI C restrictions. The message is issued only when lint is invoked with **-Xc**.

```
1  void main()
2  {}
```

=====

```
(2) warning: function must return int: main()
```

function returns pointer to [automatic/parameter]

Format: Simple

A function returned a pointer to an automatic variable or a parameter. Since an object with automatic storage duration is no longer guaranteed to be reserved after the end of the block, the value of the pointer to that object will be indeterminate after the end of the block.

```

1  int *fun(int x)
2  {
3      int a[10];
4      int b;
5      if (x == 1)
6          return a;
7      else if (x == 2)
8          return &b;
9      else return &x ;
10 }
=====
(6) warning: function returns pointer to automatic
(8) warning: function returns pointer to automatic
(9) warning: function returns pointer to parameter

```

function returns value that is always ignored

Format: Compound

A function contained a **return** statement and every call to the function ignored its return value.

file: f1.c

```

1  int fun()
2  {
3      return 1;
4  }

```

file: f2.c

```

1  extern int fun();
2  int main()
3  {
4      fun();
5      return 1;
6  }

```

```

=====
function returns value that is always ignored
    fun

```

function returns value that is sometimes ignored

Format: Compound

A function contained a **return** statement and some, but not all, calls to the function ignored its return value.

file: f1.c

```
1  int fun()
2  {
3      return 1;
4  }
```

file: f2.c

```
1  extern int fun();
2  int main()
3  {
4      if(1) {
5          return fun();
6      }
7      else {
8          fun();
9          return 1;
10     }
```

=====

function returns value that is sometimes ignored
fun

function value is used, but none returned

Format: Compound

A non **void** function did not contain a **return** statement, yet was used for its value in an expression.

file: f1.c

```
1  extern int fun();
2  main()
3  {
4      return fun();
5  }
```

file: f2.c

```
1  int fun()
2  {}
```

=====

function value is used, but none returned
fun

logical expression always false: op "&&"

Format: Simple

A logical AND expression checked for equality of the same variable to two different constants, or had the constant 0 as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```

1 void fun(a)
2 int a;
3 {
4     a = (a == 1) && (a == 2);
5     a = (a == 1) && (a == 1);
6     a = (1 == a) && (a == 2);
7     a = (a == 1) && 0;
8     /* CONSTCOND */
9     a = (0 && (a == 1));
10 }
=====
(4) warning: logical expression always false: op "&&"
(6) warning: logical expression always false: op "&&"
(7) warning: logical expression always false: op "&&"

```

logical expression always true: op "||"

Format: Simple

A logical OR expression checked for inequality of the same variable to two different constants, or had a nonzero integral constant as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```

1 void fun(a)
2 int a;
3 {
4     a = (a != 1) || (a != 2);
5     a = (a != 1) || (a != 1);
6     a = (1 != a) || (a != 2);
7     a = (a == 10) || 1;
8     /* CONSTCOND */
9     a = (1 || (a == 10));
10 }
=====
(4) warning: logical expression always true: op "||"
(6) warning: logical expression always true: op "||"
(7) warning: logical expression always true: op "||"

```

malformed format string

Format: Compound

A **[fs]printf()** or **[fs]scanf()** control string was formed incorrectly. (See also `/* PRINTFLIKEn */` and `/* SCANFLIKEn */` in the list of directives in "Using lint" (page 159).)

```
1  #include <stdio.h>
2  main()
3  {
4      printf("%y");
5  }
=====
malformed format string
      printf      test.c(4)
```

may be indistinguishable due to truncation or case

Format: Compound

External names in a program may be indistinguishable when it is ported to another machine due to implementation-defined restrictions as to length or case. The message is issued only when lint is invoked with **-Xc** or **-p**. Under **-Xc**, external names are truncated to the first 6 characters with one case, in accordance with the ANSI C lower bound; under **-p**, to the first 8 characters with one case.

file: f1.c

```
1  int foobar1;
2  int FooBar12;
```

file: f2.c

```
1  int foobar2;
2  int FOOBAR12;
```

=====

under -p

may be indistinguishable due to truncation or case

```
FooBar12  f1.c(2)  :: FOOBAR12  f2.c(2)
```

under -Xc

may be indistinguishable due to truncation or case

```
foobar1   f1.c(1)  :: FooBar12   f1.c(2)
foobar1   f1.c(1)  :: foobar2     f2.c(1)
foobar1   f1.c(1)  :: FOOBAR12    f2.c(2)
```


name declared but never used or defined

Format: Compound

A non **static** external variable or function was declared but not used or defined in any file. The message is suppressed when lint is invoked with **-x**.

file f.c

```
1  extern int fun();
2  static int foo();
```

=====

```
name declared but never used or defined
      fun          f.c(1)
```

name defined but never used

Format: Compound

A variable or function was defined but not used in any file. The message is suppressed when lint is invoked with **-u**.

file f.c

```
1  int i, j, k = 1;
2  main()
3  {
4      j = k;
5  }
```

=====

```
name defined but never used
      i          f.c(1)
```

name multiply defined

Format: Compound

A variable was defined in more than one source file.

file f1.c

```
1  char i = 'a';
```

file f2.c

```
1  long i = 1;
```

=====

```
name multiply defined
      i          f1.c(1) :: f2.c(1)
```

name used but not defined

Format: Compound

A non **static** external variable or function was declared but not defined in any file. The message is suppressed when lint is invoked with **-u**.

file f.c

```
1  extern int fun();
2  int main()
3  {
4      return fun();
5  }
=====
name used but not defined
      fun      f.c(4)
```

nonportable bit-field type

Format: Simple

A bit-field type other than **signed int** or **unsigned int** was used. The message is issued only when lint is invoked with **-p**. Note that these are the only portable bit-field types. The compilation system supports **int**, **char**, **short**, and **long** bit-field types that may be **unsigned**, **signed**, or “plain.” It also supports the **enum** bit-field type.

```
1  struct u {
2      unsigned v:1;
3      int      w:1;
4      char     x:8;
5      long     y:8;
6      short    z:8;
7  };
=====
(3) warning: nonportable bit-field type
(4) warning: nonportable bit-field type
(5) warning: nonportable bit-field type
(6) warning: nonportable bit-field type
```

nonportable character constant

Format: Simple

A multi-character character constant in the program may not be portable. The message is issued only when lint is invoked with **-Xc**.

```
1  int c = 'abc';
=====
(1) warning: nonportable character constant
```

only 0 or 2 parameters allowed: main()

Format: Simple

The function **main()** in your program was defined with only one parameter or more than two parameters, in violation of the ANSI C requirement. The message is issued only when lint is invoked with **-Xc**.

```
1  main(int argc, char **argv, char **envp)
2  {}
=====
(2) warning: only 0 or 2 parameters allowed: main()
```

pointer cast may result in improper alignment

Format: Compound

A pointer to one object type was cast to a pointer to an object type with stricter alignment requirements. Doing so may result in a value that is invalid for the second pointer type. The warning is suppressed when lint is invoked with **-h**.

```
1  void fun()
2  {
3      short *s;
4      int *i;
5      i = (int *) s;
6  }
=====
pointer cast may result in improper alignment
(5)
```

pointer casts may be troublesome

Format: Compound

A pointer to one object type was cast to a pointer to a different object type. The message is issued only when lint is invoked with **-p**, and is not issued for the generic pointer **void ***.

```
1  void fun()
2  {
3      int *i;
4      char *c;
5      void *v;
6      i = (int *) c;
7      i = (int *) v;
8  }
=====
warning: pointer casts may be troublesome
(6)
```

precedence confusion possible; parenthesize

Format: Simple

An expression that mixes a logical and a bitwise operator was not parenthesized. The message is suppressed when lint is invoked with **-h**.

```

1 void fun()
2 {
3     int x = 0, m = 0, MASK = 0, i;
4     i = (x + m == 0);
5     i = (x & MASK == 0); /* eval'd (x & (MASK == 0)) */
6     i = (MASK == 1 & x); /* eval'd ((MASK == 1) & x) */
7 }
```

=====

(5) warning: precedence confusion possible; parenthesize

(6) warning: precedence confusion possible; parenthesize

precision lost in bit-field assignment

Format: Simple

A constant was assigned to a bit-field too small to hold the value without truncation. Note that in the following example the bit-field **z** may have values that range from 0 to 7 or -4 to 3, depending on the machine.

```

1 void fun()
2 {
3     struct {
4         signed x:3; /* max value allowed is 3 */
5         unsigned y:3; /* max value allowed is 7 */
6         int z:3; /* max value allowed is 7 */
7     } s;
8     s.x = 3;
9     s.x = 4;
10    s.y = 7;
11    s.y = 8;
12    s.z = 7;
13    s.z = 8;
14 }
```

=====

(9) warning: precision lost in bit-field assignment: 4

(11) warning: precision lost in bit-field assignment: 0x8

(13) warning: precision lost in bit-field assignment: 8

set but not used in function

Format: Compound

An automatic variable or a function parameter was declared and set but not used in a function.

```

1 void fun(y)
2 int y;
3 {
4     int x;
5     x = 1;
6     y = 1;
7 }
=====
set but not used in function
(4) x in fun
(1) y in fun

```

statement has no consequent: else

Format: Simple

An **if** statement had a null **else** part. Inserting `/* EMPTY */` between the **else** and semicolon suppresses the message for that statement; invoking `lint` with **-h** suppresses it for every statement.

```

1 void f(a)
2 int a;
3 {
4     if (a)
5         return;
6     else;
7 }
=====
(6) warning: statement has no consequent: else

```

statement has no consequent: if

Format: Simple

An **if** statement had a null **if** part. Inserting `/* EMPTY */` between the controlling expression of the **if** and semicolon suppresses the message for that statement; invoking `lint` with **-h** suppresses it for every statement.

```

1 void f(a)
2 int a;
3 {
4     if (a);
5     if (a == 10)
6         /* EMPTY */;
7     else return;
8 }
=====
(4) warning: statement has no consequent: if

```

statement has null effect

Format: Compound

An expression did not generate a side effect where a side effect was expected. Note that the message is issued for every subsequent sequence point that is reached at which a side effect is not generated.

```

1 void fun()
2 {
3     int a, b, c, x;
4     a;
5     a == 5;
6     ;
7     while (x++ != 10);
8     (a == b) && (c = a);
9     (a = b) && (c == a);
10    (a, b);
11 }
=====
statement has null effect
      (4)          (5)          (9)          (10)

```

statement not reached

Format: Compound

A function contained a statement that cannot be reached. Preceding an unreachable statement with `/* NOTREACHED */` suppresses the message for that statement; invoking lint with `-b` suppresses it for every unreachable **break** and empty statement. Note that this message is also issued by the compiler but cannot be suppressed.

```

1 void fun(a)
2 {
3     switch (a) {
4         case 1:
5             return;
6             break;
7         case 2:
8             return;
9             /* NOTREACHED */
10            break;
11    }
12 }
=====
statement not reached
      (6)

```

static unused

Format: Compound

A variable or function was defined or declared **static** in a file but not used in that file. Doing so is probably a programming error because the object cannot be used outside the file.

```

1  static int x;
2  static int main() {}
3  static int foo();
4  static int y = 1;
=====
static unused
      (4) y          (3) foo          (2) main          (1) x

```

suspicious comparison of char with *value*: op "*op*"

Format: Simple

A comparison was performed on a variable of type "plain" **char** that implied it may have a negative value (< 0 , ≤ 0 , ≥ 0 , > 0). Whether a "plain" **char** is treated as signed or nonnegative is implementation-defined. The message is issued only when lint is invoked with **-p**.

```

1  void fun(c, d)
2  char c;
3  signed char d;
4  {
5      int i;
6      i = (c == -5);
7      i = (c < 0);
8      i = (d < 0);
9  }
=====
(6) warning: suspicious comparison of char with negative
      constant: op "=="
(7) warning: suspicious comparison of char with 0: op "<"

```

suspicious comparison of unsigned with *value*: op "*op*"

Format: Simple

A comparison was performed on a variable of **unsigned** type that implied it may have a negative value (< 0, <= 0, >= 0, > 0).

```

1 void fun(x)
2 unsigned x;
3 {
4     int i;
5     i = (x > -2);
6     i = (x < 0);
7     i = (x <= 0);
8     i = (x >= 0);
9     i = (x > 0);
10    i = (-2 < x);
11    i = (x == -1);
12    i = (x == -1U);
13 }
```

=====

- (5) warning: suspicious comparison of unsigned with negative constant: op ">"
- (6) warning: suspicious comparison of unsigned with 0: op "<"
- (7) warning: suspicious comparison of unsigned with 0: op "<="
- (8) warning: suspicious comparison of unsigned with 0: op ">="
- (9) warning: suspicious comparison of unsigned with 0: op ">"
- (10) warning: suspicious comparison of unsigned with negative constant: op "<"
- (11) warning: suspicious comparison of unsigned with negative constant: op "=="

too few arguments for format

Format: Compound

A control string of a **[fs]printf()** or **[fs]scanf()** function call had more conversion specifications than there were arguments remaining in the call. (See also */* PRINTFLIKE*n* */* and */* SCANFLIKE*n* */* in the list of directives in "Using lint" (page 159).)

```

1 #include <stdio.h>
2 main()
3 {
4     int i;
5     printf("%d%d", i);
6 }
```

=====

too few arguments for format
printf test.c(5)

too many arguments for format

Format: Compound

A control string of a **[fs]printf()** or **[fs]scanf()** function call had fewer conversion specifications than there were arguments remaining in the call. (See also `/* PRINTFLIKE $n$  */` and `/* SCANFLIKE $n$  */` in the list of directives in “Using lint” (page 159).)

```
1  #include <stdio.h>
2  main()
3  {
4      int i, j;
5      printf("%d", i, j);
6  }
=====
too many arguments for format
    printf      test.c(5)
```

value type declared inconsistently

Format: Compound

The return type in a function declaration or definition did not match the return type in another declaration or definition of the function. The message is also issued for inconsistent declarations of variable types.

file f1.c

```
1  void fun() {}
2  void foo();
3  extern int a;
```

file f2.c

```
1  extern int fun();
2  extern int foo();
3  extern char a;
```

=====

value type declared inconsistently

```
fun      f1.c(1) void() :: f2.c(1) int()
foo      f1.c(2) void() :: f2.c(2) int()
a        f1.c(3) int  :: f2.c(3) char
```

value type used inconsistently

Format: Compound

The return type in a function call did not match the return type in the function definition.

file f1.c

```
1  int *fun(p)
2  int *p;
3  {
4      return p;
5  }
```

file f2.c

```
1  main()
2  {
3      int i, *p;
4      i = fun(p);
5  }
```

=====

value type used inconsistently

fun f1.c(3) int *() :: f2.c(4) int()

variable may be used before set: *name*

Format: Simple

The first reference to an automatic, non-array variable occurred at a line number earlier than the first assignment to the variable. Note that taking the address of a variable implies both a set and a use, and that the first assignment to any member of a **struct** or **union** implies an assignment to the entire **struct** or **union**.

```
1  void fun()
2  {
3      int i, j, k;
4      static int x;
5      k = j;
6      i = i + 1;
7      x = x + 1;
8  }
```

=====

(5) warning: variable may be used before set: j

(6) warning: variable may be used before set: i

variable unused in function

Format: Compound

A variable was declared but never used in a function.

```
1 void fun()  
2 {  
3     int x, y;  
4     static z;  
5 }
```

=====

variable unused in function

(4) z in fun

(3) y in fun

(3) x in fun

Chapter 6

dbXtra and dbxtra

dbXtra and **dbxtra** provide both Motif and **curses** based interfaces to a machine- and source-level debugger for C and assembly language programs. The functionality is a superset of the **dbx**(CP) debugger available with many UNIX systems.

Features

Features include source code presentation through windows as well as:

- C and C++ compatibility
- COFF and ELF object file compatibility
- conditional breakpointing and symbolic variable tracing
- keyboard configuration enabling debugger commands to be issued with a single keystroke
- symbolic access to all active functions and variables
- online cross-referencing capabilities
- program input and output redirection to a different screen or terminal
- simultaneous user interaction and examination of source code, variable values, and program output

Invoking dbXtra and dbxtra from the command line

The syntax for **dbXtra** and **dbxtra** is:

```
dbXtra [X11-options] [options] [objfile [corefile | processid]]
dbxtra [options] [objfile [corefile]]
```

The first form of the command is used within the X Window System Version 11 (X11). It invokes **dbXtra**(CP), the Motif version of the debugger, which uses various X11 and Motif objects. All *X11-options* are passed on to X11. See **X(X)**.

You may need to set the environment variable **DISPLAY** in order to run **dbXtra**. This variable informs your system of the address of the X Window display you are using. See **X(X)**.

The second form of the command invokes **dbxtra**(CP), the **curses** version of the debugger. That version is suitable for use with most common alphanumeric terminals as well as from within a terminal emulator window application (such as **xterm**) within X windows.

The object file specified by *objfile* contains the executable instructions of the program to be debugged. If *objfile* is not specified, you are prompted for the name of the file. If this object file was compiled with the **-g** flag, also included in the file are the symbol table information needed to identify the variables by their source names, and the line numbers and filenames of the source code for each executable source line. A warning is issued if the object file does not have this extra information.

The core file specified by *corefile* determines the state of the program when it faulted. If it is not specified, then the file named *core* in the current directory is used if it exists. If the core file does not coincide with the program being debugged, for example, if it was created when another program had faulted, it is ignored.

If the file *.dbxtrarc* exists in the current directory, then the debugger commands in it are executed. The debugger also checks for a *.dbxtrarc* in the user's home directory if there is not one in the current directory.

One or more of the following options may be specified in any order:

-C++ Invoke in C++ interaction mode; assumed by default. Use this option if the program being debugged was developed using C++. It has no effect if the program was written in C. Use of this option activates the **dbXtra** Options menu button C/C++, which toggles between C and C++ interaction modes.

For further information about debugging C++ programs, see “C++ specific debugging features” (page 235).

- C Invoke in C interaction mode only. C++ information is not read in and the **dbXtra** Options menu button **C/C++** is not activated.
- F Load only one symbol table entry for each unique definition of a structure or union or enum. This means that if more than one definition of an identifier, at the same scope, is present in a program (for example, if the definition is in an include file used by various modules), then all the definitions of the identifier are assumed to be the same.

The use of this option results in faster load time and requires significantly less memory for larger programs.

NOTE If the program contains non-unique definitions of the same identifier at the same scope, use of this option causes the symbol table of the debugger to be different from that of the program.

- I *dir* By default, the debugger searches for source files in the current directory and in the directory where *objfile* is located. The **-I** option adds the specified directory *dir* to the search path. Directories can also be added to the directory search path with the **use** command (Source menu button: Source Path). You can specify multiple **-I** options on the command line. The order of the directories in the search path is the same as the order specified on the command line.
- c *file* Execute the debugging commands in the *file* before reading from standard input. These commands are read *after* any *.dbxtrarc* files are read.
- r Execute *objfile* immediately. If it terminates successfully, the debugger exits. Otherwise, the reason for termination is reported and you are offered the option of entering the debugger or letting the program fault.

When **-r** is specified and standard input is not a terminal, the debugger reads from */dev/tty*.
- s For **dbxtra** only; do not invoke the screen manager. The debugger works in line mode only and does not respond to any of the screen-related commands (including **screen**).
- x Ignore *xref* file when initializing.

NOTE Unless **-r** is specified, **dbXtra** and **dbxtra** prompt and wait for a command upon invocation.

The dbXtra interface

When invoked, **dbXtra**'s interface has a title bar, a pull-down menu bar on the top, sidebar command buttons on the left-hand side, the source window, and the command window below it, as in Figure 6-1, "dbXtra interface".

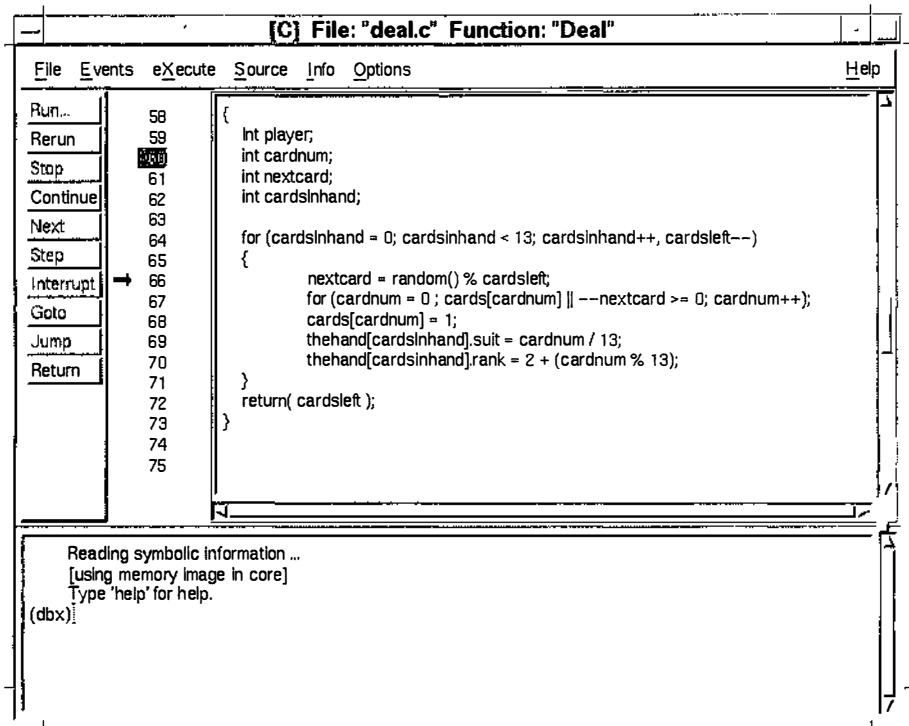


Figure 6-1 dbXtra interface

The components of the dbXtra interface are:

title bar The title bar displays the current debugging mode. [C] or [C++], and the current file and function names.

pull-down menu bar

The menu bar across the top contains several pull-down menus. These are accessed through the mouse, or through the keyboard accelerator sequence for the menu (simultaneously press the <Alt> key and the uppercase letter in the menu name). Each menu contains a logical group of buttons performing related operations. For further discussion of each menu and the command line operations performed, see "dbXtra and dbxtra commands" (page 215).

File	Events	eXecute	Source	Info	Options	Help
Write Configuration File	Events	Run	Browse file	Show	C/C++	
Exit dbXtra	Call	Rerun	Source Path	Where	Preferences	
	When	Continue	Search	Display	Environment	
	Stop	Skip	Function	Assign	Fork	
	Trace	Next	dbXref	Which	Trace	
	Interrupt	Step	Up	Print		
	Catch	Goto	Down	Xref		
	Ignore	Jump		Variables		
	Dbxtra	Return		Registers		
		Instruction		Memory		

sidebar buttons

The sidebar buttons on the left-hand side provide a quick entry point for frequently-used command buttons in the Events and eXecute pull-down menus.

source window

The source window shows the current position in the current file. This is a read-only view of the file; the program cannot be modified or edited. The current line numbers and breakpoint status are displayed to the left of the source window. The current execution point is indicated by an arrow [->]. Breakpoints (or other events) are indicated by an [S] at the appropriate lines.

Associated with the source window is a pop-up menu that appears when mouse button 3 is pressed. The menu items provide a second entry point for frequently-used command buttons in the Source and Info pull-down menus.

command window

The command window provides direct access to the complexities of the underlying debugger. Output from most debugger commands appear in this window and any **dbxtra** command can be entered from this window.

Configuring dbXtra X11 and Motif resources

As with most X11 applications, you can configure a number of resources to modify the appearance of **dbXtra** to suit your needs. See Chapter 5, "Understanding resources" in the *Graphical Environment Guide* and "About resources" in the *Graphical Environment Guide*.

The values of the resources can be specified in the *.Xdefaults-hostname* file in your home directory, where *hostname* is the name of the machine on which the X11 client (for which you want to specify resources) is running. Resources can also be specified in a file called *dbXtra* in */usr/lib/X11/app-defaults*, or in the *app-defaults* directory in your home directory.

The resources that control the appearance of **dbXtra** have the following syntax:

dbXtra*resource: *value*

See **dbXtra**(CP) for a description of **dbXtra** resources. In addition, resources can be set to manipulate the various Motif widgets utilized. For lists of Motif resources, see Appendix A, “Motif Window Manager resources” in the *Graphical Environment Guide*.

The dbxtra interface

When invoked, **dbxtra** has a source window on top of an interaction window.

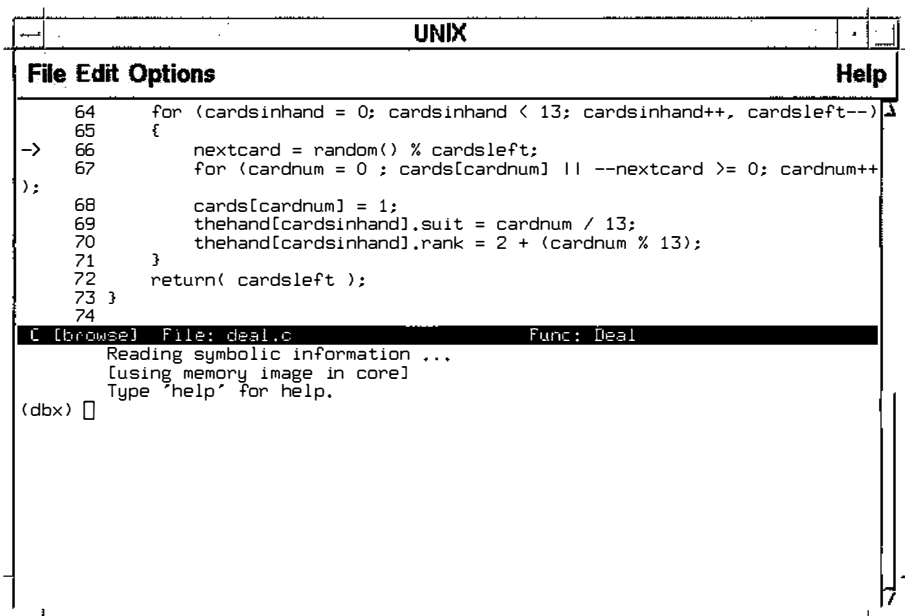


Figure 6-2 dbxtra interface

dbxtra is, by default, in screen mode as shown in Figure 6-2, “dbxtra interface”. It provides automatic segregation of the relevant debugging information into separate windows, or regions of the screen. As well, **dbxtra**’s screen mode has a configurable keyboard interface that lets users perform cursor movements, text scrolling, and command editing with a single keystroke.

dbxtra also has a line mode, which disables all screen functions as well as the configurable keyboard interface. It is invoked by specifying **dbxtra** with the **-s** option. The document focuses on **dbxtra**’s screen mode.

Screen mode

A window in **dbxtra** consists of one or more lines of the terminal display. Windows are separated by a separator line: a line displayed in reverse video (or highlighted) format.

Each window in **dbxtra** has associated with it one or more capabilities. A window capability describes the contents of that window and the functions that can be performed on it. The capability of a window is displayed at the extreme left of the separator line immediately below the window. The window at the bottom of the screen is always the interaction window, and since there is no separator line below that window, its capability is never displayed on the screen. The **dbxtra** windows are:

source	The source window displays the source file being debugged. An arrow (->) marker indicates the line to be executed next. This window has the following capabilities:
C	Enables display of the relevant C source being debugged. This capability is indicated by a "C" in the separator line.
C++	If the program being debugged is a C++ program, and C++ debugging information is available, the window contains the relevant C++ source of the program being debugged. This capability is indicated by a "C++" in the separator line.
browse	Enables display of other source files — typically through the use of the file , func , or show_xref command. This capability is indicated by the word "browse" in the separator line.
interaction	This window submits commands to the debugger and displays messages from the debugger. If the output window has not been created, this window also contains the output from the program being debugged. This window is always the bottom-most window on the screen and since no separator line is present, no indicator displays its capability.
output	This window displays the output from the program being debugged. The word "prog" indicates this capability. If this window is also the Interaction window, then "prog" does not appear because there is then no separator line for this window.

display This window displays the expressions (and their values) that have been specified through the **display** command. It can only be automatically created, if it does not already exist, when the **display** command is invoked. Expressions displayed in this window may be removed using the **undisplay** command. When no expressions remain in this window, the window itself is removed. "display" indicates this capability.

When invoked in screen mode (for example, without **-s**), the terminal display is initially partitioned into two windows — a program source window with C/C++ execution and browse capabilities, and an interaction window. It is possible to create a number of other windows to segregate the information to a greater extent. For example, Figure 6-3, "Sample dbxtra session" contains an example of a **dbxtra** screen with a number of the windows described above.

```

UNIX
File Edit Options Help
64   for (cardsinhand = 0; cardsinhand < 13; cardsinhand++, cardsleft--)
65   {
-> 66       nextcard = random() % cardsleft;
67       for (cardnum = 0 ; cards[cardnum] || --nextcard >= 0; cardnum++
;
68       cards[cardnum] = 1;
69       thehand[cardsinhand].suit = cardnum / 13;
70       thehand[cardsinhand].rank = 2 + (cardnum % 13);
71   }
72   return( cardsleft );
73 }
74

C [browse] File: deal.c                               Func: Deal                               Pid: 9138
1 deal.Deal.cardsleft = 0
2 cardsinhand = 0

display []
Dealing 1 hands...
Dealing 1 hands...
Dealing 1 hands...

prog [] a.out
(dbx) run

Floating point exception in Deal at line 66 in file "deal.c"
(dbx)

```

Figure 6-3 Sample dbxtra session

The source window at the top of the screen contains the C as well as browse capabilities. The display window is next, followed by the output window. Finally, the window at the bottom is the interaction window.

NOTE Neither the interaction nor the source window can be removed using the **close** command.

dbxtra keyboard interface

In order to use **dbxtra** screen functions (see “dbxtra screen functions” (page 202)), you must bind them to a key sequence using the **dbxtra bindkey** command.

The syntax of the command is:

```
bindkey [ key-sequence screen-function | > filename ]
```

The *key-sequence* can be any valid key designation defined in the **terminfo**(M) database. It can also be a combination of characters enclosed in double quotes (“”), including control characters as entered from the keyboard. Control characters can also be specified using the ^X notation or as an octal number using the \nnn notation. For example, the following two key definitions are equivalent:

```
(dbxtra) bindkey "[[A" "cursor_up"
(dbxtra) bindkey "\033[A" "cursor_up"
```

screen-function must be enclosed in double quotes (“”). Key sequences are interpreted as soon as they are typed in. If one sequence is a prefix of another, then it is interpreted first, effectively hiding the longer one.

If *key-sequence* is not specified, definitions of all *key-sequence* are printed. This output can be redirected to a file (using "> *filename*") so that the bindings can be used in subsequent invocations of **dbxtra**.

To obtain a list of currently bound key sequences, execute:

```
(dbxtra) bindkey
```

The directory */usr/lib/dbxtra/keys* contains key binding files for various terminal types. For example, to load the key bindings file for the ansi terminal, execute:

```
(dbxtra) source /usr/lib/dbxtra/keys/keys.ansi
```

To unbind a key bound with **bindkey**, use:

```
(dbxtra) unbindkey key-sequence
```

The syntax for *key-sequence* is the same as that for **bindkey**.

If an attempt is made to bind a key that is already bound, the first argument cannot be given verbatim unless key interpretation is disabled (using the **keys off** command).

To use a different set of bindings, make use of the contents of the **TERM** environment variables as the name of the key bindings file for each terminal

(for example, *key.at386* or *key.wy60*). Thus the correct file is automatically loaded if the following line is included in your *.dbxtrarc* file:

```
source /usr/lib/dbxtra/keys/keys.$TERM
```

If more than one *screen-function* is specified, each command must be separated by the ";" character. If no *screen-function* is specified, the current definition of the *key-sequence* is printed.

dbxtra screen functions

NOTE In order to use **dbxtra** screen functions you must bind them to a key sequence using the **dbxtra bindkey** command. See "dbxtra keyboard interface" (page 201).

The following is a listing and description of the various commands that can be specified in the *screen-function* parameter of the **bindkey** command:

bof Move cursor to the beginning of file. The screen is scrolled or redrawn as necessary. If the current window is not a source window (for instance, if it was the interaction window), then the cursor is moved to the head of the buffer. The buffer contains up to 128 lines, so if more than 128 lines were written in this window the old lines are lost.

bol Move cursor to beginning of line (not including line numbers if in source window).

close Close the current window. If the current window is a source window, then another source window (the top neighbor by priority) is expanded, inherits the capabilities of the current window, and becomes the current window. If there is no other source window, the current window cannot be closed.

If the current window is the Output window, then the dbxtra session window (that is, the Interaction window) is expanded and inherits the Output capability. The Interaction window cannot be closed.

cursor_down

Move the cursor one line down. If the cursor is at the bottom of the window but not at the end of file, then the text is scrolled one line.

cursor_left

Move the cursor one character to the left.

cursor_right

Move the cursor one character to the right.

cursor_up

Move the cursor one line up. If the cursor is at the top of the window but not at the beginning of file, then the text is scrolled one line.

- del_bol** If the cursor is on a line being entered as a **dbxtra** command, then delete from the cursor location to the beginning of the command.
- del_char**
If the cursor is on the line being entered as a **dbxtra** command, then delete the character at the cursor location.
- del_word**
If the cursor is on the line being entered as a **dbxtra** command, then delete the word (string delimited by spaces) to the left of the cursor.
- eof** Move cursor to end of file (or buffer). It is a complementary function to **bof**.
- eol** Move cursor to end of line. If the current line is the line being entered as a **dbxtra** command, then the cursor is moved one column beyond the last character on the line.
- expand** Expand the current window by two lines. If the current window has two neighbors (above and below, not necessarily immediate neighbors) that are larger than one line, then each of them is shrunk by one line. Otherwise, the boundary between the current window and its neighbor is moved by two lines.
- expand_lower**
Expand current window by moving its lower edge by one line (provided that there exists a window that can be shrunk).
- expand_upper**
Expand current window by moving its upper edge by one line (provided that there exists a window that can be shrunk).
- goto_line**
Move cursor to line number. The user is prompted to enter the line number. This function only works for source windows.
- goto_mark**
Move cursor to the window location marked by **set_mark** (see below).
- half_page_back**
Scroll half page backward.
- half_page_forw**
Scroll half page forward.
- load_file**
Load file into current window. **dbxtra** prompts for the filename and this is entered from the keyboard. This command can only be used in a window with the **browse**, **C**, and **C++** capabilities.
- line_back**
Scroll one line backward.

line_forw

Scroll one line forward.

move_capability

Move a capability to the current window. The capability must follow the command. The supported capabilities are **browse** and **C**. When the debugging is in C++ mode, the C++ capability is also recognized.

next_command

Display the next line entered by the user to the **dbxtra** interaction window.

next_window

Change current window to the next window below the current one. If the current window is the bottom window, then the next window is the top window. An asterisk (*) is displayed on the left side of the separator line of the window the cursor is in.

page_back

Scroll one page backward.

page_forw

Scroll one page forward.

prev_command

Display the previous line entered by the user to the **dbxtra** interaction window.

prev_window

Change current window to the previous window (above). If the current window is the top window, then the previous window is the bottom window. An asterisk (*) is displays on the left side of the separator line of the window the cursor is in.

quote_char

Accept next character as is, that is, do not perform key interpretation on it.

redraw Redraw the screen.

screen_function

Prompt for the name of a screen function and invoke it.
This function can invoke screen functions that are not bound to keys.

send_edit

Display the argument as a command to **dbxtra** but do not execute the command so that it can be edited before being invoked. For example, suppose the string "send_edit stop in" is bound to the key "X". Then typing <Ctrl>X causes **dbxtra** to echo "stop in" and then wait for the user to complete the command.

send_echo

Display the argument specified as a command to **dbxtra** and also execute the command. For example, binding “^X” to “send_echo stop in main” causes **dbxtra** to echo and execute the command “stop in main” when (Ctrl)X is typed.

send_noecho

Execute the command specified but do not display the command in the interaction window.

set_mark

Set a mark in a window. This mark is used by **goto_mark** (see above).

show_xref

Before using **show_xref**, the **xref** command must be used first on some identifier. When this has been done, move the cursor (using **cursor_up**, for example) to an identifier in the output of **xref** then use **show_xref**. This displays in the source window the lines around the identifier the cursor is at.

shrink Shrink the current window by two lines. If the current window has two neighbors (above and below), then each of the neighboring windows is expanded by one line. Otherwise, the boundary between the current window and its neighbor is moved by two lines. A window cannot be shrunk to size zero.

split Splitting a source window creates a new source window with no capabilities and the new window becomes the current window. Splitting the **dbxtra** session window creates a program window.

toggle_follow_cursor

Toggle between the static portion of the full screen window and the portion of the window containing the cursor. This is only useful when the program window is being used.

toggle_full_screen

Toggle between the full screen window and the standard dbx/source window layout. Since this is a screen function, it can only be used when the program is not executing.

toggle_line_numbers

Toggle with/without line numbers in source windows.

toggle_more

Toggle paging for interaction window.

toggle_wrap_around

Toggle wraparound of long lines in source windows.

word_left

Move cursor one word (single non-alphanumeric character or a string of alphanumeric and underline characters) left.

word_right

Move cursor one word (single non-alphanumeric character or a string of alphanumeric and underline characters) right.

Some of the commands require arguments. In these cases, the string following the command name, until a “;” or the end of the line is encountered, is used as the parameter. If the “;” character is preceded by a “\” (backslash) character, it is interpreted as part of the parameter rather than a parameter terminator. Any number of *editing variables* can be specified in the parameter string. The following are the editing variables supported and their functions:

\$command

evaluates to the full **dbxtra** command at the cursor line. For example, the sequence “cursor_up; send_edit \$command” causes **dbxtra** to display the command at the previous line and wait for the user to edit it.

\$events

evaluates to the *event-ids* of breakpoints at the cursor line (applies to source windows only).

\$expr

evaluates to the expression at the cursor location. The expression is derived according to the following rules:

1. If the character at the cursor location is alphanumeric (or “_”), then all adjacent alphanumeric characters to the left are included.
2. Otherwise, if the character at the cursor location is the dereference operator (“*”), it is included along with any spaces on its right.
3. If the next character on the right is alphanumeric (or “_”), it is included.
4. If the next character on the right is “(” or “[”, then it is included along with all the characters up to (and including) the matching “)” or “]”.
5. If the next character on the right is the field operator (“.” or “->”), it is included.
6. If the next two characters on the right are “::” (the C++ class member operator) they are included.

Steps 3 through 6 are repeated until no more suitable characters are found.

\$file evaluates to source filename in the current window

\$keyboard

causes **dbxtra** to prompt the user with a ":" to enter a line and evaluates that line. For example, "move_capability \$keyboard" prompts the user for the name of the capability to be moved.

\$line evaluates to the source line number at the cursor location. For example, "send_echo stop at \$line" sets a breakpoint at the current cursor location.

\$var evaluates to the longest string of alphanumeric and "_" characters at the cursor position.

Sample key bindings

The following is a set of sample key bindings, a subset of those in the file called */usr/lib/dbxtra/keys/keys.at386*, distributed with the development system. They are known to work with the *at386 terminfo* entry included with your distribution of **dbXtra** and **dbxtra**. Use the **source** command to load them into your **dbxtra** session.

Key name/sequence	Editing command
key_up	"cursor_up"
key_left	"cursor_left"
key_right	"cursor_right"
key_down	"cursor_down"
"^[G"	"page_forw"
"^[I"	"page_back"
"^[E^[H"	"bof"
"^[E^[F"	"eof"
"^[H"	"bol"
"^[F"	"eol"
"^[C"	"close"
"key_f9"	"split; move_capability C"
"key_f10"	"split; move_capability C++"
"^[E^[E"	"next_window"
"^[E^[G"	"expand"
"^[E^[I"	"shrink"
"^[L"	"send_echo stop at \$line"
"^[E^?"	"send_echo delete \$events"
"^P"	"prev_command"
"^N"	"next_command"

dbxtra tutorial session

Using a sample program, designed for bidding in the card game “bridge”, we will explore some of the basic features of **dbxtra**.

Preparing files for dbxtra

Included on the media with your copy of **dbxtra** are files named *dealer.c*, *bidding.h*, and *deal.c* that you can use to participate in this session. These files are installed in the directory */usr/lib/dbxtra/demo*. Locate these files, copy them to a working directory, and compile them with:

```
$ cc -g dealer.c deal.c
```

NOTE To use **dbxtra** for source-level debugging, use the **-g** compiler option to place symbolic information, source line numbers, and variable names in the object file.

Execute *a.out*, and the system responds with:

```
a.out
Dealing 1 hands...
Illegal instruction - core dumped
$
```

NOTE The exact wording of the second line may vary from system to system. The message indicates that a *core* file, used for debugging, has been written to the current directory.

Starting dbxtra

The program that you compiled has a serious problem and we are going to find it. To debug the executable file called *a.out*, enter the following at the shell prompt:

```
$ dbxtra a.out
```

This starts the **curses**-based version of **dbxtra**. To initiate the **Motif** version, enter the following at the system prompt:

```
$ dbXtra a.out
```

In this tutorial session, only the **curses** version of the debugger will be described. Although the method of presentation is different, the functionalities of the commands in both versions are identical. You may, of course, specify the name of any executable file you wish to debug. If you do not specify the name of an executable file, **dbxtra** prompts you for one.

At startup, **dbxtra** informs you that it is reading in the symbolic information in the executable file and that it is using the memory image in the core file.

The dbxtra screen

The figure below resembles a window with two panes separated by a narrow horizontal bar. The top pane contains lines of C code from the source file *deal.c*, and an arrow is pointing at line 66 where the illegal instruction that caused the core dump is located. The horizontal bar identifies the file (*deal.c*) and the function name (**Deal**). This bar constantly displays the name of the active function and the source file in which it appears.

The bottom pane, which contains the (dbxtra) prompt, is where you interact with **dbxtra.**

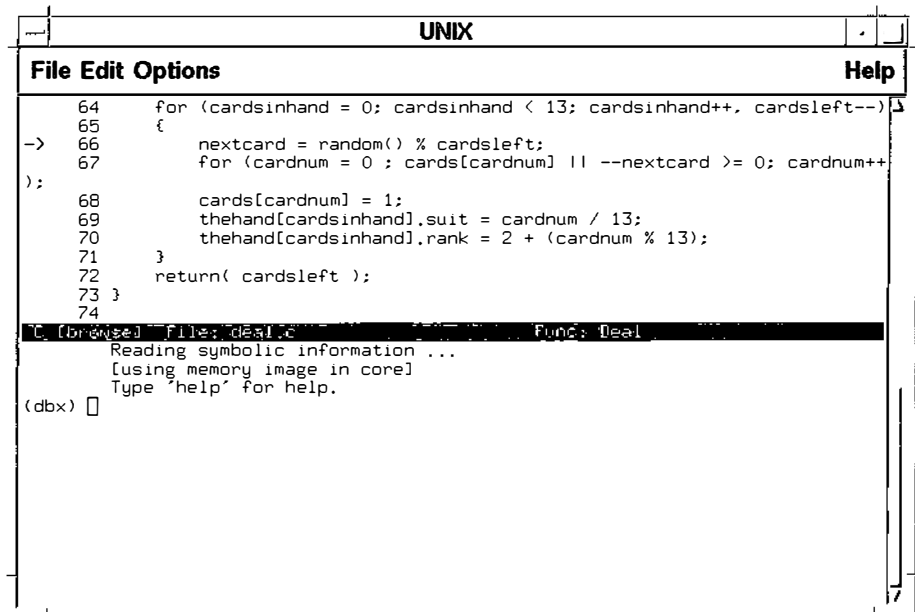


Figure 6-4 Display contents after executing dbxtra a.out

The dbxtra help facility

Above the `(dbxtra)` prompt, `Type help for help`, is displayed. Help can be invoked in two ways, first by typing **help** at the prompt, a general list of **dbxtra**'s commands are displayed. For a more detailed explanation of a specific command, invoke help with the **dbxtra** command name as an argument. For example:

```
(dbxtra) help forward
forward [n]
backward [n]
Scroll the source window forward and backward n lines
(the default is one window).
```

The screen's appearance may vary depending on your terminal type. **dbxtra** tries to take advantage of the emphasis modes available on each.

Printing the value of variables

After considering the line of code for a moment, you probably want to know the value of the variable **cardsleft**.

```
(dbxtra) print cardsleft
0
```

As suspected, the illegal instruction is the result of an attempt to divide by zero — by way of the modulus operator — but that is not the source of the problem. This variable should never have had a zero in it: we'll have to investigate why it did.

NOTE Besides simple variables like **cardsleft**, **dbxtra** can also print the contents of aggregates such as arrays and structures. Aggregates can contain many elements, and the bottom window is not large enough to display them all. **dbxtra** provides simple paging by stopping the output when the window is full and displaying a highlighted prompt `-- MORE --`.

Viewing source code and aliasing

From the previous figure, it is clear that **cardsleft** must have been assigned a zero before line 60. To see the source code that did this, we have to scroll the source code window back.

```
(dbxtra) backward
```

The **backward** command scrolls the source window back one full window. To scroll the window forward to see again how **cardsleft** is used, we use the command **forward**.

Typing these screen motion commands in full each time is tedious, so **dbxtra** provides the **alias** command to rename commands. For example, to rename the **backward** command to **b**, for example, type:

```
(dbxtra) alias b backward
```

dbxtra provides a set of default aliases. To view them, type **alias** itself at the (dbxtra) prompt.

Having moved back one screen, we see that **cardsleft** is a parameter of **Deal()**. Therefore, to see how it was passed the value zero, we have to find out which function called it. This can be done by looking at the *call stack*.

Viewing the call stack

To see all active functions within a program, use the **where** command to view the call stack.

```
(dbxtra) where
Deal(thehand = 0x4040e4, cards = 0x7ffffb04, cardsleft = 0), line 66 in "deal.c"
deal_hands(pdeal = 0x403f44), line 49 in "deal.c"
main(argc = 1, argv = 0x7ffffc18, 0x7ffffc20), line 55 in "dealer.c"
(dbxtra)
```

The **where** command lists all active functions, their arguments, the line number in which they stopped, and the source file in which they appear. Each function is called by the function that appears below it. In this case, **Deal()** was called by **deal_hands()**, which in turn was called by **main()**.

Traversing the call stack

You can see from the stack trace that **cardsleft** is passed to **Deal()** by **deal_hands()**. Let's go there:

```
(dbxtra) up
```

Notice that the source window has changed. The filename is still *deal.c*, but the current function is now **deal_hands()**. The window now shows the code from **deal_hands()** with line 49 marked with an arrow.

Monitoring the contents of variables

The **dbxtra** commands **when**, **trace**, and **stop** monitor the contents of a program's variables. The **trace** and **stop** commands are not as robust as the **when** command. For further details see "Events menu: managing execution time events" (page 215).

In our Bridge example, the variable **cardsleft** is passed to the function **Deal()**. It may be a good idea to have the value of **cardsleft** printed out every time the call to **Deal()**, at line 49, is made. To do this use the **when** command. At the (**dbxtra**) prompt, enter:

```
(dbxtra) when at 49 {print "at 49: cardsleft = ", cardsleft;}
[1] when at "deal.c":49 { print "at 49: cardsleft = ",
deal.deal_hands.cardsleft;}
```

dbxtra acknowledges your command, and automatically prepends the names of the blocks containing the variable to the variable name and the filename to the line number. Notice the **W** beside line 49; this indicates a **when** has been issued for this function.

To verify how a variable is monitored, use the **status** command:

```
(dbxtra) status
[1] when at "deal.c":49 { print "at 49: cardsleft = ",
deal.deal_hands.cardsleft;}
```

There are two things to note here:

- The symbol [1] is called an event-id. This is the first command that **dbxtra** need not execute immediately, but is kept in the background until a particular event occurs.
- **deal.deal_hands.cardsleft** is the *qualified name*, referring to the **cardsleft** variable. The qualified name indicates the respective scopes enclosing the variable. In this case, it refers to the **cardsleft** variable in function **deal_hands()** in *deal.c*. It is useful to ensure that the correct variable has been located, since a program can contain many variables with the same name.

Now let's run *a.out* to see what happens:

```
(dbxtra) run
Dealing 1 hands...
at 49: cardsleft = 52
at 49: cardsleft = 39
at 49: cardsleft = 26
at 49: cardsleft = 13
at 49: cardsleft = 0
Floating point exception in Deal at line 66 in file "deal.c"
```

Analyzing the results

You are back where you started, at line 66 in *deal.c*, and at this point it helps to know two things about the game of Bridge:

- Bridge is a game for four players.
- Fifty-two cards are dealt.

Each time a player receives cards, there are 13 cards less in the pack to be dealt. From the output produced by **when**, we can see that at each line after the first, the number of cards is reduced by 13. Therefore, we conclude that **cardsleft** indicates the cards remaining after players receive cards, and that each time a line is printed, the program is about to deal cards to a player. Since five lines were printed, the program was trying to distribute cards to five players, clearly an error. Hence, it is likely that there is a loop somewhere that is out of bounds.

Examining a different function

Let's go back and examine the values in **deal_hands()**:

```
(dbxtra) func deal_hands
```

The code in the source window now includes the offending line:

```
for( player=0; player<=4; player++ )
```

The limit on the loop counter (**player**) should have been set to 3 instead of 4, so there is four iterations instead of five. How can we test this suspicion?

Using a breakpoint

One way to test our suspicion is to force the loop to iterate only four times. To do so we first need to stop the loop after it has iterated that many times. **dbxtra** allows us to do this by letting us set breakpoints within programs using the **stop** command.

```
(dbxtra) stop at 49 if player=3
[2] stop at "deal.c":49 if deal.deal_hands.player == 3
```


Line 49 is the last line of the loop after all statements within it have been executed and if **player** is 3, we know that it has iterated four times. Rerunning the program now produces:

```
(dbxtra) rerun
Dealing 1 hands...
at 49: cardsleft = 52
at 49: cardsleft = 39
at 49: cardsleft = 26
at 49: cardsleft = 13
[2] stopped in deal_hands at line 49 in file "deal.c"
```

To prevent the fifth iteration of the loop, we just need to increment **player** by one:

```
(dbxtra) assign player=player+1
(dbxtra) print player
4
```

The counter has been incremented, so there will not be another pass when you continue the program.

Continuing after a breakpoint

You may now continue the execution of the program using the **cont** command:

```
(dbxtra) cont

      *You*   LHO   Partner   RHO
Bidding  -----
Your Hand
Spades
Hearts
Diamonds
Clubs

Enter your bid:
```

Exiting dbxtra

Our suspicions were correct! We, therefore, need to modify the function **deal_hands()** in the file *deal.c*. To do so, we first need to exit **dbxtra**. Before we can do so, however, notice that the cursor is now positioned after the last line of the program output shown above, and the (dbxtra) prompt has not appeared. This is because *a.out* is still running, so we need to interrupt it before we can exit **dbxtra**. This can be done by pressing the user-defined interrupt key. On most UNIX systems, it is the <Del> key. Once the (dbxtra) prompt is displayed, enter **quit** to exit **dbxtra**.

dbXtra and dbxtra commands

This section describes the commands that can be used with both **dbXtra** and **dbxtra**. Although the Motif interface provides buttons and menus to access commands, the commands described here can be entered via the keyboard by using the **Dbxtra** option in the Events menu. The **dbxtra** command is shown between the “[” and the “] ” characters following the name of each button.

File menu: saving and quitting

This menu accesses commands to save configuration information and exit **dbXtra**.

Write Configuration File

Prompts for a filename, then creates a file containing **dbXtra** window geometry commands suitable for inserting in an *.Xdefaults* file.

Exit dbXtra

[dbxtra command: quit]

Exits **dbXtra**, after prompting to ensure that you really want to quit.

Events menu: managing execution time events

The Events menu accesses various commands associated with creating, removing and querying breakpoint and trace events.

Events

[dbxtra command: status]

Displays the currently active events (breakpoints, trace expressions, and so on).

Call

[dbxtra command: call *function(arguments)*]

Executes the object code associated with a named function, passing arguments to it at the same time. The number and types of arguments must match, except when **\$unsafecall** is set. Execution of the function halts if any breakpoints are encountered, and **dbXtra** reenters the command level.

When

[dbxtra command: **when arguments**]

Executes one or more **dbXtra** commands whenever one of the following events occurs during the execution of the program being debugged:

- A particular source line is reached.
- A function is called.
- A variable is accessed or changed.
- Some condition is now true.

The simplest form executes commands depending on some arbitrary condition:

```
when [inst] condition { command ; [ command ; ... ] }
```

Before each statement is executed, *condition* is evaluated and if it is true, the specified command(s) are executed. If the **inst** keyword is specified, then *condition* is evaluated before each instruction is executed. If a debug register is available, it monitors the state of the *condition*.

```
when at line-number [if condition] { command-list }  
when inst at address [if condition] { command-list }
```

The first form of this command causes the specified command(s) to be executed whenever *line-number* line is executed. The second form causes the command(s) to be executed whenever the instruction at the specified *address* is executed.

In both cases, if the optional **if** clause is specified, the *condition* is evaluated before the specified *line-number* or *address* is executed, and the command(s) are executed only when the *condition* evaluates to true.

```
when in routine [if condition] { command ; [ command ; ... ] }
```

The specified command(s) are executed before the first statement in *routine* is executed. If the optional **if** clause is specified, then *condition* is evaluated and the command(s) are executed only if the condition evaluates to true.

```
when [inst] change variable [if condition] { command-list }  
when [inst] change address [if condition] { command-list }
```

If the first form is used, the specified commands are executed whenever the contents of the specified *variable* are modified. If the second form is used, the commands are executed whenever any of the four bytes starting at *address* are modified.

In either case, if the **inst** keyword is specified, the commands are executed before the next instruction (as opposed to the next statement) is executed.

If a debug register is available, it monitors changes to the specified *variable* or *address*. Otherwise, the contents of *variable* or *address* are monitored after each statement (or instruction) is executed.

```
when [inst] access variable [if condition] { command-list }
when [inst] access address [if condition] { command-list }
```

If the first form is used, the specified commands are executed whenever the contents of *variable* are accessed. If the second form is used, the commands are executed whenever any of the four bytes starting at *address* are accessed.

In either case, if the **inst** keyword is specified, the commands are executed before the next instruction (as opposed to the next statement) is executed.

NOTE This format for the **when** command can only be used if a debug register is available. If no debug registers are available, an error message is reported and the commands do not have any effect.

Stop

[dbxtra command: **stop arguments**] Sets breakpoints within a program. Sub-menu provides access to the various modes of the **stop** commands, which cause program execution to stop depending on whether:

- A particular source line is reached.
- A function is called.
- A variable is accessed or changed.
- Some condition is now true.

The syntax for the simplest form is:

```
stop [inst] if condition
```

This command stops the execution of the program being debugged before the next statement is executed whenever the Boolean condition specified evaluates to true. If the **inst** keyword is used, then execution stops before the next machine instruction is executed.

If a 386 debug register is available, it monitors execution. Otherwise, the Boolean condition is evaluated after each statement or instruction has been executed.

The next two forms allow for specific locations where execution is to be stopped:

```
stop at line-number [if condition]
stop inst at address [if condition]
```

If the first form is used, then execution of the debugged program stops before the statement on line *line-number* is executed. If the second form is used, then execution stops before the instruction at the specified *address* is executed.

If the optional **if** clause is used, then when execution reaches the specified line or address, the *condition* specified is evaluated and, if it evaluates to true, execution stops.

stop [inst] in routine [if condition]

Through this variant, the execution of the program being debugged stops before the *first* statement in the specified *routine* is executed. If the **inst** keyword is used, then execution stops before the first instruction in the specified *routine* is executed.

NOTE The first instruction is typically in the prologue section of the routine and may not correspond to the first instruction in the program source.

In either case, if the optional **if** clause is used, then the *condition* is evaluated before the first instruction or statement is executed, and the execution is stopped only if the *condition* evaluates to true.

stop [inst] [change] variable [if condition]

stop [inst] [change] address [if condition]

The execution of the program being debugged stops whenever the contents of *variable* are modified. If *address* is specified instead of *variable*, it is interpreted as an address in the data section of the program, in which case execution stops whenever any of the four bytes starting at that address are modified.

If the **inst** keyword is specified, then execution stops before the next instruction is executed (instead of before the next statement is executed). The **change** keyword is included for the sake of consistency with the other variants of the **stop** command, and its inclusion does not alter the behavior described.

In either case, if the optional **if** clause is used, then the specified *condition* is evaluated whenever the contents of the variable or memory location are modified, and execution stops only if the *condition* evaluates to true.

If a 386 debug register is available, it monitors execution. Otherwise, variable or memory locations are tested after each instruction or statement (depending on whether the **inst** keyword was specified) to determine if execution should be stopped.

stop [inst] access variable [if condition]

stop [inst] access address [if condition]

The execution of the program being debugged stops whenever the contents of *variable* are accessed.¹ If *address* is specified instead of *variable*, it is

1. This command is also triggered when variables are modified since variables have to be accessed to be modified.

interpreted as an address in the data section of the program, in which case execution stops whenever any of the four bytes starting at that address are accessed.

If the **inst** keyword is specified, then execution stops before the next instruction is executed (instead of before the next statement is executed).

In either case, if the optional **if** clause is used, then the specified *condition* is evaluated whenever the contents of the variable or memory location are modified, and execution stops only if the *condition* evaluates to true.

NOTE This command is only accepted if a 386 hardware debug register is available. If no such registers are available, an error message is displayed. To free up a debug register, you have to cancel a command that has been allocated a debug register. The **status** command can list all the commands that are active.

Trace

[dbxtra command: **trace arguments**]

Submenu provides access to the various modes of the **trace** commands, which can have **dbXtra** display diagnostic messages whenever one of the following events occurs during the execution of the program being debugged:

- A particular source line is reached.
- A function is called.
- A variable is accessed or changed.
- Some condition is now true.

As these commands continuously update the source window and/or display messages, it is useful to specify a time interval between updates or displays. This is achieved through the use of the **\$sleep dbXtra** variable, which is described in “Info menu: debugging registers, memory locations, and expressions” (page 226). The commands affected by the **\$sleep** variable are marked with a “+” symbol.

Each variant of **trace** has a trace region associated with it, which is the set of program statements or instructions executed while the specified events are monitored. By default, the trace region consists of all the statements (and instructions) in the program being debugged. A trace is triggered whenever a specified event occurs during the execution of the statements (or instructions) in the trace region.

This syntax for the simplest form of this command is:

trace [inst] [if condition] +

This command causes the source window to be updated so that the source

line pointer is placed at the line containing the next statement to be executed. If **inst** is specified, then each instruction is disassembled and displayed in the **interaction** window before it is executed.

If the optional **if** clause is specified, then *condition* is evaluated after each statement (or instruction) in the trace region is executed, and tracing information displayed only when *condition* evaluates to true. The trace region for this command is the whole program.

```
trace line-number [if condition]  
trace inst address [if condition]
```

The first form of this command causes a message to be displayed whenever *line-number* is executed. This line must be from the file displayed in the source window. The second form causes a message to be displayed whenever the instruction at *address* is executed.

In both cases, if the optional **if** clause is specified, the *condition* is evaluated before the specified *line-number* or *address* is executed, and if it evaluates to true, the trace message is displayed.

```
trace [inst] in routine [if condition]  +
```

This command is only slightly different from the first **trace** variant described above — it limits the trace region to the statements that are executed while the specified *routine* is active.

```
trace routine [in routine2] [if condition]
```

A message is displayed every time the specified *routine* is called, and also every time the routine returns. If the optional **in** clause is used, then the trace region is limited to those statements executed while *routine2* is active.

If the optional **if** clause is specified, then the *condition* is evaluated whenever the trace is triggered, and trace information is displayed only if the *condition* evaluates to true.

```
trace [inst] [change] variable [in routine] [if condition]  
trace [inst] change address [in routine] [if condition]
```

If the first form is used, a message is displayed whenever the contents of *variable* are modified. If the second form is used, then the message is displayed whenever any of the four bytes starting at the specified *address* are modified.

In either case, if the **inst** keyword is specified, the message is displayed before the next instruction (as opposed to the next statement) is executed. Please note that in the first form of the command, the **change** keyword is optional and is included for consistency with the other variants of the **trace** command; its inclusion does not alter the behavior described.

If the “*in routine*” clause is specified, the trace region is limited to the statements (instructions) executed while the specified *routine* is active. If the optional *if* clause is specified, then whenever the trace is triggered, the specified *condition* is evaluated, and the message is displayed only if the condition evaluates to true.

If a debug register is available, it monitors changes to the specified *variable* or *address*. Otherwise, the code in the trace region is single stepped, and the contents of *variable* or *address* are monitored after each statement (or instruction) is executed.

trace [*inst*] **access** *variable* [*in routine*] [*if condition*]

trace [*inst*] **access** *address* [*in routine*] [*if condition*]

If the first form is used, a message is displayed whenever the contents of *variable* are accessed. If the second form is used, the message is displayed whenever any of the four bytes starting at the specified *address* are accessed.

In either case, if the *inst* keyword is specified, the message is displayed before the next instruction (as opposed to the next statement) is executed.

If the *in* clause is specified, the trace region is limited to the statements (instructions) executed while the specified *routine* is active. If the optional *if* clause is specified, then whenever the trace is triggered, the specified *condition* is evaluated, and the message is displayed only if the condition evaluates to true.

NOTE This command is only accepted if a 386 hardware debug register is available. If no such registers are available, an error message is displayed. To free up a debug register, you have to cancel a command that has been allocated a debug register. The **Events** menu button can list all the commands that are active.

Interrupt

Stops the execution of the program; the source window is updated to display the current execution point.

Catch

[dbxtra command: **catch** [*signal-name* | *number*]]

Catches signals destined for your program; upon receipt of one of these signals, stops your program. You may elect to send the signal to the program through the **Continue** menu button. The **catch** command is useful when a program being debugged does not handle signals such as interrupts. A signal can be specified by a number or by a name. Signal names are case insensitive and the **SIG** prefix is optional. By default, all signals are caught except **SIGCONT**, **SIGCHILD**, **SIGALRM**, and **SIGKILL**. For more information on signals, see **signal(S)**.

Ignore

[dbxtra command: **ignore** [*signal-name* | *number*]]

Deletes a signal from the list of signals intercepted by the debugger. The **ignore** command specifies that the debugger send the specified signals on to the program being debugged. This command is useful when a program being debugged handles signals such as interrupts. A signal can be specified by a number or by a name. Signal names are case insensitive and the SIG prefix is optional. By default, all signals are caught except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL.

Dbxtra

Allows you to enter complex commands for direct execution by **dbx**. The output from the debugger is displayed in a separate, read-only window that can be removed when no longer required.

eXecute menu: executing the program being debugged

The **eXecute** menu controls the execution of your program.

Run

[dbxtra command: **run** *arguments*]

Starts execution of the program being debugged, clearing the current stack and all variables from memory. The syntax is:

run [*args*] [< *filename*] [>[>] *filename*]

args is always treated as a new list of arguments, replacing any that came before. If < *filename* is present, then it directs input from a file to the program. > *filename* directs output from the program to a file, while >> *filename* appends the output to a file.

Rerun

[dbxtra command: **rerun** *arguments*]

Restarts a program that was executed with **run**. Its syntax is:

rerun [*args*] [< *filename*] [>[>] *filename*]

If any arguments *args* are passed to **rerun**, they are appended to the arguments passed to **run** before being passed on to the program being restarted. If < *filename* is present, then it directs input from a file to the program. > *filename* directs output from the program to a file, while >> *filename* appends the output to a file.

Continue

[dbxtra command: **cont** [*number* | *signal-name*]]

Restarts execution of the program being debugged from where it was suspended (from the current breakpoint or event). You can optionally specify

that a signal be sent to the program upon resumption of execution. If a signal is specified, the function continues as though it received the signal. Otherwise, the execution of the function is resumed as though it had not been stopped.

Skip

[dbxtra command: **skip** *[number-of-events]*]

Resumes execution and ignores a specified number of breakpoints; the target program and ignores *n*, a given number of breakpoints (default is 1).

Next

[dbxtra command: **next** *[n]*]

Executes one or more source lines (default is 1), then stops. More than one program statement can be in a source line. When a function call is encountered, it is treated as a single statement. Execution of the function is only stopped if there is a breakpoint set at a statement executed by the function or by any function that it calls. **next** does not follow function calls but stops after a function call. Use **Step** to follow execution into called functions.

Step

[dbxtra command: **step** *[n]*]

Executes one or more source lines (default is 1) then stops. **step** follows the execution into called functions. Unlike **next**, the **step** command does not treat functions as single statements. Instead, each statement in a function is treated as a separate statement.

Goto

[dbxtra command: **goto** *line-number*]

Inserts a temporary breakpoint at the selected source line, then continues execution of the program until it reaches that line. The **goto** command resumes execution of the program from where it was suspended until a specified line is reached. As intervening breakpoints are encountered, you may need to issue several **goto** commands before the specified line is reached.

Jump

[dbxtra command: **jump** *line-number*]

Continues execution of the target program at *<line-number>*, skipping (not executing) the intervening lines in the process. If a breakpoint is present at that point, it is ignored. The *<line-number>* must be in the function at the top of the call stack.

Return

[dbxtra command: **return** [*function*]]

Continues execution of the program being debugged from where it was stopped until control returns to a specified function. The source window displays the line from which the original function was called. If *function* is not specified, the calling function is assumed.

Instruction

Submenu provides access to single-stepping by machine instructions.

Next [dbxtra command: **nexti** [*n*]]

Executes one or more machine instructions (default is 1), then stops. If a call to a function is encountered, the call instruction as well as all the instructions until the return statement represent one instruction.

Step [dbxtra command: **stepi** [*n*]]

Executes one or more machine instructions (default is 1) then stops. **stepi** follows the execution into called functions.

Source menu: accessing source files

The Source menu navigates your program source code, accessing commands for browsing or scanning program sources.

Browse file

[dbxtra command: **file** [*source-file-name*]]

Switches to another source file in the source window. If no argument is specified, then the current *source-file-name* is displayed.

Source Path

[dbxtra command: **use** *directory-list*]

Specifies the list of directories to be used by dbXtra in locating source files. The specified list replaces the list currently being used to locate source files.

Search

/regular-expression[/]

?regular-expression[?]

Searches forward [/] or backward [?] for a given pattern within the currently displayed file. Regular expressions may be used in the search string, which defaults to the currently selected text within the source window.

Function

[dbxtra command: **func** [*function-name*]]

Prompts for a function within the current program, then displays the source text around the first executable statement within that function. The new function then becomes the *current function*. If no argument is specified, the name of the current function is displayed.

DbXref

[dbxtra command: **dbxref** *arguments*]

dbXref prompts for the **dbxref** command line arguments, then generates cross reference information for the current executable. The **dbxref** command generates detailed information about the nature of the programming elements used in the program sources. It is similar to the **cxref** command available on most UNIX systems and in fact, you may execute the **dbxref** command using the syntax described here from the UNIX shell or from within your *make files*.

The syntax for the command is:

dbxref [-i] [*options*] *files*

dbxref generates the cross-reference database for the specified *files*. If the **-i** flag is used, it generates the cross-reference information incrementally. In this case, *files* should only be the ones that contain new information to be put into the database. This assumes that the cross-reference database for the program already exists.

The *options* allowed are **-I**, **-D**, and **-U**, and these have the same meanings as those for the C compiler and preprocessor.

The database is saved in the file *programname.xref*, where *programname* is the name of the executable file being debugged. This can be overridden using the **-ofile** flag, where *file* is the name of the new file to contain the database.

Up

[dbxtra command: **up** [*n*]]

Moves up the call stack by a specified number of levels (default is 1). The source window changes to display the function that called the current function. This function now becomes the current function and is used in resolving identifiers. Variable scoping rules default to this function.

Down

[dbxtra command: **down** [*n*]]

Moves down the call stack by a specified number of levels (default is 1). The source window changes to display the function that was called by the current function. This function now becomes the current function and is used in resolving identifiers. Variable scoping rules default to this function.

Info menu: debugging registers, memory locations, and expressions

The Info menu accesses commands that examine and modify registers and memory locations, and examine expressions.

Show

[dbxtra command: **whatis** *arguments*]

[dbxtra command: **print** *arguments*]

Shows the type and value of a program variable or expression. For input, use selected text or enter the text. For further discussion, see **dbXtra**(CP). The **whatis** command obtains the declaration of a specified name. The syntax is:

whatis [enum | union | struct | class] *name*

The *name* can be qualified with block names. By default, *name* is assumed to be the name of a variable in the program. Any of the optional keywords can specify that *name* be used as the name of an aggregate rather than a variable.

NOTE The “()” characters can specify that *name* be used as the name of a function rather than a variable.

Where

Obtains a listing of the active functions and their arguments. The **Up** and **Down** Source menu buttons can change the current function to another one in the call stack.

Display

[dbxtra command: **display** *arguments*]

Adds a variable or expression specified by the *<selected-text>* or by the user to the Display window. Information is updated and displayed every time a stop point is reached. The Display window is created if it does not exist. All the scrolling capabilities available for other windows are also applicable to the Display window. If more expressions are specified than can be accommodated by the window, some of the expressions scroll off the screen. Their space is accessible by manipulating the scrolling region or by increasing the size of the Display window.

When used with the **trace**, **stop**, or **when** commands, the **display** command can constantly monitor the values of expressions. The syntax is:

display [*expression* [, *expression* ...]]

The values are updated whenever the prompt is printed. If no expression is specified, then a list of the current expressions being displayed is output to the Interaction window (**dbxtra**).

If *expression* evaluates to a (char *) data type — that is, if a string expression is specified — the maximum number of characters displayed will depend on the value of the \$stringlen variable, the default value of which is 512.

Assign

[dbxtra command: **assign** *assignment-statement*]

Assigns a value of an expression to a named variable specified by the selected text or by the user. The syntax is:

assign *variable* = *expression*

Which

[dbxtra command: **which** *identifier*]

Displays the full qualification of an identifier selected or entered by the user. dbXtra returns the name of the identifier prepended with the names of the enclosing scopes that the identifier is associated with.

Print

[dbxtra command: **print** *arguments*]

Displays the value of an expression. The syntax is:

print *expression* [, *expression* ...]

If *expression* evaluates to a (char *) data type — that is, if a string expression is specified — the maximum number of characters displayed depends on the value of the \$stringlen variable.

Xref

[dbxtra command: **xref** *identifier*]

The **xref** command displays cross-reference information for a specified *identifier*, a variable, function, or macro. The cross-reference information is displayed in up to three sections. A section containing a list of functions and file or line number pairs, where the identifier is referenced, is always displayed. The other two sections are displayed only if the specified *identifier* is a function or macro. The second section contains a list of functions or macros that are called by the specified *identifier*. The third section contains a list of global variables, as well as parameterless macros, which are referenced by the named function. In both of these sections, an “M” as the first character of a line indicates that the identifier listed on the line is a macro.

For example, the following is the sample output when a global variable is used:

```
(dbxtra) xref suit_strings
Referenced from:
    format_hand "deal.c": 102 107
    log_deal    "dealer.c": 329
```

Whereas, the following is the sample output when a function name is used:

```
(dbxtra) xref deal_hands
Referenced from:
    main      "dealer.c": 49
References functions/macros: "deal.c"
    Deal      45
    qsort      46
References variables/macros: "deal.c"
    card_compare 46
```

The bindable **dbXtra** editing command **show_xref**, as described in “dbxtra screen functions” (page 202), can list the sources surrounding each of the line numbers produced by the **xref** command.

Variables

[dbxtra command: **dump arguments**]

Displays the names and values of variables local to the current function. The syntax is:

dump [*function*] [> *filename*]

If *function* is not specified, then the current one is used. If the function is given as ".", then all active variables are dumped.

Registers

[dbxtra command: **regs**]

[dbxtra command: **fpregs**]

Displays the contents of the general and floating point hardware registers of the processor.

Memory

Displays the contents of program memory.

Options menu: controlling the debugging environment

This menu sets **dbXtra** options and the environment variables of the program being debugged.

C/C++

Switches between the C++ source code and the C code that is a result of the compilation process. This is useful for programmers interested in the C language equivalent of a C++ construct. The C++ program must be compiled with the **-dbxC** flag.

Since **dbXtra** is a C++ and C debugger, the C code that is produced during the compilation of a C++ program can be debugged with **dbXtra**.

Preferences

This submenu contains a number of **dbXtra** switches that can be toggled to suit user preferences.

Noclobber

[**dbxtra** command: **set/unset \$noclobber**]

If set, then redirecting the output of a command into an existing file fails.

Hex chars

[**dbxtra** command: **set/unset \$hexchars**]

When set, **dbXtra** prints out characters in hexadecimal.

Hex ints

[**dbxtra** command: **set/unset \$hexints**]

When set, **dbXtra** prints out integers in hexadecimal.

Hex offsets

[**dbxtra** command: **set/unset \$hexoffsets**]

When set, **dbXtra** prints out offsets from registers in hexadecimal.

Hex strings

[**dbxtra** command: **set/unset \$hexstrings**]

When set, **dbXtra** prints out character pointers in hexadecimal.

No debug registers

[**dbxtra** command: **set/unset \$nodebugregs**]

When set, **dbXtra** cannot attempt to use the 386 hardware debug registers. The **trace access**, **when access**, and **stop access** commands need this variable to be enabled to function. The hardware debug registers cannot detect access or modification performed by the underlying UNIX operating system (such as those resulting from **read** and **write** system calls), and it may be desirable to turn off use of the debug registers when debugging program segments that make many system calls.

NOTE If floating-point hardware is not available, the operating system must emulate floating-point operations, with the result that certain operations may not be detected by the debug registers.

No debug warnings

[**dbxtra** command: **set/unset \$nodebugwarn**]

When **dbXtra** attempts to use the 386 debug registers, and there are no free registers, a warning message is printed, and slow emulation is performed. If **\$nodebugwarn** is set, the warning message is suppressed.

PCC disassembly

[**dbxtra** command: **set/unset \$pccformat**]

Print bases

[**dbxtra** command: **set/unset \$printbases**]

Unsafe Assigns

[**dbxtra** command: **set/unset \$unsafeassign**]

When set, strict type checking between the two sides of an **assign** statement is turned off.

Unsafe Calls

[**dbxtra** command: **unset/set \$unsafecall**]

When set, strict type checking is turned off for arguments to subroutine or function calls (for example, in the **call** statement). The **\$unsafeassign** and **\$unsafecall** variables should be used only with great care, because they severely limit **dbXtra**'s usefulness for detecting errors.

Verbose disassembly

[**dbxtra** command: **set/unset \$machinecode**]

When set, the contents of disassembly output is changed: the "function:line number" field in this output is replaced with the hexadecimal bit pattern representing the instructions decoded.

Word wrap

[**dbxtra** command: **function wordwrap**]

Environment

Submenu contains buttons to set and unset environment variables accessible by the program being debugged.

Set [**dbxtra** command: **setenv name value**]

Modifies values of environment variables that are accessible by the program being debugged. For example, if the program's output is being redirected to another screen, then the **TERM** environment variable accessed by the program would contain a different value from that accessed by **dbXtra**. Set the value of the environment variable *name*, in the program being debugged, to a specific *value*.

Clear [**dbxtra** command: **unsetenv name**]

Nullifies the values of environment variables, which are accessible by the program being debugged.

The help menu

This menu contains various buttons providing online help information about the debugger.

curses interface commands

The commands in this section are all effective through the **curses** interface version (**dbxtra**) and, where indicated through **dbXtra**.

forward/backward

forward [*n*]

backward [*n*]

Scroll the source window forward and backward *n* lines (the default is one window). This command is also available through **dbXtra**.

keys

keys [**on** | **off**]

Toggle keyboard translation functionality of **dbxtra**. If neither **on** nor **off** is specified, **on** is assumed.

NOTE Keyboard translations are never attempted when the keyboard input is meant for the program being debugged.

more

more [**on** | **off**]

Toggle pagination of output to the interaction window. If neither **on** nor **off** is specified, **on** is assumed. Note that only the output from the current command is paginated. This means that a page break only occurs if the output from a command exceeds the number of lines in the Interaction window.

screen

screen [**on** | **off**]

The **screen** command Change between screen mode and line mode. If neither **on** nor **off** is specified, **on** is assumed. The screen command can “clear up” or redraw the screen if the screen has become contaminated with unexpected output.

NOTE The **screen** command does not work if the **-s** option has been used on the command line when **dbxtra** is invoked.

sh and !

sh *shell command*

! *shell command*

Pass a command line to the shell for execution. The **SHELL** environment variable determines which shell is used. The character “!” can be used instead of **sh**.

source

source *filename*

Read and execute **dbxtra** commands from a named file. Each line of the named file is read and interpreted by **dbxtra** as if it were entered from the standard input stream. When all the lines have been read, commands are again read from the standard input stream. Any input to the program being debugged is always read from the standard input stream.

undisplay

undisplay

undisplay *number*

undisplay [*expression* [, *expression* ...]]

The first form of the command removes all the expressions associated with the last line in the display window.

The second form of the command removes all the expressions associated with a specified *number* line of the display window. If the line does not exist, an error message is printed.

If the third form is used, the specified *expression* is removed from the first line in the display window in which the expression appears.

version

This command simply prints version information about **dbxtra**.

Default command aliases

dbxtra and **dbXtra** provide a set of default command aliases. They are:

Alias	dbXtra/dbxtra command
c	cont
d	delete
h	help
e	edit
l	list
n	next
p	print
q	quit
r	run
rr	rerun
s	step
st	stop
j	status
t	where
sc	screen
noscreen	screen off
di	display
ud	undisplay
f	forward
b	backward
stopi	stop inst
tracei	trace inst
wheni	when inst

To view aliases, type **alias** at the (dbxtra) prompt.

dbXtra and dbxtra expressions and conditions

Variables and constants can be combined with operators to form expressions and conditions. Expressions have a value that can be printed or assigned to a variable. Conditions have a Boolean value that can be used to issue **dbXtra** and **dbxtra** commands selectively.

Table 6-1, “Operators in expressions” (page 234) contains a list of operators that can form expressions for various debugging commands. If a C++ program being debugged has overloaded any of them, the overloaded operators are not used. Note that the semantics of the casting operator are slightly different from those of the C/C++ casting operator.

Identifier resolution

A variable in an expression must be one of the identifiers in the program being debugged, and **dbXtra** first checks to ensure that this is true before resolving it. Variable names are resolved by first using the static scope of the

current function. By default, this function is the one in which control has stopped in (for example, due to a breakpoint). It can be redefined to another function by using the **up/down** or **func** commands. If the name is not defined in the static scope, then the dynamic scope is used. If static and dynamic searches do not yield a result, another variable with the same name is used and a message is printed showing the fully qualified name. The name resolution procedure may be overridden by qualifying an identifier with the scopes containing it, for example, *sfile.variable*, where *sfile* is the name of the source file (without the .c) that contains the global *variable*.

dbXtra and **dbxtra** recognize class scope within C++ member functions. See also “C++ specific debugging features” (page 235).

Table 6-1 Operators in expressions

Operator	Description
. or ->	structure or union member
[]	array indexing
sizeof(<i>type</i>)	size in bytes of <i>type</i>
~	one's complement
!	logical not
& <i>identifier</i>	address of <i>identifier</i>
* <i>identifier</i>	dereference of <i>identifier</i>
(<i>type</i>)	type cast
*	multiplication
/	floating point division
div	integer division
% or mod	modulo (or remainder)
+	arithmetic addition
-	arithmetic subtraction
<<	bitwise left shift
>>	bitwise right shift
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to
&	bitwise and
	bitwise or
&& or and	logical and
or or	logical or

Types and casting

The types of expressions are checked before being processed by **dbXtra**. They can be overridden by using the syntax *type(expression)*. As a special case, *expression* can be cast to a pointer to a named type by using “&”, as in **&type(expression)**.

You can also use the casting format typical in C, **(type)expression**, to a limited extent. In such cases, the **type** specifier cannot be the name of a derived type but it can be a structure or union tag. For example, in the following program segment:

```
struct foo { int i; };
typedef struct foo bar;

main()
{
    bar x;

    ...
}
```

The debugger recognizes the following casts:

```
(struct foo) x
foo(x)
```

But it does not recognize:

```
(bar) x
```

Although the debugging and C/C++ casting operators have the same usage, they have different semantics. In particular, if a value is cast by **dbXtra** or **dbxtra** to be of another type, then the bits representing that value are treated by the debugger as being of the new type. Thus, for example, **(int) x**, where **x** is of type **float**, is not converted to an integer by discarding its fractional part. Instead, the bits representing its value are treated as an integer.

C++ specific debugging features

dbXtra and **dbxtra** facilitate debugging of C or C++ programs. This section details the features that apply to debugging programs written in the C++ programming language.

Usage requirements

The C++ option is assumed by default when invoking **dbXtra** and **dbxtra**. The proper C++ compiler flags must be used to compile the program. The following describes the debugger's behavior when **-C++** is used along with other SCO C++ compiler flags.

If the **-dbxC** flag was used to compile the C++ program, the compiler also generates the C source that is derived from the C++ program. This enables the debugger to display both C and C++ source (see “The dbXtra and dbxtra cc command” (page 237)), and to display comprehensive information about C++ constructs. Commands that take expressions are also able to recognize both C++ identifiers and the resulting C identifiers. These capabilities allow debugging both C and C++ source. This flag, however, does result in additional files being created and, therefore, requires more hard disk space.

If the **-dbx** flag was used, the compiler does not produce C source. Hence, C source cannot be displayed. Commands, however, can recognize both C++ identifiers and their C derivatives, called “mangled” names. C++ information produced by **dbXtra** or **dbxtra** when this flag is used is as good as when the **-dbxC** flag is used. Although additional files are created, their number is less than the number created when **-dbxC** is used. This flag is useful when C++ source alone is to be debugged.

Like **-dbx**, the compiler does not produce C source with the **-g** flag. The debugger cannot display C source, although C++ identifiers and their C derivatives are recognized by debugger commands. However, the very limited amount of C++ debugging information produced prevents **dbXtra** and **dbxtra** from displaying much about C++ constructs. Hence, among the three flags, **-g** is the most restrictive in terms of allowing the debugger to debug C++ programs.

Overloaded functions and operators

Overloaded functions and operators are recognized by **dbXtra** and **dbxtra**. If an overloaded function or operator is specified as an argument to a debugger command, all of its instances are listed and the user is asked to select those which should be used as arguments to the command. For example:

```
(dbxtra) whatis print
print is overloaded or ambiguous; do you mean
1 : card.Hand::print()
2 : card.CardGroup::print()
3 : card.Card::print()
Your choice (1-3/all/n,m,...) :
```

Responding with a digit to the prompt selects one of the instances of **print**. If some are required, then a comma separated list of digits is entered at the prompt. Otherwise, typing “all” selects all the instances.

An overloaded operator is specified as a function whose name is specified as **operator<op>**, where **<op>** is replaced by the actual overloaded operator. The exception to this is that the **new** and the **delete** operators are not prefixed with the **operator** string.

The dbXtra and dbxtra cc command

When C++ programs are debugged, this command toggles **dbXtra** between the C and C++ interaction modes.

In the C++ mode, variables as well as other identifiers produced by **dbXtra** commands appear as the C++ identifiers. In C mode, the identifiers appear as translated (or mangled) identifiers. Regardless of the interaction mode, either the translated or untranslated identifiers can be specified as arguments to any of the **dbXtra** commands.

If the C++ program was compiled using the **-dbxC** compiler flag, use of the **cc** command also toggles the appearance of the contents of the program source window between C++ source and C source. This option can be useful in examining the translation of specific code segments as well as in executing segments of the C code rather than the C++ code.

In addition to having either C++ or C source displayed in the source window, **dbxtra** permits viewing both C++ as well as the generated C source at the same time as shown in the next figure. Simultaneous C and C++ source viewing is not available in **dbXtra**.

```

xterm
58
59 __ct__8subclassFv ( & __1S , (struct base *)0 ) ;
60 }
61
62
63 /* the end */
~
C [classes.c] File: classes.c      Func: main      Pid: 23034
26
27 main () {
28     subclass S;
29 }
30
~
~
~
C++ [ ] File: classes.C      Func: main      Pid: 23034
(dbx)
(dbx)
(dbx)
(dbx)
(dbx)
(dbx)
(dbx)
(dbx)
(dbx) cc
C debugging only enabled.
(dbx) stop in main
[1] stop in main
(dbx) run
[1] stopped in main at line 59
(dbx)

```

Figure 6-5 Display C++ and C sources at the same time

This command is only effective for C++ modules in the program being debugged. If the program being debugged is a C program, **dbXtra** attempts to unmangle any identifiers that appear to be C++ identifiers.

Debugging child processes

ptrace(S) allows processes to be traced by processes other than their parent. **dbXtra/dbxtra** uses this feature to allow you to debug the child process when a process forks. On prior releases of the debuggers, when a debugged process forked, the child ran to completion. You can now control what happens by setting the **\$fork** variable. This command must be typed into the interaction window in one of the following ways:

set **\$fork**="parent"

This is the default, and acts like previous releases. The child runs to completion, while the parent remains under debugger control.

set **\$fork**="child"

The parent will be released from the debugger, and the debugger will take control of the child. The parent will run normally to completion. Note that the child will not completely terminate after an **exit(S)** call until you quit the debugger.

set **\$fork**="push"

The parent will be suspended, and a new debugger will take control of the child. When the child completes, and you quit the child's debugger, debugging of the parent will resume.

set **\$fork**="both"

For **dbXtra** only, a new instance of the debugger and screen interface will be created to debug the child. Both debugger and their controlled processes will be active simultaneously.

set **\$fork**="prompt"

dbx will prompt for one of the above choices after each fork.

The following comments apply to the set **\$fork**="both" option.

The debugger stops the child right after the fork, before it has executed any code. Be sure to set breakpoints in the child before you allow it to continue. If you rerun the child, the original child process exits with **SIGTRAP** as the terminating signal. This is the result the parent process will get from **wait(S)**. If the parent is in wait, the wait will return as soon as you say rerun. This is rarely what you want to do.

The parent continues running normally after the fork, although it is under the control of the debugger. If you want it to stop, be sure to set a breakpoint in the code the parent will run so that it stops.

It can be confusing to debug two processes at the same time. Using the set `$fork="push"` is simpler if it will do the job for you.

When the child exits, you are notified in the debugger message window. You must exit the child `dbXtra` before the child process actually terminates. If the parent is going to `wait(S)` for the child, the `wait(S)` will not complete until this happens.

If the child process has not executed an `exec(S)` system call, quitting the debugger (either from the menu, or a quit command) will cause the `dbx` controlling the parent process to exit as well. This causes the entire debugging session to terminate, and the parent `dbXtra` to issue an error message.

The workaround is to go to another window, and `kill(C)` (`SIGTERM` works) the child screen manager process, that is, the instance of `dbXtra` that is associated with the child process. This instance of `dbXtra` will have a `dbx` as its parent process.

This will cause the child and its debugger to terminate, and things will operate normally from there.

If the child has executed an `exec(S)`, it seems that you can quit the child debugger normally.

Attaching to a running process

`ptrace(S)` allows a process to take control of another process while it is running. This is subject to various security controls: you must own the process you are attaching to, you must be able to read its code file, and the "attacked" process must not have different effective and saved UIDs or GIDs.

`dbXtra/dbxtra` (and `dbx`) uses these features to allow you to debug a running process. This is done by specifying the PID of the target process on the command line instead of the name of a core file. Remember that you still have to specify the name of the object file.

The process will be stopped and the debugger will take control of the process. Note:

- Remember that signals sent to the process will be intercepted by the debugger. If you execute `cont` (continue), any signals that arrived while you were setting up your debugging session will be lost. You can set `dbXtra/dbxtra` to ignore certain signals, but signals that arrive before you set that up and issue the first `cont` will still be lost. Recall that you can deliver a signal to a process with a `cont signal-number` debugger command.

In particular, this applies to **alarm(S)**. If the process you attach to is sleeping in **alarm(S)** when you take control, and the alarm timer expires before the process is continued, the alarm signal will be lost. You should use the **cont** command to issue the alarm signal in this case.

Also remember that a **ptrace(S)** restriction is that only one signal can be delivered when a process is continued. If more than one signal was pending, the others will be lost.

- If you want the process to continue to run after you stop the debugger, you must issue the **detach** command. **dbXtra/dbxtra** will remove all breakpoints and issue the **ptrace detach** function to tell the operating system to turn the process loose. You will have no further control of the process.

Any changes you made (like assigning to variables) will remain in effect.

If you do not issue a **detach** command, the process will be killed when the debugger terminates. This prevents a possibly modified process from continuing when it should not.

- If **dbXtra/dbxtra** cannot attach to the desired process, it issues a warning message and starts an ordinary debugging session.
- Make sure you are in the source directory, or use **-I** arguments to specify the location of the source for the program.

Chapter 7

sdb: symbolic debugger

The **sdb**(CP) program is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source-language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and displayed in the correct format.

When executing, breakpoints may be placed at selected statements, or the program may be single-stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines, which give formatted printouts of structured data.

The **sdb** program only supports COFF files. To debug programs which use the ELF format, use **dbXtra** (page 193), **dbxtra** (page 193), or **adb** (page 255).

Using **sdb**

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of shell commands for debugging a core image is:

```
cc -g prgm.c -o prgm
prgm
Bus error - core dumped
sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred, causing a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *****, which shows that it is waiting for a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

Here **sdb** was called with one argument, **prgm**. In general, **sdb** takes three arguments on the command line:

1. The first argument is the name of the executable file that is to be debugged. It defaults to **a.out** when not specified.
2. The second argument is the name of the core file, defaulting to **core**.
3. The third argument is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory.

In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the **-g** option, **sdb** prints the function name and the address at which the error occurred.

The current line and function are set to the first line where execution stopped. (The default is the first executable line in **main**.) If **main** was not compiled with the **-g** option, **sdb** will print a warning message, but debugging can continue for those routines that were compiled with the **-g** option.

To see a more extensive example of **sdb** use, see “An sdb session” (page 251).

Printing a stack trace

It is often useful to obtain a listing of the function calls that led to the error. This is obtained with the **t** command. For example,

```
*t
sub(x=2,y=3)      [prgm.c:25]
inter(i=16012)    [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c) [prgm.c:15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file *prgm.c*. The **sub** function was called with the arguments *x*=2 and *y*=3 from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine with three arguments often referred to as *argc*, *argv*, and *envp*.

NOTE *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining variables

The **sdb** program can be used to display variables in the stopped program. A variable is displayed by typing its name followed by a slash.

```
*errflag/
```

This causes **sdb** to display the value of variable **errflag**. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the following to display variable **i** in function **sub**:

```
*sub:i/
```

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol ***** is used to match any sequence of characters of a variable name and **?**, to match any single character. Consider the following commands:

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with **x**, the second prints the values of all two-letter variables in function **sub** beginning with **y**, and the last prints all variables. In the first example, only variables accessible from the current function are printed. The last example, prints global program and **libc** variables. The following command displays the variables for each function on the call stack and **libc** variables.

```
**:/
```

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, place a specifier after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

- b** one byte
- h** two bytes (half word)
- l** four bytes (long word)

The length specifiers are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** format to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed. There are a number of format specifiers available:

- c** character
- d** decimal
- u** decimal unsigned
- o** octal
- x** hexadecimal
- f** 32-bit single-precision floating point
- g** 64-bit double-precision floating point
- s** assume variable is a string pointer and print characters starting at the address pointed to by the variable, until a null is reached
- a** print characters starting at the variable's address until a null is reached
- p** pointer to function
- i** interpret as a machine-language instruction

For example, the variable **i** can be displayed with the following to print out the value of **i** in hexadecimal:

```
*i/x
```

The **sdb** program also knows about structures, arrays, and pointers, so that all of the following commands work.

```
*array[2][3]/  
*sym.id/  
*psym->usage/  
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers.

NOTE As a special case, the following displays the structure pointed to by **psym** in decimal.

```
*psym[0]
```

Core locations can also be displayed by specifying their absolute addresses.

```
*1024/
```

The above command displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal, so the above command is equivalent to both:

```
*02000/
```

```
*0x400/
```

To print out the contents of **argv** in the stack trace example above:

```
*0x7fffff54/s
```

It is possible to mix numbers and variables so that the following refers to an element of a structure starting at address 1000:

```
*1000.x/
```

Where as the following refers to an element of a structure whose address is at 1000. For commands of the types `*1000.x/` and `*1000->x/`, the **sdb** program uses the structure template of the last structure referenced.

```
*1000->x/
```

The address of a variable is printed with `=`. The following displays the address of **i**.

```
*i=
```

Another feature whose usefulness will become apparent later is the following command which redisplay the last variable typed:

```
*./
```

Source file display and manipulation

The **sdb** program is designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided that perform context searches within the source files of the program being debugged and display selected portions of the source files. The commands are similar to those of the UNIX system text editor, **ed**(C). Like the editor, **sdb** has a notion of current file and line within the current file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function.

Displaying the source file

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and determining the context of the current line. The commands are as follows:

p	prints the current line
w	window; prints a window of ten lines around the current line
z	prints ten lines starting at the current line; advances the current line by ten
control-d	scrolls; prints the next ten lines and advances the current line by ten; used to display cleanly long segments of the program

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the current source file or function

The **e** command is used to change the current source file.

```
*e function
*e file.c
```

Either of above forms may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the current line in the source file

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line. There are two commands for searching for instances of regular expressions in source files. They are:

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression, and the second searches backwards. The trailing **/** and **?** may be omitted from these commands. Regular expression matching is identical to that of **ed(C)**.

The `+` and `-` commands may be used to move the current line forward or backward by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that, the following advances the current line by 15 and then prints ten lines.

```
*+15z
```

A controlled environment for program testing

After entering **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single-stepping. The **sdb** program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single-step through a function that has not been compiled with the **-g** option, execution proceeds until a statement in a function compiled with the **-g** option is reached. It is also possible to have the program execute one machine-level instruction at a time. This is particularly useful when the program has not been compiled with the **-g** option.

Setting and deleting breakpoints

Breakpoints can be set at any line in a function compiled with the **-g** option. The command format is as follows:

```
*12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source-file display commands. The second sets a breakpoint at the first line of **proc**. The last sets a breakpoint at the current line.

Similarly, breakpoints are deleted with the **d** command:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or a **d**, the breakpoint is deleted.

The **B** command prints a list of the current breakpoints and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command:

```
*12b t;x/
```

This causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell.

```
*r args
```

The above command runs the program with the given arguments as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases, after an appropriate message is printed, control returns to the user.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, for example:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that stopped the program back to the program.

This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

Will continue at line 17 of the current function. One use for this command is to avoid executing a section of code that is known to be bad. The user should not attempt to continue execution in a function different from that of the breakpoint.

The **s** command is used to run the program for a single statement. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when you are confident that the called function works correctly but are interested in testing the calling routine.

The **i** command is used to run the program, one machine-level instruction at a time, while ignoring the signal that stopped the program. Its uses are similar to the **s** command. There is also an **I** command that causes the program to execute one machine-level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print-structured data. There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions. (It executes the function and prints the value that it returns.) The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

The above functionality requires that the user link with the **-lg** option; if this is not the case, **sdb** will print a warning message.

Machine language debugging

The **sdb** program has facilities for examining programs at the machine-language level. It is possible to print the machine-language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying machine language statements

To display the machine language statements associated with line 25 in function **main**, use the command:

```
*main:25?
```

The **?** command is similar to the **/** command except that it searches backwards. The default format for printing text space is the **i** format, which interprets the machine-language instruction. The **control-d** command may be used to print the next ten instructions. Absolute addresses may be specified instead of line numbers by appending a **:** to them, so that the following example displays the contents of address 0x1024 in text space:

```
*0x1024:?
```

The command displays the instruction corresponding to line 0x1024 in the current function:

```
*0x1024?
```

It is also possible to set or delete a breakpoint by specifying its absolute address, so that the following sets a breakpoint at address 0x1024:

```
*0x1024:b
```

Manipulating registers

The **x** command prints the values of all the registers. Also, individual registers may be named by appending a **%** sign to their name. For example, on the 80386 the following displays the value of register **eax**:

```
*eax%
```

Other commands

To exit **sdb**, use the **q** command.

The **!** command (when used immediately after the ***** prompt) is identical to that in **ed(C)** and is used to have the shell execute a command. The **!** can also be used to change the values of variables when the program is stopped at a breakpoint.

```
*variable!value
```

This sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb session

An example of a debugging session using **sdb** follows. Comments (preceded by a pound sign, **#**) have been added to help you see what is happening.

Example 7-1 Example of sdb Usage (Page 1 of 3)

```
sdb myoptim - ../common      # enter sdb command
Source path: ../common
No core image
*window:b                    # set a breakpoint at start of window
0x80802462 (window:1459+2) b
*r < m.s > out.m.s           # run the program
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean(*func)(); {
*t                            # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()    [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endif)    [local.c:483]
yylex()    [local.c:229]
main(argc=0,argv=0xc00201bc,-1073610300)    [optim.c:227]
```

Example 7-1 Example of sdb Usage (Page 2 of 3)

```

*z                                     # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:  extern NODE *initw();
1462:  register NODE *pl;
1463:  register int i;
1464:
1465:  TRACE(window);
1466:
1467:  /* find first window */
1468:
*s                                     # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                     # step
window:1465:  TRACE(window);
*s                                     # step
window:1469:  wsize = size;
*s                                     # step
window:1470:  if ((pl = initw(n0.forw)) == NULL)
*S                                     # step through procedure call
window:1475:  for (opf = pf->back; ; opf = pf->back) {
*pl                                   # show variable pl
0x80886b38
*x                                     # print the 80386 register contents
  eax/ 1                             ecx/ 0x402e2c         edx/ 0xbffffca8
  eox/ 0x17                           esp/ 0xbffffc54         edp/ 0xbffffc60
  esi/ 0x16                           edi/ 0x15             eip/ 0x16b
  flag/ 0x13296                       trap/ 0xe           err/ 7
0x16b (main+4):  lea  -8(%ebp),%edi  [0x7fffffff]

```

Example 7-1 Example of sdb Usage (Page 3 of 3)

```

*pl[0]                                # dereference the pointer
pl[0].forw/ 0x80886b6c
pl[0].back/ 0x80886ac8
pl[0].ops[0]/ pushw
pl[0].unqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]                          # dereference the pointer
pl->forw[0].forw/ 0x80886ca0
pl->forw[0].back/ 0x80886b38
pl->forw[0].ops[0]/ call
pl->forw[0].unqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw                          # replace pl with pl->forw
*pl                                  # show pl
0x80886b6c
*c                                  # continue
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean (*func)(); {
*s                                  # step
window:1459: window(size, func) register int size; boolean (*func)(); {
*s                                  # step
window:1465: TRACE(window);
*size                              # show function argument size
3
*D                                  # delete all breakpoints
All breakpoints deleted
*c                                  # continue
Process terminated
*q                                  # quit sdb
$

```


sdb: symbolic debugger

Chapter 8

adb: absolute debugger

adb(CP) is a source level debugger for C and assembly language programs. It provides a set of commands to examine, debug and repair executable binary files as well as examine non-ASCII data files. **adb** provides a method of looking at the contents of files in a variety of formats and structures. The following aspects of **adb** are outlined in this chapter:

- “Starting adb” (this page)
- “Displaying program instructions and data” (page 257)
- “Debugging program execution” (page 265) discusses running, setting breakpoints, and single-stepping through a program
- “Using the adb memory maps” (page 275)
- “Miscellaneous features” (page 278)
- “Patching binary files and memory” (page 283)

The **adb** program supports both the COFF and ELF executable file formats.

Starting adb

By using the following form, any executable C or assembly language program file can initiate the debugger:

adb [*filename*]

filename is the name of the file to be debugged. The type of file can be data, program or core. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. By initiating **adb** with the **-w** option, changes and corrections can be made in a program file. See “Patching binary files and memory” (page 283) for details. Once started, **adb** prompts with an asterisk (*) and waits for commands to be entered.

If the filename provided does not exist or is in the wrong format, **adb** displays an error message. For example:

```
adb sample
adb: cannot open 'sample'
```

adb can be started without a filename. In this case, **adb** searches for the default file *a.out* in the current working directory. The executable file *a.out* is created by the C compiler when a program is compiled and linked successfully. **adb** can be invoked by either of the following methods:

```
adb
or
adb a.out
```

Starting adb with a core image file

Core image files, resulting from fatal system errors can be examined through **adb**. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time the error occurred.

To examine a core image file with its corresponding program, provide both the core and the program file names. If no *corefile* name is provided, **adb** searches for the default core image file *core* in the current directory and uses it regardless of whether or not it belongs to the program. If the default core image file is used **adb** will issue a warning that the core image file does not match the object file. In the following example the *programfile* is the filename of the program causing the error and *corefile* is the filename of the core image file generated by the system:

```
adb programfile corefile
```

To prevent **adb** from opening the default core image file, use a hyphen (-) in place of the core filename. For example:

```
adb sample -
```

Exiting adb

To exit **adb** and return to the system shell use the **\$q** or **\$Q** command. To stop the debugger press <Ctrl>D.

NOTE **adb** cannot be stopped by pressing the Quit or Interrupt key. **adb** ignores Quit; Interrupt is caught by **adb** and causes it to wait for a new command.

Displaying program instructions and data

The **adb** program provides several commands for displaying the instructions and data of a given file.

```
address [, count ] = format
address [, count ] ? format
address [, count ] / format
```

In the above command lines *address* is a value or expression giving the location of the instruction or data item. *count* is an expression giving the number of items to be displayed, and *format* is an expression defining how to display the items. The equal sign (=), question mark (?), and slash (/) tell **adb** from what source to take the item for display.

Forming addresses

In **adb**, every address has the following form:

```
[ segment :] offset
```

In the above command line *segment* is an expression giving the address of a specific segment of 8086/286/386 memory, and *offset* is an expression giving an offset from the beginning of the specified segment to the desired item. Segments and offsets are formed by combining numbers, symbols, variables, and operators. The following are some valid addresses:

```
0:1
0x0bce:772
```

The *segment*: is optional. If not given, the most recently typed segment is used.

Forming expressions

Expressions contain decimal, octal, and hexadecimal integers, symbols, **adb** variables, register names, and a variety of arithmetic and logical operators.

Symbol names

A symbol is the name of a global variable or function defined within the program being debugged, and is equal to the address of the given variable or function. Symbols are stored in the program's symbol table, and are available if the symbol table has not been stripped from the program file.

Expressions that include references to a function can be evaluated by specifying the function's name or its symbol table name. Symbols in the symbol table are no more than 31 characters long. The following are examples of symbols:

```
main      _start  hex2bin __out_of
```

By using the question mark (?) command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Sometimes the command gives a function name when it displays data. This does not happen if the ? command for text (instructions) and the slash (/) command is used for data. Local variables cannot be addressed.

adb variables

adb automatically creates a set of its own variables whenever the debugger is initiated. These variables are set to the addresses and sizes of various parts of the program file as defined below:

Variable	Definition
b	base address of data segment
d	size of data
e	entry address of the program
m	execution type
n	number of segments
s	size of stack
t	size of text

A user can access storage locations using the **adb** defined variables.

The following request prints these variables:

```
$v
```

adb reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

To use the current value of an **adb** variable in an expression, precede the variable name with a less than (<) sign. For example, the current value of the base variable *b* is:

```
<b
```

To create or change the value of an existing variable by assigning a value to a variable name, use the greater than (>) sign. The assignment has the following form:

```
expression > variable-name
```

In the above command line *expression* is the value to be assigned to the variable, and *variable-name* must be a single letter.

For example, the following assignment gives the hexadecimal value "0x2000" to the variable *b*:

```
0x2000>b
```

To display the values of all currently defined **adb** variables, followed by their values in the current format, use the **\$v** command. The command displays any variable whose value is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise it is displayed as a number.

Current address

The **adb** program has two special variables that keep track of the last address to be used in a command and the last address to be typed with a command. The dot (.) variable, also called the current address, contains the last address to be used in a command. The double quotation mark (") variable contains the last address to be typed with a command. The dot and double quote variables are usually the same except when you use implied commands, such as the Newline and caret (^) characters. (These automatically increment and decrement dot, but leave " unchanged.)

Both the dot and the double quote can be used in any expression. The less than (<) sign is not required. For example, the following command displays the value of the current address:

```
. =
```

The following command displays the last address to be typed:

```
" =
```

Register names

adb allows the use of the current value of the CPU registers when evaluating expressions. To specify the value of a register, precede its name with the less than (<) sign.

For example, the value of the **eax** register can be evaluated in an expression by specifying the register as follows:

```
<eax
```

NOTE Register names cannot be used unless **adb** has been started with either a *core* file, or the program is currently being run under **adb** control.

Operators

Integers, symbols, variables, and register names can be combined with the following operators:

Unary	Meaning
~	Not
-	Negative
*	Contents of location
Binary	Meaning
+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise And
	Bitwise inclusive Or
^	Modulo
#	Round up to the next multiple

Unary operators have higher precedence than binary operators. Expressions are evaluated from left to right. All binary operators have the same precedence. Therefore, the following expression evaluates to 10:

$2 * 3 + 4$

whereas the following expression evaluates to 18:

$4 + 2 * 3$

To change the precedence of the operations in an expression use parentheses. For example, the following expression evaluates to 10:

$4 + (2 * 3)$

NOTE The **adb** program uses 32-bit arithmetic. Values that exceed 2,147,483,647 (decimal) are displayed as negative values.

The unary ***** operator treats an expression as a pointer to an address. An expression using this operator evaluates to the value stored at the given address. For example, the following expression evaluates to the value stored at the address "0x1234":

$*0x1234$

Whereas the following is just equal to "0x1234":

$0x1234$

Choosing data formats

Data of different forms can be displayed by specifying a string of format commands. A format command is a letter that specifies the format in which data is displayed. One or more letter commands can be concatenated with an integer to specify the number of times the letter commands are displayed.

The following partial list illustrates each letter command and associated data format displayed. For the complete list, see **adb(CP)**.

Letter	Format
o	2 bytes in octal
d	2 bytes in decimal
D	4 bytes in decimal
x	2 bytes in hexadecimal
X	4 bytes in hexadecimal
u	2 bytes as an unsigned integer
f	4 bytes in floating point
F	8 bytes in floating point
c	1 byte as a character
s	A null terminated character string
i	Machine instruction
b	1 byte in octal
a	The current symbolic address
A	The current absolute address
n	A Newline
r	A blank space
t	A horizontal TAB

A letter command can be used by itself or combined with other commands to present a combination of data in different forms.

Use **D** and **X** to display **long** variables or 32-bit values. The **f** and **F** commands can be used to display single- and double-precision floating-point numbers. The **c** command displays **char** type variables, and the **s** command is for arrays of **char** that end with a null character (null terminated strings).

The **i** command displays machine instructions in 8086/286/386 mnemonics. The **b** command displays individual bytes and is useful for displaying data associated with instructions, or the high or low bytes of registers.

To make the display more readable, combine the **a**, **r**, and **n** commands with other commands. For example, the following format commands display the current address after each instruction:

```
ia
```

To repeat each format, precede each format with a number. For example, the following format commands display four ASCII characters:

```
4c
```

Also, combine format requests to provide elaborate displays. For instance, the following commands display four octal words followed by their ASCII interpretation from the data space of the core image file:

```
<b,-1/4o4^8Cn
```

In this example, the display starts at the address "<b", the base address of the program's data. The display continues until the end-of-the-file since the negative count "-1" causes an indefinite execution of the commands until an error condition, such as the end of the file, occurs. The command **4o** displays the next four words (16-bit values) as octal numbers. The command **4^** then moves the current address back to the beginning of these four words and the **C** command redisplay them as eight ASCII characters. Finally, **n** sends a Newline character to the terminal. The **C** command causes values to be displayed as ASCII characters if they are in the range 32 to 126. If the value is in the range 0 to 31, it is displayed as an at sign (@) followed by a lowercase letter. For example, the value 0 is displayed as @. The at sign itself is displayed as a double at sign (@@).

Using the = command

The equal sign (=) command displays a given address in a given format. This is used primarily to display instruction and data addresses in simpler form, or to display the results of arithmetic expressions. For example, the following displays the absolute address of the symbol "main" (giving the segment and offset):

```
main=A
```

The following displays (in decimal) the sum of the variable *b* and the hexadecimal value *0x2000*:

```
<b+0x2000=D
```

If a count is given, the same value is repeated that number of times. For example, the following displays the value of "main" twice:

```
main,2=x
```

If no address is given, the current address is used instead. This is the same as the following command:

```
.=
```

If no format is specified, the previous format, specified for this command, is used. For example, in the following sequence, both “main” and “_start” are displayed in hexadecimal:

```
main=x
_start=
```

Using the ? and / commands

Use ? and / commands to display the contents of a text or data segment. The commands have the following form:

```
[ address ] [, count ] ? [ format ]
[ address ] [, count ] / [ format ]
```

In the above command lines *address* is an address with the given segment. *count* is the number of items to display, and *format* is the format of the items displayed.

Use the ? command to display instructions in the text segment. For example, the following command displays five instructions starting at the address “main”, and the address of each instruction displays immediately before it:

```
main,5?ia
```

The following command displays the instructions, with no addresses other than the starting address:

```
main,5?i
```

Use the / command to check the values of variables in a program, especially variables for which no name exists in the program’s symbol table. For example, the following command displays the value (in hexadecimal) of a local variable:

```
<ebp-4?x
```

Local variables are generally at some offset from the address indicated by the **ebp** register.

An example: simple formatting

The following example illustrates how to combine formats in `?` or `/` commands to display different types of values when stored together in the same program. This program has the following source statements:

```
char    str1[ ] = "This is a character string" ;
int      one      = 1 ;
int     number    = 456 ;
long    lnum      = 1234 ;
float   fpt       = 1.25 ;
char    str2[ ] = "This is the second character string" ;

main()
{
    one = 2;
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample -
```

To display the value of each individual variable by specifying its name and corresponding format in a `/` command. For example:

```
str1/s
```

displays the contents of *str1* as a string:

```
str1:  This is a character string
```

The following command:

```
number/d
```

displays the contents of *number* as a decimal integer:

```
number: 456.
```

You can choose to view a variable in a variety of formats. For example, you can display the **long** variable *lnum* as a 4-byte decimal, octal, and hexadecimal number by typing the following:

```
lnum/D
lnum:  1234
lnum/O
lnum:  02322
lnum/X
lnum:  0x4d2
```

You can also examine all variables as a whole. For example, if you wish to see them all in hexadecimal, type:

```
str1,5/8x
```

This command displays eight hexadecimal values on a line, and continues for five lines.

Since the data contains a combination of numeric and string values, it is worthwhile to display each value as both a number and a character to see where the actual strings are located. You can do this with one command by typing:

```
str1,5/4x4^8Cn
```

In this case, the command displays four values in hexadecimal, then the same values as eight ASCII characters. The caret (^) is used four times, immediately before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, insert a tab between the values and characters, and give an address for each line, for example:

```
str1,5/4x4^8t8Cna
```

Debugging program execution

The **adb** program provides a variety of commands to control the execution of programs being debugged. The following sections explain how to use these commands as well as how to display the contents of memory and registers.

Executing a program

Execute a program using the **:r** or **:R** command. The command has the following form:

```
[ address ] [ ,count ] :r [ arguments ]
[ address ] [ ,count ] :R [ arguments ]
```

In the above command lines *address* gives the address at which to start execution. *count* is the number of breakpoints to be skipped before one is taken, and *arguments* are the command line arguments, such as filenames and options, to be passed to the program.

If no *address* is given, then the start of the program is used. Thus, to execute the program from the beginning, type:

```
:r
```

If a *count* is given, **adb** will ignore all breakpoints until the given number has been encountered. For example, the following command causes **adb** to skip the first 5 breakpoints:

```
, 5:r
```

To specify arguments, separate each one by at least one space. The arguments are passed to the program in the same way the system shell passes command-line arguments to a program.

The **:R** command passes the command arguments through the shell before starting program execution. This allows the use of shell metacharacters in the arguments to refer to multiple files or other input values. The shell expands arguments containing metacharacters before passing them on to the program.

The **:R** command is especially useful if the program expects multiple filenames. For example, the following command passes the argument “[a-z]*.s” to the shell where it is expanded to a list of the corresponding filenames before being passed to the program:

```
:R [a-z]*.s
```

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This kills any previous copy of the program that may have been running.

Setting breakpoints

Breakpoints are typically set to stop program execution at a specific place in the program, such as the beginning of a function, so that the contents of registers and memory can be examined. To set a breakpoint in a program, use the **:br** command. Breakpoints cause execution of the program to stop when it reaches the specified address. Control then returns to **adb**. The command has the following form:

```
address [, count ] :br command
```

In the above command line *address* must be a valid instruction address. *count* is a count of the number of times the breakpoint is to be skipped before it causes the program to stop, and *command* is the **adb** command to be executed when the breakpoint is taken.

The following command sets a breakpoint at the start of the function named “main”:

```
main:br
```

The breakpoint is taken just as control enters the function and before the function’s stack frame is created.

A breakpoint with a count is typically used within a function that is called several times during execution of a program, or within the instructions that correspond to a **for** or **while** statement. Such a breakpoint lets the program continue to execute until the given function or instructions have been executed for the specified number of times. For example, the following command sets a breakpoint at the fifth repetition of the function “light”:

```
light,5:br
```

The breakpoint does not stop the function until it has been called at least five times.

NOTE No more than 16 breakpoints at a time are allowed.

Displaying breakpoints

To display the location and count of each currently defined breakpoint, use the **\$b** command. This command displays a list of the breakpoints given by address. If the breakpoint has a count or a command, these are given as well.

Continuing execution

To continue a program execution, after it has been stopped by a breakpoint, use the **:co** command. The command has the following form:

```
[ address ] [,count] :co [signal]
```

In the above command line *address* is the address of the instruction at which to continue execution. *count* is the number of breakpoints to ignore, and *signal* is the number of the signal to send to the program.

If an *address* is not specified, the program starts at the next instruction after the breakpoint. If *count* is specified, **adb** ignores the first *count* breakpoints.

Stopping a program with interrupt and quit

By pressing the INTERRUPT, (Ctrl)\, or Quit keys a program's execution can be stopped at any time. These keys stop the current program and return control to **adb**. The keys are especially useful for programs that have infinite loops or other program errors.

NOTE By pressing the INTERRUPT (Ctrl)\, or Quit keys, to stop a program, **adb** automatically saves the signal and passes it to the program if the program is re-started by using the **:co** command. To continue a program's execution without sending the signals, specify:

```
:co 0
```

0 prevents a pending signal from being sent to the program.

Single-stepping a program

To execute one instruction at a time, use the **:s** command. The command executes one instruction and returns control to **adb**. The command has the following form:

```
[address] [, count] :s
```

In the above command line *address* must be the address of the instruction to execute, and *count* is the number of commands to execute.

If *address* is not specified, **adb** uses the current address. If *count* is specified, **adb** continues to execute each successive instruction until *count* instructions have been executed. For example, the following command executes the first 5 instructions in the function *main*:

```
main,5:s
```

Killing a program

To kill the program currently being debugged, use the **:k** command. The command kills the process created for the program and returns control to **adb**. The command is typically used to clear the current contents of the CPU registers and stack and begin the program again.

Deleting breakpoints

To delete a breakpoint from a program, use the **:dl** command. The command has the following form:

```
address :dl
```

In the above command line *address* is the address of the breakpoint to be deleted. The **:dl** command is typically used to delete breakpoints no longer required. The following deletes the breakpoint set at the start of the function "main":

```
main:dl
```

Displaying the C stack backtrace

To trace the path of all active functions, use the **\$c** command. The command lists the names of all functions that have been called but have not yet returned control, as well as the address from which each function was called, and the arguments passed to it.

For example, the following command displays a backtrace of the C language functions called:

```
$c
```

By default, the **\$c** command displays all calls. To display a specific amount, supply a count of the number of calls required. For example, the following command displays up to 25 calls in the current call path:

```
,25$c
```

NOTE Function calls and arguments are put on the stack after the function has been called. By setting breakpoints at the entry point to a function, the function will not appear in the list generated by the **\$c** command. To remedy this, place breakpoints a few instructions into the function.

Displaying CPU registers

To display the contents of all CPU registers, use the **\$r** command. The command displays the name and contents of each register in the CPU as well as the current value of the program counter, and the instruction at the current address.

The **adb** program displays registers in the following format; the value of each register is given in the current default format:

eax	0x81000	efl	0x246
ebx	0x0	eip	0x142
ecx	0x0	cs	0x3f
edx	0x8	ds	0x47
edi	0x0	es	0x47
esi	0x0	fs	0x47
ebp	0x0	gs	0x47
esp	0x7fef8	ss	0x47
0x17:0x142:	push	ebp	

Displaying external variables

To display the values of all external variables in a program, use the **\$e** command. External variables are variables in a program that have global scope, or have been defined outside of any function. This may include variables defined in library routines used by your program.

adb: absolute debugger

For example, use the `$e` command to display the following external variables and their values in hexadecimal format in a program:

```
environ:      0xff08
fcnt:  0x0
gcnt:   0x0
hcnt:   0x0
errno:  0x0
_bufend: 0x0
```

An example: tracing multiple functions

The following example illustrates how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. The program to be examined has the following source statements:

```
int fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++ ;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++ ;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++ ;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}
```

The program is compiled and stored in a file named *sample*. To start the session, type:

```
adb sample
```

This starts **adb** and opens the corresponding program file. There is no core image file.

The first step is to set breakpoints at the beginning of each function. Use the **:br** command. For example, to set a breakpoint at the start of function “f”, type:

```
f:br
```

Use similar commands for the “g” and “h” functions. Once the breakpoints are created, display their locations by typing:

```
$b
```

This command lists the address, optional count, and optional command associated with each breakpoint. In this case, the command displays:

```
breakpoints
count  bkpt      command
1      h
1      g
1      f
```

The next step is to display the first five instructions in the “f” function. Type:

```
f,5?ia
```

This command displays five instructions, each preceded by its symbolic address:

```
f:      push  ebp
f+0x1:  mov   ebp, esp
f+0x3:  sub   esp, 0x8
f+0x9:  push  ebx
f+0xa:  push  edi
f+0xb:
```

Display five instructions in the “g” function without their addresses by typing:

```
g,5?i
```

In this case, the display is:

```
g:      push  ebp
        mov   ebp, esp
        sub   esp, 0x8
        push  ebx
        push  edi
```

To begin program execution, type:

```
:r
```

then **adb** displays the following message and begins to execute:

```
sample: running
```

As soon as **adb** encounters the first breakpoint (at the beginning of the “f” function), it stops execution and displays the following message:

```
breakpoint      f:      push    ebp
```

Since execution to this point caused no errors, you can remove the first breakpoint by typing:

```
f:dl
```

To continue the program, type:

```
:co
```

adb displays the following message and begins program execution at the next instruction:

```
sample: running
```

Execution continues until the next breakpoint, where **adb** displays the following message:

```
breakpoint      g:      push    ebp
```

To trace the path of execution, type:

```
$c
```

The commands show that only three functions are active: “f”, “main”, and “_start”:

```
f (0x1, 0x1)                                from main+0x15
main(0x1, 0x187ef20, 0x187ef28) from _start+0x39
```

The values 0x187ef20, 0x187ef28, and 0x39 will vary.

Although the breakpoint has been set at the start of function “g”, it will not be listed in the backtrace until its first few instructions have been executed. To execute these instructions, type:

```
,5:s
```

The **adb** program responds with a message indicating it has single-stepped the first five instructions. Now you can list the backtrace again. Type:

```
$c
```

This time, the list shows four active functions:

```
g (0x2,0x3)                                from f+0x2c
f (0x1,0x1)                                from main+0x15
main (0x1, 0x187ef20, 0x187ef28) from _start+0x39
```

To display the contents of the integer variable *fcnt*, type:

```
fcnt/D
```

This command displays the value of *fcnt* found in memory. The number should be 1. To continue execution of the program and skip the first 10 breakpoints, type:

```
,10:co
```

adb starts the program; then it displays the running message again. It does not stop the program until it encounters exactly ten breakpoints. It displays the following message:

```
breakpoint      g:      push      ebp
```

To show that these breakpoints have been skipped, display the backtrace again by typing:

```
$c
```

The system displays:

```
f (0x2,0x11)      from h+0x29
h (0x10,0xf)      from g+0x2b
g (0x11,0x20)     from f+0x2c
f (0x2,0xf)       from h+0x29
h (0xe,0xd)       from g+0x2b
g (0xf,0x1c)      from f+0x2c
f (0x2,0xd)       from h+0x29
h (0xc,0xb)       from g+0x2b
g (0xd,0x18)      from f+0x2c
f (0x2,0xb)       from h+0x29
h (0xa,0x9)       from g+0x2b
g (0xb,0x14)      from f+0x2c
f (0x2,0x9)       from h+0x29
h (0x8,0x7)       from g+0x2b
g (0x9,0x10)      from f+0x2c
f (0x2,0x7)       from h+0x29
h (0x6,0x5)       from g+0x2b
g (0x7,0xc)       from f+0x2c
f (0x2,0x5)       from h+0x29
h (0x4,0x3)       from g+0x2b
g (0x5,0x8)       from f+0x2c
f (0x2,0x3)       from h+0x29
h (0x2,0x1)       from g+0x2b
g (0x2,0x3)       from f+0x2c
f (0x1,0x1)       from main+0x15
main (0x1,0x187ef20,0x187ef28) from _start+0x39
```

Exit **adb** by typing:

```
$q
```

Using the adb memory maps

adb prepares a set of maps for the text and data segments in a program and uses these maps to access items that are requested for display. The following sections describe how to view these maps, and how they are used to access the text and data segments.

Displaying the memory maps

To display the contents of the memory maps, use the **\$m** command. The command has the following form:

\$m [*segment*]

In the above command line *segment* is the number of a segment used in the program.

The command displays the maps for all segments in the program using information taken from either the program and core files or directly from memory.

If **adb** is started but the program has not executed, the **\$m** command display has the following form:

Text Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0

Data Segments		File - 'sample'		
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000	0xe90	0x460	0x1880000

Each entry provides the segment number, file position, and physical size of a segment. The segment number is the starting address of the segment. The file position is the offset from the start of the file to the contents of the segment. The physical size is the number of bytes the segment occupies in the program or core file. The filenames to the right of the display are the program and core filenames.

If the program is executed, the command displays the following form:

Text Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x3f	0x400	0xb48	0xb48	0x0

Data Segments File - 'sample'				
Seg #	File Pos	Vir Size	Phys Size	Reloc Base
0x47	0x1000	0x1880e90	0x460	0x1880000

Virtual size is the number of bytes the segment occupies in memory. This size is sometimes different from the size of the segment in the file and will often change as the program is executed. This is due to expansion of the stack or allocation of additional memory during program execution. The filenames to the right always name program files. The file position value is ignored.

Giving segment numbers

By providing a segment number with the command, **adb** displays information only about that segment. For example, the following command displays a map for segment 63 only:

```
$m 63
```

Changing the memory map

To change the values of a memory map, use the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the following forms:

```
?m segment-number file-position size
/m segment-number file-position size
```

In the above command lines *segment-number* gives the number of the segment map to be changed. *file-position* gives the offset in the file to the beginning of the given address, *size* is the segment size in bytes, **?m** assigns values to a text segment entry, and **/m** assigns values to a data segment entry.

For example, the following command changes the file position for segment 0x3f in the text map to 0x2000:

```
?m 0x3f 0x2000
```

The following command changes the file position for segment 0x47 in the data map to 0x0:

```
/m 0x47 0x0
```

Creating new map entries

To create new segment maps and add them to a memory map, use the **?M** (create a text segment entry) and **/M** (create a data segment entry) commands. Unlike **?m** and **/m**, these commands create a new map instead of changing an existing one. These commands have the following form:

```
?M segment-number file-position size
/M segment-number file-position size
```

In the above command lines *segment-number* is the number of the segment map to be created and must be unique, *file-position* is the offset in the file to the beginning of the given address, and *size* is the segment size in bytes.

NOTE A new map entry cannot be created if that has the same number as an existing one.

The **?M** and **/M** commands are useful in accessing segments that are otherwise allocated to your program. For example, the following command creates a text segment entry for segment 0x47 whose size is 0x9c8 bytes:

```
?M 0x47 0x0 0x9c8
```

Validating addresses

adb checks each address to ensure it is valid. To validate the address, **adb** uses the segment number, file position, and size values in each map entry. If an address is correct, **adb** carries out the command; otherwise, it displays an error message.

The first step **adb** takes when validating an address is to check the segment value to make sure it belongs to the appropriate map. Segments used with the **?** command must appear in the text segments map; segments used with the **/** command must appear in the data segments map. If the value does not belong to the map, **adb** displays a bad segment error.

The next step is to check the offset to see if it is in range. The offset must be within the following range:

```
0 <= offset <= segment-size
```

If it is not in this range, **adb** displays a bad address error.

If **adb** is currently accessing memory, the validating segment and offset are used to access a memory location and no other processing takes place. If **adb** is accessing files, it computes an effective file address like the following, then uses this effective address to read from the corresponding file:

```
effective-file-address = offset + file-position
```


Miscellaneous features

The following sections explain a number of useful commands and features of **adb**.

Combining commands on a single line

To specify more than one command on a line, separate each commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format are carried to the next command. If an error occurs, the remaining commands are ignored. For example, the following command searches for and displays a string that begins with the characters *Th*:

```
/l 'Th'; ?s
```

Creating adb scripts

By redirecting **adb**'s standard input file at invocation, **adb** can be directed to read commands from a text file instead of the keyboard. To redirect the standard input, use the standard redirection symbol < and supply a filename. For example:

```
adb sample <script
```

The file supplied must contain valid **adb** commands. Such files are called script files, and can be used with any invocation of the debugger.

Scripts are typically used to display the contents of core files after a program error. For example, this file containing the following commands will display most relevant information about a program error:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,-1/8xna
```

Setting output width

To set the maximum width (in characters) of each line of output created by **adb**, use the **\$w** command. The command has the following form:

```
n$w
```

where *n* is an integer number giving the width in characters of the display. The default width, when **adb** is first invoked, is 80 characters.

The command is typically used when redirecting output to a line printer or special terminal. For example, the following command sets the display width to 120 characters, a common maximum width for line printers:

```
120$w
```

Setting the maximum offset

The **adb** program normally displays memory and file addresses as the sum of a symbol and an offset. This helps associate the instructions and data being viewed with a given function or variable. When first invoked, **adb** sets the maximum offset to 255. This means instructions or data that are no more than 255 bytes from the start of the function or variable are given symbolic addresses. Instructions or data beyond this point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason, **adb** lets you change the maximum offset to accommodate larger programs. To change the maximum offset, use the **\$s** command. The command has the following form where *n* is an integer giving the new offset:

```
n$s
```

For example, the following command increases the maximum possible offset to 4095:

```
4095$s
```

All instructions and data that are no more than 4095 bytes away are given symbolic addresses.

NOTE To disable all symbolic addressing, set the maximum offset to zero. All addresses will be given numeric values instead.

Setting default input format

To set the default format for numbers used in commands, use the **\$d** (decimal), **\$o** (octal), and **\$x** (hexadecimal) commands. The default format tells **adb** how to interpret numbers that do not begin with *0* or *0x*, and how to display numbers when no specific format is given.

The commands are useful when working with a combination of decimal, octal, and hexadecimal numbers. For example, if the following combination is used, hexadecimal addresses can be specified without prepending each with *0x*. Furthermore, **adb** displays all numbers in hexadecimal except those that are specifically requested to be in some other format:

\$x

adb's default format is decimal. To change or restore this at any time, use the **\$d** command.

Using UNIX commands

To execute UNIX commands without leaving **adb**, use the **adb** escape command **!**. The escape command has the following form:

! *command*

where *command* is the UNIX command to be executed. When finished, the shell returns control to **adb**. For example, to display the date, type:

! date

The system displays the date at your terminal and restores control to **adb**.

Computing numbers and displaying text

To perform arithmetic calculations while in **adb**, use the **=** command. This command directs **adb** to display the value of an expression in a given format.

Use the **=** command to convert numbers in one base to another, to double-check the arithmetic performed by a program, and to display complex addresses in easier form. For example, the following command displays the hexadecimal number "0x2a" as the decimal number 42:

0x2a=d

However, the following command displays it as the ASCII character asterisk (*):

0x2a=c

Expressions in a command may have any combination of symbols and operators. To compute the value of external symbols, type:

main+5=X

NOTE The `=` command can also be used to display literal strings. This is useful in an **adb** script where you may wish to display comments about the script as it performs its commands. For example, the following command spaces three lines:

```
=3n"C Stack Backtrace"
```

The system then displays the following message:

```
C Stack Backtrace
```

An example: directory and inode dumps

This example illustrates how to create **adb** scripts to display the contents of a directory file and the inode map of a UNIX file system. The directory file is assumed to be named *dir*, and contains a variety of files. The UNIX file system is assumed to be associated with the device file */dev/src*, and has the necessary permissions for you to read it.

To display a directory file, create an appropriate script, then start **adb** with the name of the directory, redirecting its input to the script.

First, create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned *inumber* and a 14-character filename. To display this information, add the following command to the script file:

```
0,5?ut14cn
```

This command displays one entry for each line, separating the number and filename with a tab. The display continues until the first five entries have been printed. If this command is placed at the beginning of the script, **adb** displays the strings as headings for the columns of numbers:

```
= "inumber"8t "Name"
```

Once you have the script file, type:

```
adb dir - <script
```

(The dash (-) is used to prevent **adb** from attempting to open a core file.) The **adb** program reads the commands from the script and displays the following:

```
inumber name
652      .
82       ..
5971     cap.c
5323     cap
0        pp
```

To display the inode table of a file system, you must create a new script, then start **adb** with the filename of the device associated with the file system (such as the hard disk drive).

The structure of an inode table entry is defined in the file */usr/include/sys/ino.h*. Each inode entry includes:

- an unsigned short containing the mode and type of the file
- a short containing the number of links to the file
- an unsigned short containing the owner's user ID
- an unsigned short containing the owner's group ID
- a long containing the size of the file in bytes
- an array of 40 bytes containing the disk block addresses (only 39 of the 40 address bytes are in use: 13 addresses of 3 bytes each)
- a long giving the time the file was last accessed
- a long giving the time the file was last modified
- a long giving the time the file was created

The inode table starts at the address "04000" of the filesystem. This is the address of the beginning of block 2 of the file system.

Display the first entry by typing:

```
04000,1?onororon2un40b3Y2na
```

Several Newlines are inserted within the display to make it easier to read.

To use the script on the inode table of */dev/src*, type:

```
adb /dev/src - <script
```

(Again, the dash (-) is used to prevent an unwanted core file.) Each entry in the inode table display has the following form:

```
IO.: 040755
      046    03      03
      640    0
      0121   06     0    0    0    0    0    0
      0      0     0    0    0    0    0    0
      0      0     0    0    0    0    0    0
      0      0     0    0    0    0    0    0
      0      0     0    0    0    0    0    0
      1986 Dec  4 13:55:49  1986 Nov 20 20:46:13  1986 Nov 20 20:46:13
```

Patching binary files and memory

To make corrections or changes to any file, including executable binary files, use the **w** and **W** commands and invoke **adb** with the **-w** option. The following sections describe how to locate and change values in a file.

Locating values in a file

To locate specific values within a file, use the **l** and **L** commands. The commands have the following form:

```
[ address ] [ ? / ]l value
[ address ] [ ? / ]L value
```

In the above command lines *address* is the address at which to start the search, the **l** command searches for 2 byte values, the **L** command searches for 4 byte values, and *value* is the value (given as an expression) to be located. **?** and **/** indicate whether to conduct the search in the text or data section.

The following command starts the search at the current text address, and continues until the first match or the end of the file. If the value is found, the current address is set to that value's address:

```
?l
```

Writing to a file

To write to a file, use the **w** and **W** commands. The commands have the following form:

```
[ address ] [ ? / ]w value
[ address ] [ ? / ]W value
```

In the above command lines *address* is the address of the value to be changed, the **w** command writes 2 byte values, the **W** writes 4 byte values, and *value* is the new value. **?** and **/** indicate whether to change the data in the text or data section. For example, the following commands change the data word "This" to "The":

```
/l 'Th'
/W 'The'
```

NOTE **W** is used to change all four characters.

Making changes to memory

Changes to memory, whenever a program has been executed, can also be made. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than the file. This is useful if you wish to make changes to a program's data as it runs, for example, to temporarily change the value of program flags or constants.

Chapter 9

C programmer's productivity tools

This chapter will teach you how to use the C Programmer's Productivity Tools (CPPT). Step-by-step instructions are provided through basic examples, so you can start using these tools right away. Additional examples demonstrate various options that allow you to make the best use of the tools.

The CPPT package consists of two tools: **cscope**, a browser, and **lprof**, a profiler.

The **cscope**(CP) browser is an interactive program that locates specified parts of code in a set of C source files and allows you to edit the files. It can reduce significantly the amount of time you must spend searching for functions, function calls, macros, and variables in the code. It is especially helpful for a programmer working on unfamiliar code.

A profiler is a tool for analyzing a program's run-time behavior, a procedure known as dynamic analysis. The **lprof**(CP) tool allows a programmer or tester to identify those parts of the source code that are most often executed and those that are never executed when a program is run. It provides line by line frequency profiling, reporting how many times each line of source code is executed. You can request test coverage analysis so that **lprof** only reports which lines of code are not executed at run time. **lprof** can be used over a set of tests such as a test suite.

lprof

There are two profilers available for dynamic analysis of C programs written in a UNIX System environment:

- the **prof** tool performs time-profiling; it reports how much time is spent executing various portions of a program
- the **lprof** tool performs line-by-line frequency-profiling; it reports how many times each line of source code is executed

This chapter explains the use of **lprof**; refer to the **prof**(CP) manual page for information on using **prof**.

Creating a profiled version of a program

If a program is to be profiled with **lprof**, it must be compiled with the **-ql** option to the **cc** command so that line count data will be saved. For example:

```
$ cc -ql travel.c
```

If you wish to create relocatable object files and link them later, you must specify **-ql** when you link as well as when you compile:

```
$ cc -ql -c travel.c
$ cc -ql -c misc.c
$ cc -ql -o travel travel.o misc.o
```

To profile an individual source file, rather than the source for the whole program, create a profiled version by using the **-ql** option when you compile the source file in question, and again when you link.

Running the profiled program

When you execute the program run-time data is stored in a data file whose name consists of the program name plus the extension **.cnt**. When the program ends, a message such as the following one is printed to **stderr**:

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' created
```

The PROFOPTS environment variable

The environment variable **PROFOPTS** provides run-time control over profiling, allowing you to override some of the default behavior. When the profiled program is about to terminate, it examines the value of **PROFOPTS** to determine how the profiling data is to be handled.

The **PROFOPTS** environment variable is a comma-separated list of options interpreted by the program being profiled. If **PROFOPTS** is not defined in the environment, then the default action is taken: the profiling data is saved in a file (with the default name) in the current directory. If **PROFOPTS** is set to the null string, no profiling data is produced.

The following options can be specified for **PROFOPTS**. They are explained in more detail in the examples.

msg =[<i>y</i> <i>n</i>]	If msg=y is specified, a message is printed to <i>stderr</i> stating that profile data is being created. If msg=n is specified, only profiling error messages are printed. The default is msg=y .
merge =[<i>y</i> <i>n</i>]	If merge=n is specified, data files are not merged after successive runs; the data file will be overwritten after each execution. If merge=y is specified, the data will be merged. The merge will fail if the program has been recompiled between runs; the data file will be stored in TMPDIR . The default is merge=n .
pid =[<i>y</i> <i>n</i>]	If pid=y is specified, the name of the data file will include the process ID of the profiled program. This allows the creation of different data files for programs calling fork(S) . If pid=n is specified, the default name is used. The default is pid=n .
dir = <i>dirname</i>	The data file is placed in the directory <i>dirname</i> if this option is specified. Otherwise, the data file is created in the directory that is current at the end of execution.
file = <i>filename</i>	<i>filename</i> is used as the name of the data file created by the profiled program if this option is specified. Otherwise the default name is used. (See "Profiling programs that fork" for an example.)

Examples of using PROFOPTS

The following examples show how **PROFOPTS** can be used to configure the environment in typical profiling situations.

Turning off profiling

If you do not want to profile a particular run, you can set **PROFOPTS** to the null string on the command line when you run a profiled version of a program:

```
$ PROFOPTS="" a.out
```

However, this value will remain in effect for only one execution of one program.

If you want to turn off profiling for more than one program or run, you must export the value of **PROFOPTS**:

```
$ PROFOPTS=""; export PROFOPTS
$ a.out
```

Exporting the variable eliminates the need to specify it every time you run the program. Once you have exported **PROFOPTS**, it keeps the value you have given it until you unset or redefine that variable. This makes the value of **PROFOPTS** accessible to all runs of any profiled programs.

Merging data files

Information collected from multiple executions can be collected into a merged data file. When data files created with the **lprof** compiling option are merged, the execution counts for all files are added together.

The following example shows how to configure the environment if you want data files from successive runs to be merged:

```
$ PROFOPTS="merge=y"
$ export PROFOPTS
$ travel
```

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' created
```

```
$ travel
```

```
dumping profiling data from process 'travel' . . .
CNTFILE 'travel.cnt' updated
```

Keeping data files in a separate directory

To avoid cluttering current directory, you may want to create a separate directory for data files. This directory must be specified on **PROFOPTS** For example:

```
$ PROFOPTS="dir=cntfiles" travel
```

All the data files will be created in the subdirectory *cntfiles*.

Profiling within a shell script

You may want to write a shell script to run profiled programs automatically. This facilitates specifying several **PROFOPTS** conditions for a particular program. The shell script in the next example implements the following conditions:

- no notification that profiling data is being created
- data merged automatically
- a name is specified for the data files

Here is the script:

```
PROFOPTS="msg=n, merge=y, file=test1.cnt" myprog < test1
```

Profiling programs that fork

If a program uses the system call **fork(S)**, the data files of the parent and child processes will have the same name by default. You can avoid this by using the **PROFOPTS** option **pid**. By setting **pid=y**, you ensure that the data file name will include the process ID of the program being profiled. As a result, multiple data files will be created, each with a unique name.

If you run a program that forks without setting **pid=y** option, then:

- if you have set **merge=y**, the data will be merged; data from separate processes will be indistinguishable
- if you have set **merge=n**, the last process to dump data will overwrite the data file

The following example shows how the **pid** option works. Notice names of the data files that are created (as reported by the messages sent to *stderr*):

```
$ PROFOPTS="pid=y" forkprog
```

```
dumping profiling data from process 'forkprog' . . .
CNTFILE '922.forkprog.cnt' created
```

```
dumping profiling data from process 'forkprog' . . .
CNTFILE '923.forkprog.cnt' created
```

Interpreting profiling output

You can use **lprof** to:

- produce source listing reports of profile data
- produce summary reports of profile data
- merge profile data files

Viewing the profiled source listing

Along with profiling information, **lprof** produces a source listing by default. Once you have executed your profiled program and the data file has been created, you can view the profile data by entering the following command:

```
$ lprof
```

The **lprof** output consists of a source listing with profiling information in the left margin, as shown in the following example:

```

                                #include <stdio.h>
                                main()
1 [3]      {
                                /* Note that declarations are not executable and
                                therefore have no line-number or execution
                                status associated with them. */

                                int i;

10 [10]      for (i=0; i < 10; i++)
10 [11]      sub1();
1 [12]      }

                                sub1()
10 [15]      {

                                /* The following declaration *is* an executable statement. */
10 [20]      int i = 0;

10 [22]      if (i > 0) {
                                /* The next line is never executed. */
0 [25]      sub2();
0 [26]      }
0 [27]      else {
10 [28]      sub3();
10 [29]      }
10 [30]      }

                                sub2()
0 [33]      {

                                /* do nothing */

0 [35]      }

                                sub3()
10 [38]      {

                                /* do same as sub2() */

10 [40]      }

```

The square brackets enclose line numbers for the file. Each number to the left of a line number shows how many times the corresponding source line was executed.

Showing unexecuted lines

If you use the **-x** option to **lprof**, the output highlights the lines that have not been executed. Lines that have been executed are marked only by line numbers. Lines that have not been executed are marked with a line number preceded by a [U]. The following example shows an example of output produced by the **-x** option:

```

#include <stdio.h>
main()
[3]      {
           /* Note that declarations are not executable and
           therefore have no line-number or execution
           status associated with them. */

           int i;

[10]         for (i=0; i < 10; i++)
[11]             subl();
[12]     }
    subl()
[15]     {
           /* The following declaration *is* an executable
           statement. */
[20]         int i = 0;
[22]         if (i > 0) {
               /* The next line is never executed. */
[U] [25]             subl();
[U] [26]         }
[U] [27]         else {
[28]             subl();
[29]         }
[30]     }
    subl()
[U] [33]     {
           /* do nothing */
[U] [35]     }
    subl()
[38]     {
           /* do same as subl() */
[40]     }

```

In any **lprof** output, certain lines (such as declarations, comments, and blank lines) do not have line numbers associated with them. This allows you to distinguish between lines that were not executed during a particular run from those that are not executable. In the previous example, neither line 24 nor line 25 in **sub1** was executed, but line 25 is marked with a line number while line 24 is not. This is because line 24 is not executable; line 25 is executable but was not executed in the run that produced this output.

Specifying program and data files to lprof

By default, **lprof** expects the profiled program to be called *a.out*, and the data file, *a.out.cnt*.

To run **lprof** on a program with a name other than *a.out*, specify the name after the **-o** option. For example:

```
$ lprof -o sample
```

lprof will assume that the data file is called *sample.cnt*.

You can specify a data file other than *sample.cnt* by using the **-c** option. For example:

```
$ lprof -c newdata.cnt
```

Files needed by lprof

The **lprof** tool must have access to three kinds of files in order to produce profiling information.

- the executable program (compiled with **-ql**)
- the data file associated with one or more program runs
- the source code files for the program being profiled
- Depending on which operation you request, **lprof** will not run if it cannot find one of these files.

When you run a profiled program, the name of the program is stored in the data file exactly as it appears on the command line. If you do not specify the **-o** option when you run **lprof**, it consults the data file to obtain the name of the program. Therefore, you may invoke **lprof**, specifying the name of the data file, and let **lprof** determine the name of the program. Because the name of the data file is not stored in the program itself, the reverse is not true: you cannot specify the name of the program and expect **lprof** to determine the name of the data file if it is not the default name.

The **lprof** tool will not be able to display data if you perform the following steps in the order shown:

1. use a relative path name for the executable file when you run your profiled program
2. run **lprof** from a different directory, specifying only the name of the data file (that is, without specifying the program name)

When you run **lprof** from a directory other than the one in which you have executed your profiled program, and you have used a relative path name when executing the profiled program, then you must specify the **-o** option to

lprof with either the profiled program's full pathname or the program's path-name relative to your current directory. The following example illustrates this:

```
$ cd $HOME
$ mybin/a.out
$ cd src
$ lprof -o ../mybin/a.out -c ../a.out.cnt
```

In this example, the **-c** is necessary because you are no longer in the directory where the program was run (and hence where the data file was created). The **-o** is necessary because otherwise **lprof** will look for an executable file called *mybin/a.out* in the current directory (*\$HOME/src*). It is assumed in this example that the source files are in the current directory.

Source files in a different directory

The names of the source files for the profiled program are also stored in the data file. However, only the file names, not the directories, are stored. Consequently when **lprof** searches for source files, it must assume that the source files are in the current directory. If they are in another directory, you must specify their location with the **-I** option and a path name.

The following example is a modified version of the previous one that demonstrates the use of **-I**. As before, assume that the source files are in the directory *\$HOME/src*.

```
$ cd $HOME
$ mybin/a.out
$ cd doc
$ lprof -o $HOME/mybin/a.out -c $HOME/a.out.cnt -I $HOME/src
```

In this example, **lprof** is run from the directory *\$HOME/doc*, but the source files are in *\$HOME/src*.

You can specify multiple **-I** arguments.

Source listing for a subset of files

If you want profiling output for a limited number of selected files, use the **-r** option:

```
$ lprof -r file1.c -r file2.c
```

This command line will produce output only for *file1.c* and *file2.c*. This is useful if you want to examine a few files rather than an entire program.

Summary option

You can obtain a summary report of the profile data by using the **-s** option:

```
$ lprof -s -c sample.cnt
```

Because a source listing is not produced with **lprof -s**, the **-r** and **-l** options do not need to be specified: **lprof** needs only to find the data file. The following example shows output produced with the **-s** option:

```
Coverage Data Source: a.out.cnt
Date of Coverage Data Source: Wed Sep 18 16:42:19 1991
Object: ./a.out

percent   lines   total function
covered  covered  lines name

100.0      4       4  main
 66.7      6       9  sub1
  0.0      0       2  sub2
100.0      2       2  sub3

70.6      12      17  TOTAL
```

This table gives the percentage of lines in each function that are actually executed during the program run.

Merging option

As described in the section “The **PROFOPTS** environment variable” data files can be merged automatically at run-time. You can also merge existing data files using the **lprof** command, as illustrated in the following example:

```
$ lprof -d destfile -m file1.cnt file2.cnt file3.cnt
```

The **-d** option is followed by the name of the file that will contain the merged data. The **-m** option is followed by the names of two or more data files to be merged. The data files must have been created by the same profiled program; if they have not, **lprof** will issue an error message:

```
$ lprof -d merged.cnt -m prog1.cnt prog2.cnt
ERROR: 'prog1', 'prog2'
      Object file entry names & timestamps don't match.
*** no merged output ***
```

You may have multiple data files created by the same program that have different time stamps. This will happen, for example, if you recompile a program. If you want to merge data from runs of different versions of the same program, you can override the time-stamp check by specifying **-T** (time stamp override).

NOTE You must be extremely cautious when using the `-T` option. If the control flow of the recompiled program has changed, the new merged data file is very likely to be erroneous, and `lprof` will produce an incorrect report.

Cautionary notes on using `lprof`

This section describes solutions for several problems that may arise while you are using `lprof`.

Trouble at compile time

Sometimes if you are compiling with one or more forms of optimization together with the profiling option, you will get an error warning that there is an incompatibility between obtaining profiling information and optimization. This is usually a permissible combination of options, but occasionally the compiler does not accept it.

You may not need to have the function in question profiled. If not, ignore this warning; data will be collected in the data file for all other functions. If you do want data for the function in question, compile your program again with the profiling option but without optimization. The warning should not reappear.

Non-terminating programs

If the profiled program does not terminate normally, no profiling data will be saved. The profiling data is saved at termination by the system call `exit(S)`. If `exit` is never called, no profiling data is saved.

Failure of data to merge

If a program has been recompiled, a new data file will be created in a temporary directory. The path name of the new file will be printed to `stderr`.

Trouble at the end of execution

At the end of execution, you may see the following error message:

```
dumping profiling data from process 'a.out' . . .  
***unable to seek to symbol table
```

Usually this is caused by running a stripped version of a profiled program. Never strip files to be profiled. If necessary, change "makefiles" so that they do not produce stripped files.

No data collected

Sometimes you may get no data after running a profiled program. The program terminates normally, and you receive neither a message about data being saved, nor an error message. This may be caused by one of two problems:

- You may not have specified **-ql** at both compile time and link time. If you forget to specify **-ql** when you link, the profiled program will run, but a data file will not be created.
- The profiled program may include a call to **_exit** that is causing the program to quit without calling **exit(S)**, the procedure that saves your profiling data. Replace calls to **_exit** with calls to **exit(S)** to save profiling data.
- The **PROFOPTS** variable may be set to **NULL**.

Data file not found

Occasionally, **lprof** may not be able to find the data file, despite the fact that the profiled program has terminated normally and you have received a message saying that the data file has been created.

The profiled program creates the data file in the directory in which the program is located when it terminates. If the program changes directories, the data file may be created in a directory different from both the one from which you executed the program and the one in which the shell is located when the program terminates.

Use the **dir** option of **PROFOPTS** to specify exactly where the data file is to be created, so you will be able to find it.

Using lprof with program generators

Program-generators such as **lex** and **yacc** make use of the **#line** preprocessor directive to change line numbering. This generally disrupts line profiling such as is done with **lprof**.

Using lprof with shared libraries

It is recommended that when profiling with **lprof**, you use archived versions of libraries rather than shared versions.

Improving performance with prof and lprof

The **prof** and **lprof** profilers can help a programmer locate the time-consuming parts of a C program.

prof provides a time profile, that is, a list of the most time consuming functions and the amount of time taken by each. The **lprof** tool provides a list of the lines that are being executed most frequently. Once these potential problem areas have been identified, it is the programmer's job to rewrite those parts of the code so that the program runs more efficiently.

Although either of these profilers can be used singly, they are most efficient if you use them together. First, profile your program with **prof** to identify the most time-consuming functions. Then, profile only those functions (rather than the entire program) with **lprof** to determine which lines are being executed most frequently. This two-step approach takes the guesswork out of determining which lines of code are the most time-consuming.

It is important to profile programs with data that are typical of what the program will encounter in normal use. Most test cases fail to provide profiling data representative of typical usage.

Improving test coverage with lprof

It is difficult to write test suites that fully exercise programs if you have no way of determining how much of the code is exercised. The **lprof** tool removes the guesswork by showing which lines of code are executed. This allows the tester to know exactly what has been tested. It also makes it easier to refine and improve tests.

To measure how well a given test suite tests a program, profile the program and look at the summary output, to see how much of the code is exercised. More specifically, you can examine individual functions that do not have 100% coverage (that is, not all lines in the function were executed). This analysis may suggest ways of improving the tests.

The following two examples illustrate why certain functions may not have 100% coverage. The first example demonstrates how to uncover a feature that is usually missed in the test suite. The second example shows how to uncover a function that is never called.

Example 1: searching for untested features

First, examine a source listing to see what parts of the code are not executed. The portion of code profiled in the following example processes command-line options.

```

while((c=getopt(argc, argv, "blcnsvi")) != EOF)
    [32]         switch(c) {
                        case 'v':
[U] [34]                 vflag++;
                        break;
                        case 'c':
[37]                 cflag++;
                        break;
                        case 'n':
[40]                 nflag++;
                        break;
                        case 'b':
[43]                 bflag++;
                        break;
                        case 's':
[46]                 sflag++;
                        break;
                        case 'l':
[49]                 lflag++;
                        break;
                        case 'i':
[52]                 iflag++;
                        break;
                        case '?':
[U] [55]                 errflg++;
[56]                 }

```

The output shows that the code for the **-v** option was never executed. To correct this, create a test that exercises the option.

Example 2: functions that are never called

Consider the following **lprof** summary.

```
Coverage Data Source: test.cnt
Date of Coverage Data Source: Wed Mar 6 11:11:58 1991
Object: myprog

percent   lines   total function
covered  covered  lines name

    91.5     97    106 compile
   100.0     18     18 step
   100.0     73     73 advance
   100.0      4      4 getrnge
    42.9     12     28 main
   100.0     29     29 execute
   100.0     19     19 succeed
    42.9      3      7 putdata
     0.0      0     19 regerr
   100.0     21     21 fgetl

    85.2    276    324 TOTAL
```

None of the lines in the function **regerr** are executed. To find out why, invoke **cscope** (described later in this chapter) and request a list of the functions that call it. If **cscope** reports that no function calls **regerr**, it may be possible to delete it from the code.

cscope

The **cscope** browser builds a cross-reference symbol table for the functions, function calls, macros, and variables in designated source files. It then allows you to query that table about the locations of symbols. **cscope** presents a menu and asks you to choose the type of search you would like to perform. When **cscope** has completed this search, it prints a list of the lines on which it has found the item requested. You can then indicate which of these lines you want to examine. After you have requested a subset of the lines, **cscope** allows you to edit a line or to begin another search.

Throughout a **cscope** session, you have the option of returning to the menu from the editor to request a new search. There are a variety of single-character commands available for manipulating the menu.

Because the procedure you follow will depend on the task you select, there is no single set of instructions for using **cscope**. To show how this browser works, a tutorial is provided later in the chapter.

Configuring the environment

Before you can use **cscope**, you must ensure that it can employ the editor and terminal that you wish to use.

Setting the terminal type

Check the value of the **TERM** environment variable to make sure you have set it to the correct terminal type for your terminal. This is done with commands such as the following:

```
TERM=ansi; export TERM
```

These commands may be used at the UNIX System prompt or in a *.profile* file.

When you invoke **cscope**, you may see the following error message:

```
cscope: "term" is not in the terminal database.
```

If this message appears, your terminal may not be listed in the terminal information (terminfo) database that is currently loaded. Try reloading the database from the Terminal Information Utilities.

You may also see the message:

```
cscope: TERM variable is not set or is not exported in your .profile
```

If this message appears, set and export the **TERM** variable.

Choosing an editor

By default, **cscope** invokes the **vi** editor. If you want to use another editor, such as the **emacs** editor, you must set the **EDITOR** environment variable. This is done with commands such as the following:

```
EDITOR=emacs; export EDITOR
```

These commands may be used at the UNIX System prompt or in a *.profile* file.

cscope expects **vi** and any other editor it uses to have a standard command-line syntax of the following form:

```
editor +linenum filename
```

However, if the editor you want to use does not conform to this command-line syntax, you must write an interface between **cscope** and the editor.

For example, suppose you want to use **ed**. Because **ed** does not allow specification of a line number on the command line, you will not be able to edit or view any files while using **cscope**. To solve this problem, write a shell script called, for example, **myedit**, that consists of the following line:

```
/bin/ed $2
```


Then set the value of **EDITOR** to the name of your shell script:

```
EDITOR=myedit; export EDITOR
```

Normally, when **cscope** invokes the editor, it uses a command line such as:

```
vi +17 file.c
```

Now, when **cscope** invokes the editor, it will call the shell script **myedit** with the same arguments; this script will discard the line number and call **ed** correctly with the filename (\$2).

NOTE The **ed** tool has one other drawback as a **cscope** editor that you should take into consideration when selecting an editor: it cannot move you to specified lines in the file. If you use the shell script shown in the previous example, you will have to move to specified lines manually.

Choosing a browser

If you want to use **cscope** only for browsing (without editing) you can set the **VIEWER** environment variable to **pg** and export **VIEWER**. **cscope** will then invoke **pg** instead of **vi**.

Using cscope

If all the source files for the program to be browsed (with the possible exception of standard system header files) are in the current directory, invoke **cscope** without any arguments:

```
$ cscope
```

By default, **cscope** builds its cross-reference table for all the C, **lex**, and **yacc** source files in the current directory. Therefore, typing **cscope** without any arguments is equivalent to the following command line:

```
$ cscope *.chly
```

cscope also searches the standard directories for any header files that you include with **#include**.

To browse through specific files, invoke **cscope** with file names as arguments on the command line:

```
$ cscope file1.c file2.c file3.h
```

A list of all the files to be browsed may be read from a file: use the **-i** option to specify this file. For example:

```
$ cscope -i list
```

If the source files are in a directory tree, the following commands will allow you to examine all the source files easily:

```
$ find . -name '*.chly' -print | sort > filelist
$ cscope -i filelist
```

Searching for **#include** files

cscope automatically searches for the **#include** files that it encounters when scanning. The **-I** option for **cscope** is similar to the **-I** option for **cc**. It directs **cscope** to search specified directories for **#include** files. For example:

```
$ cscope -I ./hdr
```

The **cscope** tool searches for **#include** files in directories in this order:

1. the current directory
2. directories specified by the **-I** option
3. the standard location for header files (usually */usr/include*).

Searching for source files

By default, **cscope** searches for source files only in the current directory. The environment variable **VPATH** can be used to extend your search for source files from a single directory to a set of directories.

VPATH should be set to the list the directories you want searched, in the order you want them searched. Separate the directory names with colons. You must specify the current directory in **VPATH** if you want it to be searched. The current directory can be represented by the dot "." symbol. The **VPATH** variable may be set with the commands such as the following:

```
$ VPATH=/fs2/mydirectory:/fs1/delivered:/fs1/proj/official:.
$ export VPATH
```

These commands may be used on the command line or in a *.profile* file. In this example, **cscope** will first search for files in the directory */fs2/mydirectory*. If the file is not in that directory, **cscope** continues searching for it in the other directories specified in **VPATH** until it finds the file. The current directory, represented by ".", is the last directory to be searched in this example.

Running cscope

After **cscope** has been invoked and the cross-reference information processed, the **cscope** task menu appears on the screen:

```
cscope          Press the ? key for help
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

Press the **<Tab>** or **<Return>** key to move the cursor down the screen (with wraparound at the bottom of the display), and **<Ctrl>p** to move the cursor up. Once the cursor is at the desired input field, enter the text to be searched, and press the **<Return>** key.

The following single-key commands are available at any time during a **cscope** session.

Table 9-1 Menu manipulation commands

<Tab>	move to next input field
<Return>	move to next input field
<Ctrl>m	move to next input field
<Ctrl>p	move to previous input field
.	search with the last text typed
<Ctrl>r	rebuild the cross-reference
!	start an interactive shell (type <Ctrl>d to return to cscope)
<Ctrl>l	redraw the screen
?	display list of commands
<Ctrl>d	exit cscope

NOTE To type control characters such as **<Ctrl>p** hold down the **<Ctrl>** key and press the letter shown.

The cross-reference file

When **cscope** is invoked it checks whether a file containing a cross-reference symbol table exists in the current directory. If it does not exist, **cscope** builds this table and refers to it during subsequent sessions. This table is created in

the current directory and is called *cscope.out*. The next time **cscope** is invoked, it checks *cscope.out* for changes. **cscope** modifies the table if the list of source files has been changed. If the table has been modified, **cscope** rebuilds only the modified portions. Because copying information is much faster than building it, subsequent calls to **cscope** should require less start-up time than the initial call.

A cross-reference file other than *cscope.out* may be specified using the **-f** option. This is useful for keeping separate symbol cross-reference files in the same directory. You may want to do this if two programs are in the same directory but do not share all the same files. For example:

```
$ cscope -f admin.ref admin.c common.c aux.c libs.c
$ cscope -f delta.ref delta.c common.c aux.c libs.c
```

In the preceding example, the source for two programs are in the same directory, but the programs involve different files. By specifying two reference files, the cross-reference information for the two programs can be kept separate.

As with *cscope.out*, if the file specified on the **-f** option does not exist, **cscope** will build the cross-reference and leave it in the file specified.

The **-d** option allows you to prevent updating of the cross-reference table and thereby save time. You should use this option only if you are sure that your source files have not been changed. Because it is usually more important to safeguard against generating erroneous data than to save time, avoid using the **-d** option unless absolutely necessary. If you specify **-d** with **cscope** under the erroneous impression that your source files have not been changed, **cscope** will give you outdated information.

A tutorial example: locating the source of the error message

In this example, the task is to locate the source of an error message “out of storage” that is printed when a program is run. Assume that you are working with unfamiliar code. You will invoke **cscope** and start your search for the problem by locating the section of code where the error message is generated. Move the cursor to the fifth menu item, that is:

```
List lines containing this text string
```

Now enter the text to be searched for:

```
out of storage
```

This process is shown in the next example.

```
cscope          Press the ? key for help
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string: out of storage
Change this text string:
List file names containing this text string:
```

Press the **<Return>** key. The **cscope** tool searches for the specified text and finds one line that contains it.

NOTE The same procedure is followed to perform any other task listed in the menu, except the sixth, changing a text string. For a description and examples of changing a text string, see “Changing a text string” (page 311).

cscope reports its findings as follows:

```
Text string: out of storage
```

```
File      Line
1 alloc.c 56 (void) fprintf(stderr,
               "\n%s: out of storage\n",
               argv[0] );
```

```
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

After **cscope** shows the results of a successful search in this way, you have several options. For example, you may want to edit one of the lines found. If **cscope** has found several lines and a list of them will not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after **cscope** has found the specified text.

Table 9-2 Commands for use after initial search

Command	Meaning
1-9	edit this line (the number you type corresponds to an item in the list of lines printed by cscope)
space	display the lines after the current line
+	display the lines after the current line
-	display the lines before the current line
(Ctrl)e	edit all lines
>	append the list of lines being displayed to a file

NOTE If the first character of the text you are searching for matches one of these commands, precede it with a “\” (backslash).

Now examine the code around the newly found line. Enter “1” (the number of the line in the list). The editor will be invoked with the file *alloc.c*; the cursor will be at the beginning of line 56 of the text file.

```

{
    return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
    if (p == NULL) {
        (void) fprintf(stderr,
                        "\n%s: out of storage\n",
                        argv[0]);
        exit(C);
    }
    return(p);
}
~
~
~
~
~
~
~
~
"alloc.c" 60 lines, 1022 characters

```

By examining the code, you learn that the error message is generated when the variable **p** is NULL. To determine how an argument passed to **alloctest** could have been NULL, you must first identify the functions that call **alloctest**.

Exit the editor by using normal write and quit conventions, and return to the menu of tasks. Now request a list of functions that call **alloctest**, as shown in the next example:

Text string: out of storage

```
File      Line
1 alloc.c 56 (void) fprintf(stderr,
               "\n%s: out of storage\n",
               argv[0] );
```

List references to this C symbol:

Edit this function or #define:

List functions called by this function:

List functions calling this function: **alloctest**

List lines containing this text string:

Change this text string:

List file names containing this text string:

cscope finds and lists three such functions:

Functions calling this function: **alloctest**

```
File      Function  Line
1 alloc.c mymalloc  26 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc  36 return(alloctest(calloc((unsigned) nelem,
               (unsigned)
               size)));
3 alloc.c myrealloc 46 return(alloctest(realloc(p, (unsigned)
               size)));
```

List references to this C symbol:

Edit this function or #define:

List functions called by this function:

List functions calling this function:

List lines containing this text string:

Change this text string:

List file names containing this text string:

Now you need to know which functions call **mymalloc**. **cscope** finds ten such functions. It lists seven of them on the screen and instructs you to press the space bar to see the rest of the list.

Functions calling this function: mymalloc

File	Function	Line
1 alloc.c	stralloc	17 return(strcpy(mymalloc(strlen(s) + 1), s));
2 dir.c	makesrcdirlist	70 srcdirs = (char **) mymalloc(nsourcedirs * sizeof(char *));
3 dir.c	makesrcdirlist	89 s = mymalloc(strlen(srcdirs[i]) + n);
4 dir.c	makefilelist	115 srcfiles = (char **) mymalloc(msrcfiles * sizeof(char *));
5 dir.c	makefilelist	116 srcnames = (char **) mymalloc(msrcfiles * sizeof(char *));
6 dir.c	addincdir	212 incdirs = (char **) mymalloc(sizeof(char *));
7 display.c	dispinit	76 displine = (int *) mymalloc(mdisprefs * sizeof(int));

* 3 more lines - press the space bar to display more *

List references to this C symbol:

Edit this function or #define:

List functions called by this function:

List functions calling this function:

List lines containing this text string:

Change this text string:

List file names containing this text string:

Because you know that the error message ("out of storage") is generated at the beginning of the program, you can guess that the problem may have occurred in the function **dispinit** (display initialization). To view **dispinit**, the seventh function on the list, type "7":

```
void
dispinit()
{
    /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
        mdisprefs = 9;
    }
    /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char    file[PATHLEN + 1];    /* file name */
    char    function[PATLEN + 1]; /* function name */
    char    linenum[NUMLEN + 1];  /* line number */
    int     screenline;           /* screen line number */
    int     width;                /* source line display
                                   width */

    register int    i, j;
    "display.c" 440 lines, 10198 characters
```

Examining this code, you will see that **mymalloc** failed because it was called with a negative number. You are now in a position to fix the problem. The program needs a mechanism such that if the value of the variable **mdisprefs** is negative, it will abort after printing a meaningful error message.

Stacking **cscope** and editor calls

cscope and editor calls can be stacked. This means that when **cscope** puts you in the editor to display one symbol reference and there is another symbol of interest, you can call **cscope** again from within the editor without exiting the current invocation of either **cscope** or the editor. You can then back up to a previous invocation by exiting the appropriate **cscope** and editor calls.

Conditional compilation directives

Conditional compilation directives such as **#if**, **#ifdef**, and **#ifndef** allow you to put more than one definition of a function in your program, provided that after the preprocessor interprets the directives, only one definition of the function is passed on to the compiler. These conditional compilation directives

are ignored by **cscope**. If there are multiple definitions of a function, **cscope** recognizes only the first definition that appears in the source text; this occurs even if the normal interpretation of the preprocessor directives would cause a different definition to be compiled.

A consequence of this is that the correct definition of a function may not be accessible via the “List functions called by this function” and “List functions calling this function” menu items. You may obtain a list of lines containing the name of the function using the “List lines containing this text string” menu item. References to the function may be obtained using the “List references to this C symbol” menu item.

Examples of using **cscope**

This section gives three examples of how **cscope** can be used to perform tasks: change a constant to a preprocessor symbol, add an argument to a function, and change the value of a variable.

Changing a text string

If you select the sixth item in the task menu, “Change this text string”, **cscope** accepts a search string, prompts you for new text and displays the lines containing the old text. You can select the lines you want changed with any of the following single-key commands:

Table 9-3 Commands for selecting lines to be changed

1-9	mark or unmark the line to be changed
*	mark or unmark all displayed lines to be changed
space	display next lines
+	display next lines
-	display previous lines
a	mark all lines to be changed
<Ctrl>d	change the marked lines and exit
<Esc>	exit without changing the marked lines

Suppose you want to change a constant, ‘100’, to a preprocessor symbol, **MAXSIZE**. Select the menu item “Change this text string” and enter **\100**.

NOTE The **1** must be preceded by a “\” (backslash) because it has a special meaning (item 1 on the menu) to **cscope**.

Now press (Return); **cscope** will prompt you for the new text string. Type **MAXSIZE**. This process is illustrated in the following example:

```
cscope                                     Press the ? key for help
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string: 100
List file names containing this text string:
To: MAXSIZE
```

The lines containing the particular text string are displayed; **cscope** waits for you to specify the lines in which you want the text to be changed.

```
Change "100" to "MAXSIZE"
```

```
File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */
```

```
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
Select lines to change (press the ? key for help):
```

Occurrences of 100 in lines 1, 2, and 3 of the list (from lines 4, 26, and 8 of the program) are to be changed to **MAXSIZE**. The occurrences of 100 in *read.c* and *err.c* (lines 4 and 5 of the list) have a different meaning; in these lines, 100 should not be changed. Enter **1, 2, and 3**.

The numbers you type are not printed on the screen. Instead, **cscope** prints a ">" (greater than) symbol after each number of the list that you type. For example, after you type **1**, a > symbol is printed after the number 1 in the list (and before the line "init.c 4 char s[100];"), as shown in the following example:

Change "100" to "MAXSIZE"

```

File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

```

List references to this C symbol:

Edit this function or #define:

List functions called by this function:

List functions calling this function:

List lines containing this text string:

Change this text string:

List file names containing this text string:

Select lines to change (press the ? key for help):

After selecting lines, type **(Ctrl)d** to change them. Then **cscope** displays the lines that have been changed:

Changed lines:

```

char s[MAXSIZE];
for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

```

Type any character to continue:

When you type a character in response to this prompt, **cscope** will pause and redraw the screen before allowing you to continue with the session.

The next step is to add the **#define** for the new symbol **MAXSIZE**. Escape to the shell by typing an exclamation mark "!". The shell prompt will appear at the bottom of the screen. Then enter the editor and add the **#define**. To resume the **cscope** session, quit the editor and type **(Ctrl)d** to exit the shell.

Adding an argument to a function

The **cscope** tool makes it easy to add an argument to a function. This involves two steps: editing the function itself and adding the new argument to each place where the function is called.

First, edit the function by using the second menu item, "Edit this function or #define". Next, find out where the function is called. By invoking the fourth menu item, "List functions calling this function", you can get a list of all functions that call it. With this list, you can either invoke the editor on each line found by entering the list number for each line individually, or invoke the editor on all lines automatically by typing (Ctrl)e. **cscope** is especially useful when making this kind of change because it guarantees that none of the functions you need to edit will be overlooked.

Changing the value of a variable

The value of **cscope** as a browser becomes apparent when you want to see how a proposed change will affect your code. Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item, "List references to this C symbol", to obtain a list of references that will be affected. Then use the editor to examine each one. This will help you predict the overall effects of your proposed change. Later, you can use **cscope** this way again to verify that your changes have been made.

Chapter 10

make

Most programming projects encompass a large number of individual files. Keeping track of file interdependencies often gets too complex to maintain on a piece of paper or by memory. **make**(CP) was designed to keep track of file-to-file relationships, the order of command executions, and general file maintenance. **make** is a command generator. It generates command sequences to be executed within a UNIX system shell. Within a **make** description file, more commonly known as a *makefile* or *Makefile*, a user defines the file interdependencies and command sequences to be executed.

If a program must be linked from object files and libraries, which are in turn created from assembly or high-level language source files, then invoking **make** performs this task automatically.

By using **make**, a programmer no longer has to be concerned with the following scenario: if file A depends on file B, and if file B was modified after file A, then file A must be compiled and linked before the program can run correctly.

A programmer can now let **make** remember:

- file-to-file interdependencies
- which files were modified recently
- which files require recompilation after source changes
- the exact sequence of operations required to generate a new version of the program

NOTE For a detailed explanation of the **make** command line usage, see **make**(CP).

Basic features

make's primary function is to execute the steps required to produce a new version of a specific (target) program. **make** operates using the following information sources:

- user-defined description file
- filesystem data and timestamp information
- set of suffix rules

The user-defined description file, conventionally called *makefile* or *Makefile*, holds the information on inter-file dependencies and command sequences. For example, a program under development consists of the following:

- three C language source files: *x.c*, *y.c*, *z.c* Assume that *x.c* and *y.c* share some declarations found in *defs.h*
- assembly language source in *assmb.s*, called by one of the C sources
- a set of library routines in */usr/fred/lib/abc.a*

The following code is a typical description file (*makefile* or *Makefile*), containing all the information required by **make** (line numbers are included for illustration purposes only).

```

1      menu : x.o y.o z.o assmb.o abc.a
2              cc x.o y.o z.o assmb.o /usr/fred/lib/abc.a -o menu
3      x.o  : x.c defs.h
4              cc -c x.c
5      y.o  : y.c defs.h
6              cc -c y.c
7      z.o  : z.c
8              cc -c z.c
9      assmb.o : assmb.s
10             as -o assmb.o assmb.s
```

The description file contains four entries. Each entry contains a line with a colon (the dependency line), and one or more command lines beginning with a tab. To the left of the colon, on the dependency line, are one or more targets; to the right of the colon are the files (components) on which the targets depend. The tab-indents lines illustrate how targets are made from their components.

For example, line 1 states that the program depends on files *x.o*, *y.o*, *z.o*, the *assmb.o* assembly object file and the library *abc.a*. Line 2 specifies the linker and compiler commands used in creating the program from its components. **make** executes this command if *menu* does not exist or if any of the component files was modified after the 'last-modified' (timestamp) date of *menu*.

make uses the remainder of the information contained in the description file to ensure that each component is up-to-date before executing the **cc**(CP) command (line 2). Line 3 indicates that *x.o* depends on *x.c* and *defs.h*. If either of these has been modified, then *x.c* is recompiled, as directed by the command line (line 4). *y.o* has similar dependencies, as described in line 5. Line 7 shows that *z.o* has only one dependency. After all the subordinate dependencies have been brought up-to-date, **make** executes the command (**cc**) shown in line 2.

If none of the source or object files have changed since the last time *menu* was created, and all of the files are current, the **make** command announces this fact and stops.

These entries can take advantage of **make**'s ability to generate files and substitute macros. See "Macro definitions" (page 320) For information about macros. For example, an entry 'save' might be included to copy a certain set of files, or an entry 'clean' might be used to throw away unneeded intermediate files. For example:

```
clean:
    rm *.o
```

If the following command is issued, all object files are deleted.

make clean

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

Maintaining a zero-length file to keep track of the time at which certain actions were performed is useful for updating remote archives and listings.

The above description file can be simplified even further by using the existing naming and compiling conventions, provided in the UNIX system environment. C language sources always have a *.c* suffix; a convention imposed by the **cc** compiler. Similarly, assembly language sources have a *.s* suffix. These conventions allow **make**, acting upon a set of 'suffix rules', to perform many tasks. For example:

```
menu : x.o y.o z.o assmb.o /usr/fred/lib/abc.a
    cc -o menu x.o y.o z.o assmb.o /usr/fred/lib/abc.a
```

In building *menu*, **make** first checks whether *x.o* is up-to-date. It checks any file in the current directory that, according to the standard suffix rules, could be used to make *x.o*. If it locates a file, *x.c*, and if that file has been changed since *x.o* was last made, **make** applies one of its own rules by invoking the C compiler on *x.c*. Similarly, this occurs with the assembler files, which have a *.s* suffix.

By knowing and using the default suffix rules, or by creating new ones, the complexity of the description files can be reduced. See “Suffixes and transformation rules” (page 324) for more information on suffixes.

Parallel make

If **make** is invoked with the **-P** option, it tries to build more than one target at a time, in parallel. (This is done by using the standard UNIX system process mechanism which enables multiple processes to run simultaneously.)

```
prog : x.o y.o z.o
      cc x.o y.o z.o -lm -o prog
x.o  : x.c defs.h
      cc -c x.c
y.o  : y.c defs.h
      cc -c y.c
z.o  : z.c
      cc -c z.c
```

For the *makefile* shown above, it would create processes to build *x.o*, *y.o* and *z.o* in parallel. After these processes were complete, it would build *prog*.

The number of targets **make** will try to build in parallel is determined by the value of the environment variable **PARALLEL**. If **-P** is invoked, but **PARALLEL** is not set, then **make** will try to build no more than two targets in parallel.

You can use the **.MUTEX** directive to serialize the updating of some specified targets. This is useful when two or more targets modify a common output file, such as when inserting modules into an archive or when creating an intermediate file with the same name, as is done by **lex(CP)** and **yacc(CP)**.

To prevent *make* from building *x.o* and *y.o* in parallel, the *makefile* above would contain a **.MUTEX** directive of the form:

```
.MUTEX: x.o y.o
```

Recursive makefiles

Another feature of **make** concerns the environment and recursive invocations. For testing purposes, if **make -n** is used all actions are printed, including output from lower-level invocations of **make**. This command prints out the commands that **make** issues without actually taking the time to execute them.

If the sequence (**\$MAKE**) appears anywhere in a shell-command line, the line is executed even if the **-n** flag is set. Because the **-n** flag is exported across invocations of **make** (through the **MAKEFLAGS** variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems.

makefiles and substitutions

A makefile contains the following:

- dependency information
- macro definitions
- executable commands

Dependency line syntax

A dependency line has the form:

```
target1 [target2 ...] [:] [dependent1 ...] [; commands] [# ...]  
[ \t commands] [# ...]  
...
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. Shell meta-characters such as `*` and `?` are expanded when the line is evaluated. Commands appear either after a semicolon on a dependency line or on lines beginning with tabs immediately following a dependency line. A command is any string of characters not including a number sign (`#`), except when the number sign is in quotes.

Comments

A number sign (`#`) denotes a comment line. All characters after the number sign, on the same line, are ignored. Blank lines are ignored.

Continuation lines

If a non-comment line is too long, it can be continued by using a backslash `"\"` at the end of the line. However, if the last character on a line is a backslash, then it and everything following it is replaced by a single blank.

Dependency information

A dependency line lists both targets and components, separated by either one or two colons. A target name can appear on more than one dependency line, but all of those lines must be of the same (single or double-colon) type. For example, if a single colon is used on a dependency line, then only a single set of commands can be associated with that target. If a target appears on more than one single-colon line, only one of those lines can have a command sequence associated with it. The occurrence of targets on multiple single-colon lines is used as a notational convenience for clarifying complex makefiles.

make concatenates all the components following each occurrence of the repeated target into a single dependency list. For example:

```
target1 : comp1 comp2
        commands
target1 : comp2 comp3
```

This is identical to:

```
target1 : comp1 comp2 comp3
        commands
```

A double colon allows the listing of the same target on two or more dependency lines, permitting commands to be associated with each target line. For example:

```
target1 target2 :: comp1 comp2
        commands
target2 :: comp3
        commands
```

Macro definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

The last definition assigns **LIBES** the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules.

make uses a simple macro mechanism for substitution in dependency lines and command strings. Macros can either be defined by command-line arguments or included in the makefile.

A macro is invoked by preceding the name with a dollar sign. Macro names longer than one character must be put in parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two macros are equivalent.

Internally generated macros: \$*, \$@, \$?, and \$<

\$*, \$@, \$?, and \$< are four special macros that change values during the execution of the command. Before any command is issued, certain internally maintained macros are set. The \$@ macro is set to the full name of the current target. The \$@ macro is evaluated only for explicitly named dependencies. The \$? macro is set to the string of names that were found to be younger than the target. The \$? macro is evaluated along with explicit rules from the *makefile*. If the command was generated by an implicit rule, the \$< macro is the name of the related file that caused the action, and the \$* macro is the prefix, shared by the current and the dependent filenames.

The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -lm
menu: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o $@
      . . .
```

make LIBES="-ll -lm"

This command loads the three objects with both the **lex** (**-ll**) and the **math** (**-lm**) libraries, because macro definitions on the command line override definitions in the makefile. (In UNIX system commands, arguments with embedded blanks must be quoted.) For example, the following command can be used:

```
cd $(<D); $(MAKE) $(<F)
```

Extensions of \$*, \$@, and \$<

The following related macros are also available: \$(@D), \$(@F), \$(*D), \$(*F), \$(<D), and \$(<F). The **D** refers to the directory part of the single-character macro. The **F** refers to the filename part of the single-character macro. These additions are useful when building hierarchical *makefiles*. They allow access to directory names for purposes of using the UNIX **cd**(C) command.

Also see **make**(CP) for descriptions of the \$% (library member) and \$! (target name) macros.

Search path macro

If a macro named **VPATH** is assigned a list of colon-separated directory names, **make** will search these directories when looking for files with relative path names, in the same way the shells use the **PATH** environment variable.

Executable commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (**-s** option) or if the command line in the description file begins with an **"@"** sign.

make normally stops if any command signals an error by returning a nonzero error code. Errors are ignored if the **-i** option has been specified on the **make** command line, if the special target name **.IGNORE** appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate.

NOTE If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the target name **.DEFAULT** are used. If **.DEFAULT** is not defined, **make** prints a message and stops.

Commands contained within a *makefile* invoke a UNIX system shell and are executed there; a shell is separately invoked for each successive command. For example, the following makefile results in three invocations of the shell:

```
Report : data1 data2 awk1
        sort - bdf data1 data2 > /usr/tmp/rptxxx
        awk - f awk1 /usr/tmp/rptxxx > report
        rm /usr/tmp/rptxxx
```

Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (**cd** and shell control commands, for instance) that have meaning only within a single shell process. These results are forgotten before the next line is executed. Consequently, a **cd** command remains in effect within a single line. In the following example, the **ls** command lists the directory where **make** was invoked, not *newdir*.

```
target1 : comp1 comp2
        cd newdir
        ls
```

To achieve the desired result, try the following example. In this example, the shell is invoked on the entire command line, and the **cd** remains in effect when the **ls** is executed.

```
target1 : comp1 comp2
        cd newdir; ls
```

In general, any shell command can be used in a makefile, but there are restrictions. Each command must be a single line, which constrains the use of the shell multi-line constructs. A way to circumvent this limitation is to use backslashes to prevent the shell interpretation of a newline. In the following example, the middle lines comprise a single shell command. The backslashes

suppress the normal interpretation of a newline, resulting in these lines being considered a single line.

```
test:
    echo "start"
    for i in 1 2 3; do \
        echo "start"; \
        echo "starting..."; \
        echo "...done"; \
    done
    echo "end"
```

Output translations

Macros in shell commands are translated when executed. The form is as follows:

```
$(macro:string1=string2)
```

For each appearance of **string1** in the evaluated macro, **string2** is substituted. Finding **string1** in **\$(macro)** means that the evaluated **\$(macro)** is considered a series of strings, each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **\$(macro)** means that a regular expression of the following form was found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. This type of translation is useful when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script that can handle all the C language programs (which are those files ending in *.c*). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB) (a.o) $(LIB) (b.o) $(LIB) (c.o)
$(CC) -c $(CFLAGS) ${?:.o=.c}
$(AR) $(ARFLAGS) $(LIB) $?
rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Suffixes and transformation rules

The **make** program uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **-r** option is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name **.SUFFIXES**. The **make** program searches for a file with any of the suffixes on the list. If one is found, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. Thus, the name of the rule to transform a **.r** file to a **.o** file is **.r.o**. If the rule is present and no explicit command sequence has been given in the user's makefiles, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro **\$*** is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro **\$<** is the full name of the dependent line that caused the action.

The order of the suffix list is significant because the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for **.SUFFIXES** in the *makefile*. The dependents lines are added to the usual list. A **.SUFFIXES** line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit rules

The **make** program uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list (in order) is as follows:

.o	object file
.c	C source file
.c~	SCCS C source file
.y	yacc C source grammar
.y~	SCCS yacc C source grammar
.l	lex C source grammar
.l~	SCCS lex C source grammar
.s	assembler source file
.s~	SCCS assembler source file
.sh	shell file
.sh~	SCCS shell file
.h	header file
.h~	SCCS header file
.C	C++ source file
.C~	SCCS C++ source file

.Y	yacc C++ source grammar
.Y~	SCCS yacc C++ source grammar
.L	lex C++ source grammar
.L~	SCCS lex C++ source grammar
.f	FORTRAN source file
.f~	SCCS FORTRAN source file

See Figure 10-1, "Summary of default transformation path" (this page) for a summary of the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the makefile.

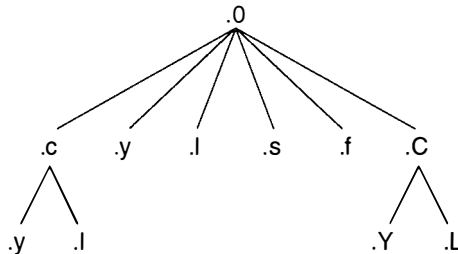


Figure 10-1 Summary of default transformation path

If the file *x.o* is needed and an *x.c* is found in the makefile directory, the *x.c* would be compiled. If there is an *x.l* and an *x.c* file and the *x.l* has been modified (that is, has a later modification date than the *x.c*), then **make** runs **lex** on the *x.l* file to re-create the *x.c* file. Otherwise, just the *x.c* file would be recompiled to produce the *x.o* file. However, if there is no *x.c* but there is an *x.l*, **make** uses the direct link as shown in Figure 10-1, "Summary of default transformation path" (this page)

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. **AS**, **CC**, **CCC**, **F77**, **YACC**, and **LEX** are the macro names of the compilers used by **make**. By specifically changing a compiler macro, a programmer can select which compiler is to be used. For example:

```
make CC=newcc
```

This causes the **newcc** command to be used instead of the usual C language compiler. The macros **ASFLAGS**, **CFLAGS**, **CCFLAGS**, **F77FLAGS**, **YFLAGS**, and **LFLAGS** can be set to cause these commands to be issued with optional flags. For example:

```
make "CFLAGS=-g"
```

This causes the **cc** command to include debugging information.

Archive libraries

The **make** program has an interface to archive libraries. A user can name a member of a library, as in these examples:

```
projlib(object.o)

projlib((entrypt))
```

The second method actually refers to an entry point of an object file within the library. (The **make** program looks through the library, locates the entry point, and translates it to the correct object-filename.)

To use this procedure to maintain an archive library, the following type of *makefile* is required for each object:

```
projlib:: projlib(pfile1.o)
$(CC) -c -O pfile1.c
$(AR) $(ARFLAGS) projlib pfile1.o
rm pfile1.o
projlib:: projlib(pfile2.o)
$(CC) -c -O pfile2.c
$(AR) $(ARFLAGS) projlib pfile2.o
rm pfile2.o
```

This is tedious and error-prone. In most cases, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time.

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the *.a* suffix. A *.c.a* rule is the rule for compiling a C language source file, adding it to the library, and removing the *.o* file. Similarly, the *.y.a*, the *.s.a*, and the *.l.a* rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are *.c.a*, *.c~.a*, *.f.a*, *.f~.a*, *.s.a*, *.s~.a*, *.C.a*, and *.C~.a*. (The tilde (~) syntax will be described shortly.)

The user may define other needed rules in the *makefile*.

The two-member library mentioned earlier is then maintained with the following shorter *makefile*:

```
projlib:      projlib(pfile1.o) projlib(pfile2.o)
              @echo projlib up-to-date
```

The internal rules are already defined to complete the preceding library maintenance. The actual *.c.a* rule is as follows:

```
.c.a:
$(CC) -c $(CFLAGS) $<
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.o
```

In this example, the **\$@** macro is the *.a* target (*projlib*); the **\$<** and **\$*** macros

are set to the out-of-date C language file, and the filename minus the suffix, respectively (*pfile1.c* and *pfile1*). The `$<` macro (in the preceding rule) could have been changed to `$*.c`.

This is what **make** does when it sees the construction:

```
projlib:    projlib(pfile1.o)
           @echo projlib up-to-date
```

Assume the object in the library is out-of-date with respect to *pfile1.c*. Also, there is no *pfile1.o* file.

1. Before using **make** on *projlib*, check each dependent of *projlib*.
2. Enter: **make projlib**.
3. *projlib(pfile1.o)* is a dependent of *projlib* and needs to be generated.
4. Before generating *projlib(pfile1.o)*, check each dependent of *projlib(pfile1.o)*.
5. Use internal rules to try to create *projlib(pfile1.o)*. (There is no explicit rule.) Note that *projlib(pfile1.o)* has parentheses in the name to identify the target suffix as *.a*. This is the key. There is no explicit *.a* at the end of the *projlib* library name. The parentheses imply the *.a* suffix. In this sense, the *.a* is hard-wired into **make**.
6. Break the name *projlib(pfile1.o)* up into *projlib* and *pfile1.o*. Define two macros, `$@` (`=projlib`) and `$*` (`=pfile1`).
7. Look for a rule *.X.a* and a file `$*.X*`. The first *.X* (in the `.SUFFIXES` list) that fulfills these conditions is *.c* so the rule is *.c.a*, and the file is *pfile1.c*. Set `$<` to *pfile1.c* and execute the rule. In fact, **make** must then compile *pfile1.c*.
8. The library has been updated. Execute the command associated with the *projlib*: dependency. That is:

```
@echo projlib up-to-date
```

It should be noted that to let *pfile1.o* have dependencies, the following syntax is required:

```
projlib(pfile1.o):    $(INCDIR)/stdio.h  pfile1.c
```

There is also a macro for referencing the archive member name when this form is used. The `$%` macro is evaluated each time `$@` is evaluated. If there is no current archive member, `$%` is null. If an archive member exists, then `$%` evaluates to the expression between the parentheses.

Tildes in SCCS filenames

Files under SCCS control have the prefix *s.* followed by the filename. However, the syntax of **make** does not directly permit referencing of prefixes. The SCCS files are the exception to this. To allow **make** to access the prefix *s.* the suffix rules use tildes (~) to identify SCCS files. For example, the default suffix rule describes how to transform a C language source file under SCCS control into an object file:

```
.c~.o:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c
```

The tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

NOTE *makefile* or *Makefiles* files under SCCS control are accessible to **make**. That is, if **make** is run and only a file named *s.makefile* or *s.Makefile* exists, **make** does a **get** on the file, then reads and removes the file.

The following SCCS suffixes and rules involving SCCS transformations are internally defined:

```
.c~:      transform SCCS C source file to executable file
.c~.a:    transform SCCS C source file to library (archive) file
.c~.c:    transform SCCS C source file to C source file
.c~.o:    transform SCCS C source file to object file
.y~.c:    transform SCCS yacc source file to C source file
.y~.o:    transform SCCS yacc source file to object file
.y~.y:    transform SCCS yacc source file to yacc source file
.l~.c:    transform SCCS lex source file to C source file
.l~.l:    transform SCCS lex source file to lex source file
.l~.o:    transform SCCS lex source file to object file
.s~:      transform SCCS assembler source file to executable file
.s~.a:    transform SCCS assembler source file to library (archive) file
.s~.s:    transform SCCS assembler source file to assembler source file
.s~.o:    transform SCCS assembler source file to object file
.sh~:     transform SCCS shell file to executable file
.sh~.sh:  transform SCCS shell file to shell file
.h~.h:    transform SCCS C header file to C header file
.f~:      transform SCCS Fortran source file to executable file
.f~.a:    transform SCCS Fortran source file to library (archive) file
.f~.f:    transform SCCS Fortran source file to Fortran source file
.f~.o:    transform SCCS Fortran source file to object file
.C~:     transform SCCS C++ source file to executable file
```

.C~.C: transform SCCS C++ source file to C++ source file
.C~.a: transform SCCS C++ source file to library (archive) file
.C~.o: transform SCCS C++ source file to object file
.Y~.C: transform SCCS **yacc**/C++ source file to C++ source file
.Y~.o: transform SCCS **yacc**/C++ source file to object file
.Y~.Y: transform SCCS **yacc**/C++ source file to **yacc**/C++ source file
.L~.C: transform SCCS **lex**/C++ source file to C++ source file
.L~.o: transform SCCS **lex**/C++ source file to object file
.L~.L: transform SCCS **lex**/C++ source file to **lex**/C++ source file

Obviously, the user can define other rules and suffixes, which may prove useful. The tilde provides a handle on the SCCS filename format so that this is possible.

The null suffix

Many programs consist of a single source file. **make** handles this case with a null suffix rule. The following example specifies how to create a file with a null suffix (visualize the null suffix occurring between the given suffix and the colon), from a file with a `.c` suffix.

```
.c:
    $(CC) $(CFLAGS) $< -o $@
```

In fact, this `.c:` rule is internally defined, so no *makefile* is necessary at all. The user only needs to type:

```
make cat dd echo date
```

All of these are UNIX system single-file programs, and all four C language source files are passed through the above shell command line associated with the `.c:` rule. The internally defined single-suffix rules are:

```
.c:
.c~:
.s:
.s~:
.f:
.f~:
.sh:
.sh~:
.C:
.C~:
```

Others may be added to the *makefile* by the user.

Creating new suffix rules

Both significant suffix rules and the rules used in transforming files of one suffix-type into another can be modified. Including the following line in the makefile adds *.q*, *.w*, *.t* to the list of available suffixes:

```
.SUFFIXES : .q .w .t
```

To delete all currently recognized suffixes, add this line:

```
.SUFFIXES :
```

Therefore, the combination of the two lines replaces the current suffixes with *.q*, *.w*, and *.t*:

```
.SUFFIXES :  
.SUFFIXES : .q .w .t
```

NOTE Be sure to include spaces or tabs between each suffix on the **.SUFFIXES** line.

To define new suffix rules, simply add the rules to the makefile. This appears as a normal entry. For example, **make** is required to help maintain a set of text files requiring formatting and printing. Add the suffixes and define suffix rules as follows:

```
.SUFFIXES :  
.SUFFIXES : .s .t .l  
TBL = tbl  
EQN = eqn  
NEQN= neqn  
ROFFARGS =  
  
.s.l : #format troff -ms source file for line printer  
      $(TBL) $<|$(NEQN)| troff -ms -e -Tlp -u5 $(ROFFARGS)>$.l  
.s.t : #format troff -ms source file for terminal  
      $(TBL) $<|$(NEQN)| troff -ms -e -Tlp -u5 $(ROFFARGS)| ul -i>$.t  
.s.a : #format troff -ms source file for apple laser printer  
      $(TBL) $<|$(EQN)| troff -ms -e -Tspc $(ROFFARGS)>$.a
```

The first line of the makefile nullifies the default suffix rule of **make**. The second line establishes three suffixes as significant. These suffixes are interpreted as *.s* suffixes, indicating their use of the **-ms** formatting macros. The first entry declares how to make a *.l* file out of an *.s* file. That provides the rule for converting a source file containing **-ms** macros into a formatted output file for a line printer. Similarly, the third entry describes how to create a *.a* file (for output to a laser printer). These rules work in the same manner as regular entries: the second suffix belongs to the ‘target’ and the first suffix belongs to the ‘component’.

include files

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the rest of the line is assumed to be a filename, which the current invocation of **make** reads. Macros may be used in filenames. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **include** files are supported.

Dynamic dependency parameters

A dependency line consists of a target to the left of the colon, and a dependent (components) to the right of the colon. A target may be a large number of executable files, having one source file each. Each of the executable files must be compiled separately using its own source file. This creates a long and tedious makefile. **make** can create a list of executables under one name. For example:

```
CMD5 = cat dd echo date cc cmp comm ar ld chown
```

When a **\$\$@** macro is used to the right of the colon in a makefile, replace the dependent (components) with the target name (which contains the list of executables requiring compilation). This is the dynamic dependency parameter. It is dynamic because each of the components contained in **CMD5** is used. For example, the following compiles all the components previously defined in the target (**CMD5**) with a two-line entry in the makefile:

```
$(CMD5) : $$@.c
$(CC) $? -o $@
```

Obviously, this is a subset of all the single-file programs. For multiple-file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **\$\$(@F)**. It represents the filename part of **\$\$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the */usr/include* directory from a *makefile* in the */usr/src/head* directory. Therefore, the */usr/src/head/makefile* looks like this:

```

INCDIR = /usr/include

INCLUDES = \
    $(INCDIR)/stdio.h \
    $(INCDIR)/pwd.h \
    $(INCDIR)/dir.h \
    $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
    cp $? $@
    chmod 0444 $@

```

This completely maintains the */usr/include* directory whenever one of the above files in */usr/src/head* is updated.

Environment variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment.

make automatically defines a macro, **MAKEFLAGS**, containing the command-line option flags given in the invocation of **make**. The intent of the **MAKEFLAGS** macro, upon startup, is to set environment variables to be used by successive invocations of **make**.

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable. If it is not present or null, the internal **make** variable **MAKEFLAGS** is set to the null string. Otherwise, each letter in **MAKEFLAGS** is assumed to be an input flag argument and is processed as such. (The only exceptions are the **-f**, **-p**, and **-r** flags.)
2. Read the internal list of macro definitions.
3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).
4. Read the *makefile*(s). The assignments in the *makefile*(s) override the environment. This order is chosen so that it is possible to know what to expect by reading the *makefile* being executed. However, if the **-e** command-line flag is used, the environment overrides the *makefile* assignments. Therefore, if **make -e ...** is typed, the variables in the environment override the definitions in the *makefile*. Also, **MAKEFLAGS** overrides the environment if assigned. This is useful for further invocations of **make** from the current *makefile*.

The priority of assignments is from least binding to most binding; the precedence of assignments is:

- internal definitions
- environment
- makefile(s)
- command line

NOTE If the **-e** option is specified, the priority of assignments becomes:

- internal definitions
- makefile(s)
- environment
- command line

This order is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

Suggestions and warnings

The most common difficulties can arise from your understanding of **make**'s specific meaning of dependency. If file *x.c* has a:

```
#include "defs.h"
```

line, then the object file *x.o* depends on *defs.h*; the source file *x.c* does not. If *defs.h* is changed, nothing is done to the file *x.c* while file *x.o* must be recreated.

To discover what **make** would do, the **-n** option is very useful. The command

```
$ make -n
```

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them.

If a change to a file is minor (such as adding a comment to an *include* file), the **-t** (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file.

make -ts

This command (**-ts** stands for touch silently) causes the relevant files to appear up-to-date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Makefile example

For example, consider a makefile to maintain the **make** command itself. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. An example of the makefile follows:

```
# makefile for the make command

FILES = Makefile defs.h main.c doname.c misc.c files.c dosys.c gram.y
OBJECTS = main.o doname.o misc.o files.o dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make: $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
    @size make

$(OBJECTS): defs.h

clean:
    -rm *.o gram.c

install:
    @size make /usr/bin/make
    cp make /usr/bin/make && rm make

lint : dosys.c doname.c files.c main.c misc.c gram.c
    $(LINT) dosys.c doname.c files.c main.c misc.c \
    gram.c
```

The **make** program prints out each command before issuing it.

The following output results from typing **make** in a directory containing only the source and makefiles:

```
cc -O -c main.c
cc -O -c doname.c
cc -O -c misc.c
cc -O -c files.c
cc -O -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -O -c gram.c
cc main.o doname.o misc.o files.o dosys.o gram.o -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an "at" (@) sign in the makefile.

Internal rules

The standard set of internal rules used by **make** is reproduced in the examples shown on the following pages.

Example 10-1 make internal rules (Page 1 of 5)

```
#
#      SUFFIXES RECOGNIZED BY MAKE
#
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~
#
#      PREDEFINED MACROS
#
AR=ar
ARFLAGS=rv
AS=as
ASFLAGS=
BUILD=build
CC=cc
CFLAGS=-O
C++C=CC
C++FLAGS=-O
F77=f77
FFLAGS=-O
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
MAKE=make
YACC=yacc
YFLAGS=
#
#      SPECIAL RULES
#
markfile.o : markfile
    A=@; echo "static char _sccsid[]=\042'grep $$A'(#)' markfile'\042;" \
    > markfile.c
    $(CC) -c markfile.c
    -rm -f markfile.c
```

Example 10-1 make internal rules (Page 2 of 5)

```
#
#      SINGLE SUFFIX RULES
#

.c:
    $(CC) $(CFLAGS) $< $(LDFLAGS) -o $@

.c~:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) $*.c $(LDFLAGS) -o $*
    -rm -f $*.c

.f:
    $(F77) $(F77FLAGS) $< $(LDFLAGS) -o $@

.f~:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) $< $(LDFLAGS) -o $*
    -rm -f $*.f

.sh:
    cp $< $@; chmod 0777 $@

.sh~:
    $(GET) $(GFLAGS) $<
    cp $*.sh $*; chmod 0777 $@
    -rm -f $*.sh
```

Example 10-1 make internal rules (Page 3 of 5)

```

#
#      DOUBLE SUFFIX RULES
#

.c~.c .f~.f .s~.s .sh~.sh .y~.y .l~.l .h~.h:
    $(GET) $(GFLAGS) $<

.c.a:
    $(CC) -c $(CFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.o

.c~.a:
    $(GET) $(GFLAGS) $<
    $(CC) -c $(CFLAGS) $*.c
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.[co]

.c.o:
    $(CC) $(CFLAGS) -c $<

.c~.o:
    $(GET) $(GFLAGS) $<
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c

.f.a:
    $(F77) $(F77FLAGS) -c $*.f $(LDLFLAGS)
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.o

.f~.a:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) c $*.f $(LDLFLAGS)
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[fo]

```

Example 10-1 make internal rules (Page 4 of 5)

```

.f.o:
    $(F77) $(F77FLAGS) -c $*.f $(LDFLAGS)

.f~.o:
    $(GET) $(GFLAGS) $<
    $(F77) $(F77FLAGS) -c $*.f $(LDFLAGS)
    -rm -f $*.f

.s~.a:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    $(AR) $(ARFLAGS) $@ $*.o
    -rm -f $*.[so]

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
    $(GET) $(GFLAGS) $<
    $(AS) $(ASFLAGS) -o $*.o $*.s
    -rm -f $*.s

.l.c :
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@

.l~.c:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    mv lex.yy.c $@

```

Example 10-1 make internal rules (Page 5 of 5)

```

.l.o:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    rm lex.yy.c
    mv lex.yy.o $@
    -rm -f $*.l

.l~.o:
    $(GET) $(GFLAGS) $<
    $(LEX) $(LFLAGS) $*.l
    $(CC) $(CFLAGS) -c lex.yy.c
    rm -f lex.yy.c $*.l
    mv lex.yy.o $*.o

.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@

.y~.c :
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.y
    mv y.tab.c $*.c
    -rm -f $*.y

.y.o:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@

.y~.o:
    $(GET) $(GFLAGS) $<
    $(YACC) $(YFLAGS) $*.y
    $(CC) $(CFLAGS) -c y.tab.c
    rm -f y.tab.c $*.y
    mv y.tab.o $*.o

```

make

Chapter 11

Source code control system (SCCS)

Source Code Control System (SCCS) is a collection of UNIX system commands that maintain and track changes made to source code or documentation. SCCS is basically a file custodian. Under SCCS, whenever changes are made to a file, SCCS records those changes and maintains the original. SCCS can:

- store text files
- retrieve specific versions of files
- control updating access to files
- identify the version of a retrieved file
- record when, why and by whom changes are made to a file

These features are important if code and documentation undergo frequent changes due to maintenance or enhancement work. Whenever changes are applied to a file under SCCS control, only the latest set of changes are added to the SCCS image of that file. Using this approach, SCCS maintains successive versions, consuming minimal amounts of increased disk space. This allows for efficient use of resources and easy regeneration of previous file versions.

This chapter covers the following topics:

SCCS for beginners	How to create, retrieve, and update SCCS files
Delta numbering	How versions are numbered and named under SCCS
SCCS command conventions	Conventions and rules applied to SCCS commands
SCCS commands	Explanation of SCCS commands, with argument usage
SCCS files	How SCCS files are formatted, protected, and audited

NOTE Installation and implementation of SCCS are not covered here.

SCCS for beginners

This section presents several terminal session fragments. The best way to learn SCCS is to use it.

Terminology

Files under SCCS are composed of one or more sets of changes applied to the original version of the file. A set of changes depends on all the previous sets. For SCCS to keep track of the changes, any alterations to a file are stored separately in another file called a *delta*.

Each *delta* is assigned a name known as an SCCS IDentification string (SID). An SID contains four components. The first two are the release and level numbers, separated by a period. The next two are the branch and sequence numbers, also separated by a period. This is explained in “Delta numbering” (page 345).

The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file (that is, its first delta) is release 1 and version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. There will be more on delta numbering later. At this point, it is enough to know that, by default, SCCS assigns SIDs automatically.

Creating an SCCS file using admin

Using a text editor, create a file called *lang*, which contains a list of following programming languages:

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of the *lang* file can be given to SCCS by using the **admin**(CP) (admin-ister) command. The following command creates an SCCS file from the *lang* file:

```
admin -ilang s.lang
```

All SCCS file names must begin with *s.*, therefore *s.lang*. The **-i** option, together with its value argument (*lang*), indicates that **admin** is to create a new SCCS file with the contents of the file *lang*.

The output of the **admin** command is:

```
No id keywords (cm7)
```

This is a warning message that can also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get(CP)** command (page 350). In the following examples, this warning message is not shown, although it may be issued.

Remove the *lang* file. It is no longer needed because it now exists under SCCS as *s.lang*. Enter:

```
rm lang
```

Retrieving a file by means of get

Using the **get** command, retrieve the file *s.lang*:

```
get s.lang
```

The output of the **get** command is:

```
1.1
5 lines
```

This indicates that **get** has retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved file has been placed in a new file known as a 'g-file'. SCCS forms the name of the g-file by deleting the prefix *s.* from the name of the SCCS file. Thus, the original *lang* file has been recreated.

An **ls(C)** command lists both *lang* and *s.lang* files in the directory. SCCS retains *s.lang* for other uses.

The **get s.lang** command creates *lang* as read-only, keeping no information regarding its creation. When a file is retrieved in the above manner, it cannot be edited. If changes are to be made to the file, use the **get -e** command as follows:

```
get -e s.lang
```

The **get -e** command causes SCCS to create *lang* for editing. It also places pertinent information about *lang* in another new file, called the 'p-file' (*p.lang*, in this case), which is needed later by the **delta(CP)** command.

The **get -e** command prints additional status information including a message informing the user that a first delta has been created for this file. The SID is included in the **get -e** output:

```
1.1
new delta 1.2
5 lines
```

Recording changes by using delta

Modify the *lang* file by adding two more programming languages:

```
      SNOBOL
      Ada
```

To record the changes made to *lang*, issue the following command:

```
delta s.lang
```

The following prompt appears:

```
comments?
```

Respond with a meaningful description of the changes just applied to the file. For example:

```
comments? added more languages
```

The **delta** command now reads the p-file (*p.lang*) and determines what changes have occurred. It does this by performing its own **get** to retrieve the original version, and applying the **diff(C)** command. The **delta** command compares the original and the edited versions of the *lang* file: it determines the differences and stores those changes in *s.lang*. The *p.lang* and *lang* files, which are no longer needed, are automatically removed.

When this process is complete, **delta** outputs the following:

```
1.2          (1.2 is the SID of the delta just created)
2 inserted   (indicates how many new lines are in the file)
0 deleted    (indicates how many lines were deleted from the file)
5 unchanged  (indicates unchanged lines in the file)
```

Additional information about get

As shown in the previous example, **get** retrieves the latest version of the file *s.lang*. This is accomplished by starting with the original version and successively applying the changes (deltas) in order until all have been applied. If the **get** command is issued now, it retrieves version 1.2 of the file *s.lang*. Any of the following invocations of **get** produces the same result:

```
get s.lang
get -r1 s.lang
get -r1.2 s.lang
```

The numbers following the **-r** are SIDs. When the level number of the SID (**get -r s.lang**) is omitted, the default is the highest level number existing within that specific release. The second command (**get -r1 s.lang**) requests the retrieval of the latest version of release 1. The third command specifically requests the retrieval of a particular version, release 1 level 2.

Whenever a significant change is made to a file, the usual identification method is to increase the release number (the first number of the SID). Because normal automatic numbering of deltas proceeds by incrementing the level number, the user must explicitly instruct SCCS to increment the release number. This is accomplished as follows:

```
get -e -r2 s.lang
```

Release 2 does not exist, so **get** retrieves the latest version before release 2. The **get** command also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output that follows indicates that version 1.2 has been retrieved, and 2.1 is the new delta version that will be created.

```
1.2
new delta 2.1
7 lines
```

If the file is now edited (for example, by deleting COBOL from the list of languages) and **delta** is executed:

```
delta s.lang
comments? deleted COBOL from the list of languages
```

Then the **delta**'s output is:

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, and so on), or another new release can be created in a similar manner.

Delta numbering

Deltas may be thought of as the nodes of a tree in which the root node is the initial version of the file. The root node (original file) is normally named 1.1 and successor deltas (nodes) are named 1.2, 1.3, 1.4, and so on. The two numbers in these names are the first two components of an SID, known as the 'release' and 'level' numbers, respectively. SCCS automatically names new deltas by incrementing the level numbers whenever a delta is made.

In addition, if a release number is incremented, indicating a major change, the new release number is applied to all subsequent deltas. (An example of this evolutionary process is represented in Figure 11-1, "Evolution of an SCCS file" (this page).)

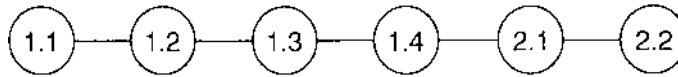


Figure 11-1 Evolution of an SCCS file

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned for a given delta that are not dependent on all previous deltas. For example, consider a program in production, at version 1.3, and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 11-1, "Evolution of an SCCS file" (this page). Assume that a problem is reported in version 1.3 that cannot wait for correction until release 2. The changes necessary to correct the problem are applied as a delta to version 1.3. This creates a new version that will then be released to the user, but does not affect the changes being applied for release 2 (that is, deltas 1.4, 2.1, 2.2, and so on). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, and so forth. This is shown in Figure 11-2, "Tree structure with branch deltas" (page 347):

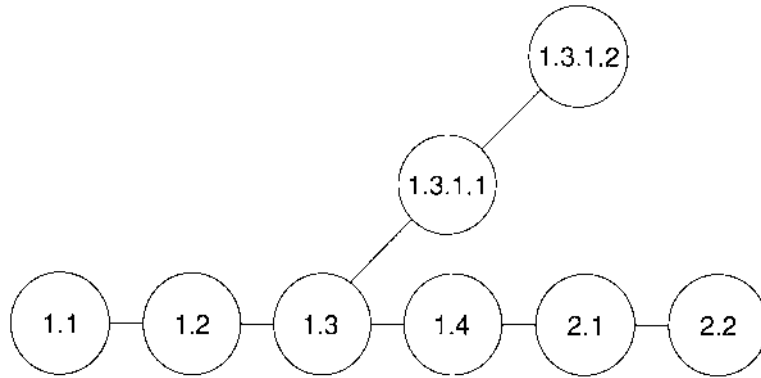


Figure 11-2 Tree structure with branch deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 11-3, “Extended branching concept” (this page). As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, and so on), but the secondary branch number remains the same.

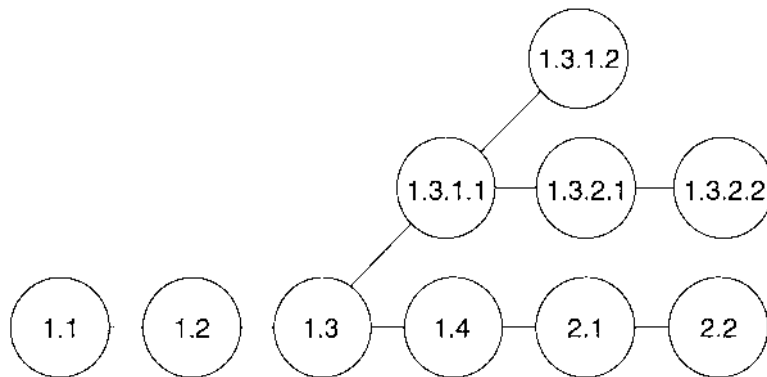


Figure 11-3 Extended branching concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown here. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible.

SCCS command conventions

SCCS commands accept two types of arguments:

- key letters
- filenames

A key letter represents an option that begins with a minus (-) sign, followed by a lowercase letter and, in some instances, a value.

File arguments, which may be file or directory names, specify the file(s) that the SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files contained in the directory, in which case non-SCCS files and files which are unreadable are silently ignored.

In general, a filename argument may not begin with a minus sign. If a filename - (a single minus sign) is specified, the command reads the standard input (usually your terminal) for lines and takes each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands **find(C)** and **ls(C)**.

Key letters are processed before filenames. Therefore, the placement of key letters is arbitrary; that is, they may be interspersed with file names. File names, however, are processed left to right. Somewhat different conventions apply to **what(CP)**, **sccsdiff(CP)**, and **val(CP)**, detailed later under SCCS commands (page 349) and documented in their respective manual pages.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags are discussed, but for a complete description see **admin(CP)**.

There is distinction between real user and effective user when discussing various actions of SCCS commands. For now, assume that the person logged into the UNIX system is both the real and effective user. See **passwd(C)** for more about real and effective users.

x-files and z-files

All SCCS commands that modify an SCCS file do so by writing a copy called the 'x-file'. This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x-file by replacing the *s.* prefix of the SCCS filename with an *x.* prefix. The x-file is created in the same directory as the SCCS file, given the same mode (see **chmod(C)**) and owned by the effective user. When processing is complete, the old SCCS file is destroyed, and the modified x-file is renamed (with *x.* replaced by *s.*) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the 'z-file'. SCCS forms its name by replacing the *s.* prefix of the SCCS filename with a *z.* prefix. The z-file contains the process number of the command that created it, and its existence prevents other commands from processing the SCCS file. The z-file is created with access permission mode 444 (read only) in the same directory as the SCCS file, and is owned by the effective user. Like the x-file, the z-file exists only for the duration of the execution of the command that creates it.

In general, users can ignore x-files and z-files. They are useful only in the event of system crashes or similar situations.

SCCS commands

This section describes the major features of the SCCS commands along with their most common arguments. Full descriptions with details of all arguments are presented in their respective manual pages.

Here is a quick-reference overview of the commands:

admin	initializes SCCS files, manipulates their descriptive text, and controls delta creation rights
cdc	changes the commentary associated with a delta
comb	combines consecutive deltas into one to reduce the size of an SCCS file
delta	applies deltas (changes) to SCCS files and creates new versions
get	retrieves versions of SCCS files
help	provides an explanation of error messages
prs	prints portions of an SCCS file in user-specified format
rmDEL	removes a delta from an SCCS file; allows removal of deltas created by mistake
sact	prints information about files that are currently out for editing
sccsdiff	shows differences between any two versions of an SCCS file
unget	undoes the effect of a get -e before the file is changed
val	validates an SCCS file
vc	a filter that may be used for version control
what	searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; useful in identifying information inserted by the get command

Error messages

SCCS commands produce error messages on the diagnostic output in this format:

```
ERROR [name-of-file-being-processed]: message text (code)
```

The code in parentheses can be used as an argument to the **help**(CP) command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

help command

The **help**(CP) command provides an explanation of the error messages. For example:

```
get lang
```

This error message is displayed:

```
ERROR [lang]: not an SCCS file (col)
```

For more information on this error message, use the **help** command in conjunction with the error code (col), for example:

```
help col
```

This provides the following explanation of why **get lang** produced an error message:

```
col:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

The **help** command, along with an error code, will provide assistance in understanding the meaning of most SCCS messages.

get command

The **get**(CP) command creates a file containing a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the 'g-file'. It is created in the current directory and owned by the real user. The mode assigned to the g-file depends on how the **get** command is used.

The most common use of **get** is:

```
get s.abc
```

This retrieves the latest version of file *abc* from the SCCS file tree trunk and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

This means version 1.3 of file *s.abc* has been retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords have been substituted in the file.

The generated g-file (file *abc*) is given access permission mode 444 (read only). This particular way of using **get** produces g-files only for inspection, compilation, and so on. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example, enter:

```
get s.abc s.xyz
```

It produces:

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID keywords in SCCS

In generating a g-file for compilation, it is useful to record the date and time of creation, the version retrieved, and the module's name within the g-file. This information can eventually appear in a load module when one is created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) keywords appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID keywords. The format of an ID keyword is an uppercase letter enclosed by percent signs (%).

%I%

In this example, **I** is the ID keyword replaced by the SID of the retrieved version of a file. Similarly, **%H%** is the current date in MM/DD/YY format and **%M%** is the name of the g-file. When **get** is executed on an SCCS file containing the following PL/1 declaration:

```
DCL ID CHAR(100) VAR INIT('%M% %I% %H%');
```

The PL/1 declaration produces the following:

```
DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/18/85');
```

For a complete list of the approximately 20 ID keywords provided, see **get**(CP).

When no ID keywords are substituted by **get**, the following message is issued:

```
No id keywords (cm7)
```

This message is normally treated as a warning by **get**, although the presence of the SCCS file **i** flag causes it to be treated as an error. For a complete list of the SCCS file flags, see **admin**(CP).

Retrieval of different versions

The version of an SCCS file that **get** retrieves is the most recently created delta of the highest-numbered trunk release. However, any other version can be retrieved with **get -r** by specifying the version's SID. The following command retrieves version 1.3 of file *s.abc* and produces the sample message on the standard output:

```
get -r1.3 s.abc
1.3
64 lines
```

A branch delta can be retrieved similarly, as demonstrated with the following command and sample output:

```
get -r1.5.2.3 s.abc
1.5.2.3
234 lines
```

When an SID is specified and the particular version does not exist in the SCCS file, an error message results.

The level number can be omitted, causing the retrieval of the trunk delta with the highest level number within the given release. The following command and sample output illustrate:

```
get -r3 s.abc
3.7
213 lines
```

If the specified release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. In the following example, release 9 is specified for retrieval. However, since the specified release does not exist, trunk delta 7.6 is retrieved as the latest version of file *s.abc* below release 9.

```
get -r9 s.abc
7.6
420 lines
```

For details on numbering by **delta**'s see "Delta numbering" (page 345).

Similarly, omitting the sequence number, for example, results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.)

```
get -r4.3.2 s.abc
4.3.2.8
89 lines
```

The **get -t** command retrieves the latest (top) version of a particular release when no **-r** is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. The following command (and sample output) illustrate this for a case where the most recent delta in release 3 is 3.5:

```
get -r3 -t s.abc
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce:

```
3.2.1.5
46 lines
```

Updating source

The **get -e** command indicates an intent to make a delta. First, **get** checks the following conditions:

- It checks whether the login name or group ID of the person executing **get** is present in the user list. The login name or group ID must be present for the user to be allowed to make deltas. (See “admin command” (page 363) for a discussion of making user lists.)
- It checks whether the release number (R) of the version being retrieved satisfies the relation:

```
floor is less than or equal to R,
which is less than or equal to ceiling.
```

This check determines whether the release being accessed is protected. The floor and ceiling are SCCS file flags, representing start and end of range.

- It checks whether the release number of the version being edited is locked against editing. The lock is an SCCS file flag.
- It checks whether multiple concurrent edits are allowed for the SCCS file by the SCCS file **j** flag.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get -e** causes the creation of a g-file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g-file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g-file while it is being edited for the purpose of making a delta. Any ID keywords appearing in the g-file are not substituted by **get -e** because the generated g-file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. As a direct result of this, **get** does not check for presence of ID keywords in the g-file. The following message is never output when **get -e** is used.

```
No id keywords (cm7)
```

In addition, **get -e** causes the creation (or updating) of a p-file that is used to pass information to the **delta** command.

For example:

```
get -e s.abc
```

The output of this command is:

```
1.3  
new delta 1.4  
67 lines
```

Undoing a **get -e**

There may be times when a file is erroneously retrieved for editing, such as when there is really no editing that needs to be done at the time. In such cases, the **unget**(CP) command cancels the delta reservation that was set up. For example, enter either of the following commands:

```
unget -r1.4 s.abc  
unget s.abc
```

These commands produce:

```
1.4
```

Additional get options

If **get -r** and/or **-t** keys are used together with the **-e** option, the version retrieved for editing is specified with **-r** and/or **-t**.

The **get -i** and **-x** commands specify a list of deltas to be included and excluded, respectively. (See **get(CP)** for the syntax of such a list.) Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. For example, two deltas can interfere when each one changes the same line of the retrieved g-file. A warning shows the range of lines within the retrieved g-file where the problem may exist.

NOTE The user should examine the g-file to determine what the problem is and take appropriate corrective steps (such as edit the file).

The **get -i** and **get -x** commands should be used with care.

The **get -k** command has two uses. The first is to regenerate a g-file that may have been accidentally removed or corrupted after **get -e**. The second use is to generate a g-file in which the replacement of ID keywords has been suppressed. A g-file generated by **get -k** is identical to one produced by the **get -e** command. However, no processing related to the p-file takes place.

Concurrent edits of different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get -e** commands may be executed on the same file unless two executions retrieve the same version or multiple concurrent edits are allowed.

The p-file created by **get -e** is named by automatic replacement of the SCCS filename's prefix *s.* with *p.*. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p-file contains the following information for each delta that is still in progress:

- SID of the retrieved version
- SID given to the new delta when it is created
- login name of the real user executing **get**

The first execution of **get -e** causes the creation of a p-file for the corresponding SCCS file. Subsequent executions simply update the p-file with a line containing the above information. Before updating, however, **get** checks to assure that the SID of the version to be retrieved has not already been retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress, and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of **get** must be carried out from different directories. Subsequent executions from the same directory attempt to overwrite the g-file, which is an SCCS error condition. In practice, this problem does not arise because each user normally has a different working directory.

For a discussion of how different users are permitted to use SCCS commands on the same files, see "Protection" (page 371).

Table 11-1, "Determination of new SID" (this page) shows the possible SID components a user can specify with **get** (left-most column), the version that is then retrieved by **get**, and the resulting SID for the delta, which **delta** creates (right-most column).

Table 11-1 Determination of new SID

SID specified in get*	-b key-letter used†	Other conditions	SID retrieved by get	SID of delta to be created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R, and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1
Footnotes *, †, ‡, §, and ** on next page.				

Footnotes to Table 11-1, "Determination of new SID" (page 357):

- * R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. For example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (that is, the maximum branch number plus 1) of level L within release R. Note that, if the SID specified is R.L, R.L.B, or R.L.B.S, then each of these specified SID numbers must exist.
- † The **-b** option is effective only if the SCCS file **b** flag is present. An entry of **-** means 'irrelevant'. See **admin(CP)** for a discussion of the **b** flag.
- ** The release hR is the highest existing release that is lower than the specified, nonexistent release R.
- ‡ This case applies if the SCCS file **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as though specified on the command line. Therefore, one of the other cases in this figure applies.
- § This forces the creation of the first delta in a new release.

Concurrent edits of same SID

Under normal conditions, more than one **get -e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get -e** is executed on the same SID.

Multiple concurrent edits are allowed if the SCCS file **j** flag is set. For example:

```
get -e s.abc
```

This produces:

```
1.1  
new delta 1.2  
5 lines
```

This can be immediately followed by (without an intervening **delta**):

```
get -e s.abc
```

This produces:

```
1.1  
new delta 1.1.1.1  
5 lines
```

In this case, a **delta** after the first **get** produces delta 1.2 (assuming that 1.1 is the most recent trunk delta), and a **delta** after the second **get** produces delta 1.1.1.1.

Key letters that affect output

The **get -p** command causes the retrieved text to be written to the standard output, rather than to a g-file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. The **get -p** command creates a g-file with an arbitrary name, as in:

```
get -p s.abc > arbitrary-filename
```

The **get -s** command suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. The **get -s** command prevents nondiagnostic messages from appearing on the user's terminal and is often used with **-p** to pipe the output, as in:

```
get -p -s s.abc | pg
```

The **get -g** command suppresses the retrieval of the text of an SCCS file. This is useful in several ways. It may be used to verify a particular SID in an SCCS file. For example:

```
get -g -r4.3 s.abc
```

This outputs the SID 4.3 if it exists in the SCCS file *s.abc*, or an error message if it does not.

Another use of **get -g** is in regenerating a p-file that may have been accidentally destroyed, as in:

```
get -e -g s.abc
```

The **get -l** command causes SCCS to create an 'l-file'. It is named by replacing the *s.* of the SCCS filename with *l.*, created in the current directory with mode 444 (read only) and owned by the real user. The l-file contains a table showing the deltas used in constructing a particular version of the SCCS file. See **get(CP)** for a description of the delta table format.

The following command generates an l-file, showing the deltas applied to retrieve version 2.3 of file *s.abc*.

```
get -r2.3 -l s.abc
```

Specifying **p** with **-l**, for example, results in the output being written to the standard output, rather than to the l-file.

```
get -lp -r2.3 s.abc
```

The **get -g** command can be used with **-l** to suppress the retrieval of the text. For example:

```
get -g -l s.abc
```

The **get -m** command identifies the changes applied to an SCCS file. Each line of the g-file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

The **get -n** command causes each line of a g-file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep(C)**. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following can be executed:

```
get -p -n -s directory | grep pattern
```

If both **-m** and **-n** are specified, each line of the generated g-file is preceded by the value of the **%M%** ID keyword and a tab (the effect of **-n**), and is followed by the line in the format produced by **-m**. Because use of **-m** and/or **-n** causes the contents of the g-file to be modified, such a g-file must not be used for creating a delta. Therefore, neither **-m** nor **-n** may be specified together with **get -e**.

NOTE See **get(CP)** for a full description of additional key letters.

delta command

The **delta(CP)** command incorporates changes made to a g-file into the corresponding SCCS file (that is, to create a delta and, therefore, a new version of the file).

The **delta** command requires the existence of a p-file (created by means of **get -e**). It examines the p-file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

The **delta** command performs the same permission checks that **get -e** performs. If all checks are successful, **delta** determines what has been changed in the g-file by comparing it, using **diff(C)**, with its own temporary copy of the g-file as it was before editing. This temporary copy of the g-file is called the d-file, and is obtained by performing an internal **get** on the SID specified in the p-file entry.

The required p-file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g-file must be the one who creates the delta. However, in the case of multiple concurrent edits, if the login name of the user appears in more than one entry, then the same user has executed **get -e** more than once on the same SCCS file. Then, **delta -r** must be used to specify the SID that uniquely identifies the p-file entry. This entry becomes the one used to obtain the SID of the delta to be created.

In practice, the following is the most common use of **delta**:

```
delta s.abc
```

This command results in the following prompt:

```
comments?
```

The user replies with a description of why the delta is being made, ending the reply with a new-line character. The user's response may be up to 512 characters long with new-lines (not intended to terminate the response) escaped by backslashes (\).

If the SCCS file **v** flag is present, **delta** first prompts with:

```
MRs?
```

Modification Requests (MRs), is a formal way of asking for corrections or enhancements to a file. After the MRs is specified, the standard output is then read for MR numbers. These are separated by blanks and/or tabs, ending with a new-line character. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report/ticket, change request, and so on, collectively known as MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change-management process.

The **delta -y** and/or **delta -m** commands can be used to enter comments and MR numbers on the command line rather than through the standard input, for example:

```
delta -y"descriptive comment" -m"mrnum1 mrnum2" s.abc
```

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two key letters are useful when **delta** is executed from within a shell procedure. See **sh(C)**.

NOTE The **delta -m** command is allowed only if the SCCS file **v** flag is present.

All comments and MR numbers, whether solicited by a **delta** or supplied by keys, are recorded as part of the entry for the delta being created. They are applicable to all SCCS files specified with the same invocation of the **delta**.

If **delta** is used with more than one file argument and the first file named has an SCCS file **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p-file) and the number of lines inserted, deleted, and left unchanged. Here is an example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a g-file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g-file. However, the total number of lines in the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g-file.

If, in the process of making a delta, **delta** finds no ID keywords in the edited g-file, the following message is issued after the prompts for commentary but before any other output:

```
No id keywords (cm7)
```

This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by:

- making a delta from a g-file that was created by a **get** without **-e**. (ID keywords are replaced by **get** in such a case.)
- accidentally deleting or changing ID keywords while editing the g-file.
- the file having no ID keywords.

In any case, the delta is created unless there is an SCCS file **i** flag present (meaning the error should be treated as fatal), in which case the delta is not created.

After the processing of an SCCS file is complete, the corresponding p-file entry is removed from the p-file. All updates to the p-file are made to a temporary copy, the 'q-file,' whose use is similar to that of the x-file (page 348). If there is only one entry in the p-file, then the p-file itself is removed.

In addition, **delta** removes the edited g-file unless **-n** is specified. For example, the following command keeps the g-file after processing:

```
delta -n s.abc
```

The **delta -s** command suppresses all output normally directed to the standard output, other than **comments?** and **MRs?** prompts. Thus, use of **-s** with **-y** (and/or **-m**) causes **delta** not to read the standard input or write the standard output.

The changes made to the g-file constitute the delta and may be printed on the standard output by using **delta -p**. The format of this output is similar to that produced by **diff(C)**.

admin command

The **admin**(CP) command administers SCCS files; that is, it creates new SCCS files and changes the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of key letters with **admin**, or are assigned default values if no key letters are supplied. The same key letters are used to change the parameters of existing SCCS files.

Two key letters are used in detecting and correcting corrupted SCCS files. (See "Auditing" (page 373).)

Newly created SCCS files are given access permission mode 444 (read only), and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS files

An SCCS file can be created by executing the following command:

```
admin -ifirst s.abc
```

The value **first** with the **-i** option is the name of the file from which the text of the initial delta of the SCCS file, *s.abc*, is to be taken. Omission of a value with **-i** indicates that **admin** is to read the standard input for the text of the initial delta.

The following command is equivalent to the previous example:

```
admin -i s.abc < first
```

If the text of the initial delta does not contain ID keywords, **admin** issues the following warning message:

```
No id keywords (cm7)
```

However, if the command also sets the SCCS file **i** flag (not to be confused with the **-i** command line option), then the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin -i**.

The **-r** option used in conjunction with **admin** creates a release number for the first delta.

```
admin -ifirst -r3 s.abc
```

This command specifies that the first delta should be named 3.1 rather than the normal 1.1. Because **-r** has meaning only when creating the first delta, its use is permitted only with **-i**.

Inserting commentary for the initial delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin -y**) and/or MR numbers (**-m**) can be entered in exactly the same way as a **delta**.

If the **-y** option is omitted, a comment line of the following form is generated:

```
date and time created yy/mm/dd hh:mm:ss by logname
```

If it is desired to supply MR numbers (**admin -m**), the SCCS file **v** flag must be set by means of the **-f** command line option. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary. For example:

```
admin -ifirst -mmrnum1 -fv s.abc
```

NOTE **-y** and **-m** are effective only if a new SCCS file is being created.

Initialization and modification of SCCS file parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin -t**.

When an SCCS file is first being created and **-t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the following command specifies that the descriptive text is to be taken from file *desc*.

```
admin -ifirst -tdesc s.abc
```

When processing an existing SCCS file, **-t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Omission of the filename after the **-t** key letter results in the removal of the descriptive text from the SCCS file. For example:

```
admin -t s.abc
```

The flags of an SCCS file may be initialized or changed by **admin -f** or deleted by means of **-d**.

SCCS file flags direct certain actions of the various commands. (See **admin(CP)** for a description of all the flags.) For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

The **admin -f** command sets flags and, if desired, their values. In the following example, the **i** and **m** (module name) flags are set.

```
admin -ifirst -fi -fmmodname s.abc
```

The value *modname* specified for the **m** flag is the value that the **get** command uses to replace the **%M%** ID keyword. (In the absence of the **m** flag, the name of the g-file is used as the replacement for the **%M%** ID keyword.) Several **-f** key letters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

The **admin -d** command deletes a flag from an existing SCCS file. For example:

```
admin -dm s.abc
```

This command removes the **m** flag from the SCCS file. Several **-d** key letters may be used with one **admin** and may be intermixed with **-f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), use **admin -a**. For example:

```
admin -axyz -awql -a1234 s.abc
```

This adds the login names *xyz* and *wql* and the group ID **1234** to the list. The **admin -a** command may be used whether creating a new SCCS file or processing an existing one.

The **admin -e** command (erase) removes login names or group IDs from the list.

prs command

The **prs(CP)** command prints all or part of an SCCS file on the standard output. If **prs -d** is used, the output is in a format called data specification. Data specification is a string of SCCS file data key words (not to be confused with **get** ID keywords) interspersed with optional user text.

Data keywords are replaced by appropriate values, according to their definitions.

```
:I:
```

In this example, **I** is defined as the data keyword replaced by the SID of a specified delta. Similarly, **:F:** is the data keyword for the SCCS filename currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data keyword. For a complete list, see **prs(CP)**.

There is no limit to the number of times a data keyword can appear in a data specification. For example:

```
prs -d":I: this is the top delta for :F: :I:" s.abc
```


This produces the following output:

```
2.1 this is the top delta for s.abc 2.1
```

Information can be obtained from a single delta by specifying its SID using **prs -r**. For example:

```
prs -d":F:: :I: comment line is: :C:" -r1.4 s.abc
```

This produces the following output:

```
s.abc: 1.4 comment line is:THIS IS A COMMENT
```

If **-r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **-l** or **-e**. Using **prs -e** substitutes data keywords for the SID designated by means of **-r** and all deltas created earlier, while **prs -l** substitutes data keywords for the SID designated by means of **-r** and all deltas created later. For example:

```
prs -d:I:-r1.4 -e s.abc
```

This produces output such as:

```
1.4  
1.3  
1.2.1.1  
1.2  
1.1
```

Another example is:

```
prs -d:I:-r1.4 -l s.abc
```

This produces output such as:

```
3.3  
3.2  
3.1  
2.2.1.1  
2.2  
2.1  
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both **-e** and **-l**.

sact command

The **sact**(CP) command is like a special form of the **prs** command that produces a report about files that are out for editing. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for editing, the SID of the impending delta, the login of the user who executed the **get -e** command, and the date and time the **get -e** was executed. It is a useful command for an administrator.

rmDEL command

The **rmDEL**(CP) command allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 11-3, "Extended branching concept" (page 347), only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The **-r** key letter is mandatory with **rmDEL**. It specifies the complete SID of the delta to be removed. For example, to specify the removal of trunk delta 2.3, use the command:

```
rmDEL -r2.3 s.abc
```

Before removing the delta, **rmDEL** checks that the release number (%R%) of the given SID satisfies the relation:

floor less than or equal to R less than or equal to ceiling

The **rmDEL** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the SCCS file **l** flag is set, the release must not be contained in the list. (See **admin**(CP).) If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

cdc command

The **cdc**(CP) command changes the commentary made when the delta was created. It is similar to the **rmDEL** command (for instance, **-r** and full SID are necessary), although the delta need not be a leaf delta.

```
cdc -r3.4 s.abc
```

The above example specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for, as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new MR numbers as well as the deletion of old MR numbers (using the **!** prefix). The following command inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

cdc -r1.4 s.abc

MRs? mrnum3 !mrnum1

(The 'MRs?' prompt appears only if **v** flag has been set.)

comments?

**deleted wrong MR number and
inserted correct MR number**

NOTE An MR is described in "delta command" (page 360).

what command

The **what**(CP) command finds identifying information within any UNIX system file whose name is given as an argument. No key letters are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword. (See **get**(CP).) It prints on the standard output whatever follows the string until the first double quote (**"**), greater than (**>**), backslash (****), new-line, or nonprinting NULL character.

For example, an SCCS file called *s.prog.c* (a C language program) contains the following line:

```
char id[] = "%W%";
```

After the following command is used, the resulting g-file could be compiled to produce *prog.o* and *a.out*:

get -r3.4 s.prog.c

Then, the following command is executed:

what prog.c prog.o a.out

```
prog.c:
prog.c: 3.4
prog.o:
prog.c: 3.4
a.out:
prog.c: 3.4
```

The string searched for by **what** need not be inserted by means of an ID keyword of **get**; you can insert it in any convenient manner.

sccsdiff command

The **sccsdiff**(CP) command determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff -r** in the same way as with **get -r**. SID numbers must be specified as the first two arguments. Any following key letters are interpreted as arguments to the **pr**(C) command (which prints the differences), and must appear before any filenames. The SCCS file(s) to be processed are named last. Directory names and a name '-' (a single minus sign) are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

```
sccsdiff -r3.4 -r5.6 s.abc
```

The differences are printed the same way as by **diff**(C).

comb command

The **comb**(CP) command lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh**(C)). on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recommended that **comb** be used as a matter of routine.)

In the absence of any key letters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 11-3, "Extended branching concept" (page 347) deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the key letters options used with this command are as follows:

- comb -s** generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.
- comb -p** specifies the oldest delta the user wants preserved.
- comb -c** specifies a list of deltas the user wants preserved. All other deltas are discarded. (See **get**(CP) for its syntax.)

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree can be altered by the reconstruction process.

val command

The **val**(CP) command determines whether a file is an SCCS file meeting the characteristics specified by certain key letters. It checks for the existence of a particular delta when the SID for that delta is specified with **-r**.

The string following **-y** or **-m** checks the value set by the **t** or the **m** SCCS file flags, respectively. See **admin**(CP) for descriptions of these flags.

The **val** command treats the special argument — differently than other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line: the standard input is read until an end-of-file ((Ctrl)d) is entered. This permits one **val** command with different values for key letters and file arguments. For example:

```
val -yc -mabc s.abc -mxyz -ypl1 s.xyz
```

val first checks whether file *s.abc* has a value *c* for its type flag and value *abc* for the module name flag. Once this is done, **val** processes the remaining file, in this case, *s.xyz*.

The **val** command returns an 8-bit code. Each bit set shows a specific error. (See **val**(CP) for a description of errors and codes.) In addition, an appropriate diagnostic is printed, unless suppressed by **-s**. A return code of 0 means all files meet the characteristics specified.

vc command

The **vc**(CP) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. See **vc**(CP) for a complete description of the command.

SCCS files

The following topics are discussed in this section:

- Protection Describes the protection mechanisms used by SCCS files
- Formatting Explains the structure and format of SCCS files
- Auditing Recommends procedures for auditing SCCS files

Protection

SCCS relies on the capabilities of the UNIX system for most of the protection mechanisms required to prevent unauthorized changes to SCCS files, that is, changes by non-SCCS commands. The only protection features directly provided by SCCS are:

- release lock flag
- release floor flag
- release ceiling flag
- user list

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories containing only SCCS files and any temporary files created by SCCS. This simplifies their protection and auditing. Directories should contain logical groupings of SCCS files: for example, subsystems of the same large project.

SCCS files must have only one link (name) because commands that modify an SCCS file do so by creating a copy of the file (see “SCCS command conventions” (page 348)). When processing is completed, the x-file is automatically renamed with an *s.* prefix. If the old file had more than one link, the renaming would break them. Rather than process these files, SCCS commands produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; the user ID owns the directories containing SCCS files. Therefore, SCCS can be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities, one user ID should be selected as the owner of the SCCS files. This person is responsible for all administration (**admin**) of the SCCS files. This limits the privileges and permissions allowed to other users. To work around this limitation, it is recommended that a project-dependent user interface be set up allowing other (non-SCCS administrator) users access to the **get**, **delta**, and **rmdel** SCCS commands.

The interface program must be owned by the SCCS administrator and must have the set-user-ID-on-execution bit on. (See **chmod(C)**). This assures that the effective user ID is that of the SCCS administrator. The owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**.

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	Line containing the logical sum of all the characters of the file (not including the checksum itself).
Delta table	Information about each delta, such as type, SID, date and time of creation, and commentary.
User names	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of SCCS commands.
Descriptive text	Usually a summary of the contents and purpose of the file.
Body	Text administered by SCCS, intermixed with internal SCCS control lines.

Details on these file sections can be found in **sccsfile(FP)**. The checksum is discussed next under “Auditing” (page 373).

Since SCCS files are ASCII files, they can be processed by non-SCCS commands like **ed**(C), **grep**(C), and **cat**(C). This is convenient when an SCCS file must be modified manually (such as when a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.

NOTE Be careful when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command issues an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed (possibly by having lost one or more blocks or by having been modified with **ed**(C)). No SCCS command processes a corrupted SCCS file except the **admin** command with **-h** or **-z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin -h** and specify all SCCS files. Either command works:

```
admin -h s.file1 s.file2 ...
admin -h directory1 directory2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the following message is produced for that file:

```
corrupted file (co6)
```

The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process does not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(C) command periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX system operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

```
admin -z s.file
```

The purpose of this command is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file is no longer detectable.

Chapter 12

lex

The software tool **lex**(CP) lets you quickly generate solutions to problems that involve *lexical analysis*, that is, the recognition of strings of characters that satisfy certain characteristics. This enables you to solve a wide class of problems drawn from text processing, code enciphering, compiler writing, and other areas. For example:

- in text processing, checking the spelling of words for errors
- in code enciphering, translating certain patterns of characters into others
- in compiler writing, determining what the tokens (smallest meaningful sequences of characters) are in the program to be compiled

It is not essential to use **lex** to handle problems of this kind: you could write programs in a standard language like C to handle them. What **lex** does is generate such C programs, based on a set of specifications that you give it. These **lex** specifications name and describe the classes of strings that you wish to recognize, and often give actions to be carried out when a particular kind of string is found. **lex** is referred to as a “program generator”: more specifically, it is a “lexical analyzer generator”. It offers a faster, easier way to create programs to perform lexical analysis. Its weakness is that it often produces C programs that are longer and execute more slowly than hand-coded programs that do the same task. In many applications size and speed are minor considerations, and the advantages of using **lex** considerably outweigh these disadvantages.

To understand what **lex** does, refer to Figure 12-1, “Creation and use of a lexical analyzer with **lex**”. It begins with a **lex** specification, sometimes referred to as a **lex** source program. The source is read by the **lex** program generator. The output of the program generator is a C program which, in turn, must be compiled in order to generate an executable program that performs the lexical analysis. The lexical analyzer program produced by this process accepts as input any source file and produces the specified output, such as altered text or a list of tokens.

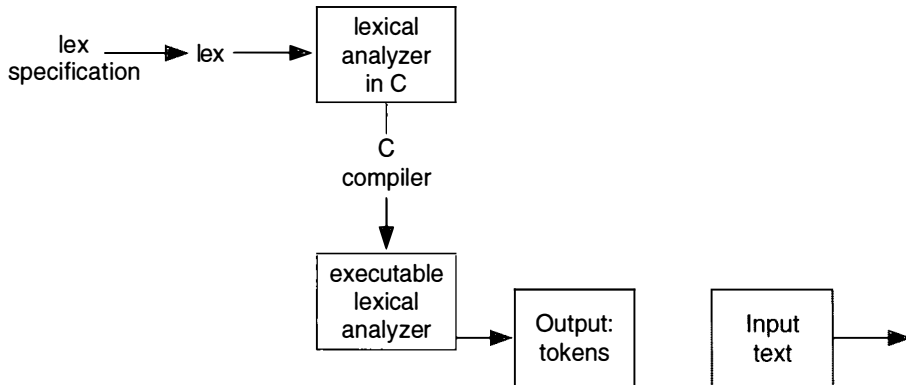


Figure 12-1 Creation and use of a lexical analyzer with **lex**

Programs generated by **lex** can also be used to collect statistical data on features of the input, such as character count, word length, and the number of occurrences of particular words. In later sections of this chapter, you will see how to:

- write **lex** specifications to perform some of these tasks
- translate **lex** specifications into C
- compile, link, and execute the lexical analyzer in C
- run the lexical analyzer program

Writing lex programs

A **lex** specification consists of a mandatory rules section, and optional sections for definitions and user subroutines.

The definitions section, if present, must be the first section in the **lex** program. The mandatory rules section follows the definitions; if there are no definitions, then the rules section is first. In both cases, the rules section must start with the delimiter `%%`. If there is a subroutines section, it follows the rules section and is separated from the rules by another `%%` delimiter. If there is no second `%%` delimiter, the rules section is presumed to continue to the end of the program.

The following is a small, though complete, **lex** specification, illustrating all three sections:

```

1      %{
2      int count;
3      %}
4      %%
5      fly          { count++; noise(); }
6      dog          { printf("Woof!\n"); }
7      man          { printf("Hello world!\n"); }
8      %%
9      noise()
10     {
11         printf("Bzzzz!\n");
12     }
```

In this example, the definitions section (lines 1-3) declares a variable which is used as a counter later in the program. The rules section (lines 4-7) consists of three rules, each of which consists of a *pattern*, followed by some C code. The subroutines section (lines 8-12) defines a function that is used in one of the actions.

Rules section

Each rule consists of a pattern to be searched for in the input, followed on the same line by an action to be performed when the pattern is matched. Because lexical analyzers are often used in conjunction with parsers, as in programming language compilation and interpretation, the patterns can be said to define the classes of *tokens* that may be found in the input.

Regular expressions

The patterns describing the classes of strings to be searched for are written using regular expressions in a notation similar to that used in **awk(C)** and **sed(C)**. The terms “pattern” and “regular expression” are often used

interchangeably. A regular expression is formed by concatenating characters and, usually, certain operators. This notation used with `lex` is summarized in the following list:

- A string of text characters with no operators at all just matches the literal string. To match the word “orange”, use:

```
orange
```

- To match a literal string that contains spaces or tabs, surround the expression with double quotes. To match the phrase “red apple”, use the expression:

```
"red apple"
```

- An expression, followed by the “*” operator, matches 0 or more occurrences of that expression. To match a string containing any number of “m”s, or the null string, use the expression:

```
m*
```

- An expression, followed by the “+” operator, matches one or more occurrences of that expression. To match a string containing one or more “m”s, but not the null string, use the expression:

```
m+
```

- An expression, followed by the “?” operator, matches 0 or 1 occurrence(s) of that expression. This is equivalent to saying that the expression is optional. To match one occurrence of the letter “m”, or the null string, use the expression:

```
m?
```

- The period character, (.), matches any single character. To match any five-letter string starting with “m” and ending with “y”, use the expression:

```
m...y
```

- Alternation in regular expressions is supported using the vertical bar, (|). To match either of the strings “love” and “money”, use the expression:

```
love|money
```

- Expressions may be grouped using parentheses, ‘(’ and ‘)’. To match a string that consists of any number of a’s and b’s, followed by a “c”, use the expression:

```
(a|b)*c
```

- The circumflex, (^), followed by a pattern, signifies that the pattern must match at the beginning of a line. The following rule matches the word “First” at the beginning of a line:

```
^First
```

- The dollar sign, (\$) is appended to a pattern to indicate that it must match at the end of a line. The following rule matches the word “cow” at the end of a line:

```
cow$
```

- To indicate that a regular expression should be matched a specific number of times, follow that expression with a number enclosed in curly braces, '{' and '}'. To match three repetitions of “cd”, that is, “cdcdcd”, use the expression:

```
(cd){3}
```

- To specify a range of repetitions, follow the expression by two numbers, separated by a comma and enclosed in curly braces. To match three, four, or five repetitions of “ab”, that is, “ababab”, “abababab”, or “ababababab”, use the expression:

```
(ab){3,5}
```

- A sequence of characters inside square brackets, '[' and ']', matches any one character in the sequence. To match any one of “d”, “g”, “k”, and “a”, use the expression:

```
[dgka]
```

If the circumflex, (^), is the first character inside the square brackets, then the pattern matches any character that does not appear inside the brackets. In this context, the circumflex does not signify the start of a line, as it does when prepended to a pattern. To match any character other than “a”, “b”, and “c”, use the expression:

```
[^abc]
```

- Ranges within a standard alphabetic or numeric order are indicated with a hyphen, (-). The following expression matches any digit, uppercase letter, or lowercase letter:

```
[0-9A-Za-z]
```

- Regular expressions can be concatenated. The resulting expression matches whatever the first expression matches followed by whatever the second expression matches. The following regular expression matches an identifier in many programming languages. An identifier, thus defined, is a letter followed by zero or more letters or digits:

```
[a-zA-Z][0-9a-zA-Z]*
```

- To treat an otherwise special character as a literal character, rather than as a special character, enclose the character in quotation marks or precede it with a backslash (\). Either of the following expressions could be used to match an asterisk followed by one or more digits:

```
\*[0-9]+
"*"[0-9]+
```

To recognize a backslash itself, either of these expressions could be used:

```
\\
"\"
```

- **lex** understands the standard C escape sequences, such as `\n` for the end-of-line.

Actions

An action is a block of C code that is executed whenever the corresponding pattern in the **lex** specification is matched. Once the **lex**-generated lexical analyzer matches a regular expression specified in a rule in the specification, it looks to the right of the rule for the action to be performed. Actions typically involve operations such as a transformation of the matched string, returning a token to a parser, or compiling statistics on the input.

The simplest action contains no statements at all. Input text that matches the pattern associated with a null action is ignored. A sequence of characters that does not match any pattern in the rules section is written to the standard output without being modified in any way. To cause **lex** to generate a lexical analyzer that prints everything in the input text with the exception of the word “orange”, which is ignored, the following rules section is used:

```
%%
orange ;
```

Note that there must be some white space (spaces or tabs) between the pattern and the semicolon.

You may want to print out a message noting that a string of text was found, or a message transforming the text in some way. To recognize the expression “Amelia Earhart”, the following rule can be used:

```
"Amelia Earhart"    printf("found Amelia's bookcase!\n");
```

To replace a lengthy medical term with its acronym, a rule such as this is called for:

```
Electroencephalogram    printf("EEG");
```

In order to count the lines in a text file, the analyzer must recognize end-of-lines and increment a counter. The following rule is used for this purpose:

```
%{
int  lineno=0;
}%
%%
\n  lineno++;
```

NOTE If an action consists of two or more C statements spread over two or more lines, the code must be enclosed in curly braces, '{' and '}'.

yytext, yyleng

When a character string matches some pattern in the **lex** specification, it is stored in a character array called **yytext**. The contents of this array may be operated on by the action associated with the pattern: it can be printed or manipulated as necessary. **lex** also provides a variable **yyleng**, which gives the number of characters matched by the pattern.

For example, the following rule directs the lexical analyzer to count the digit strings in an input text and print the running total, and print out the text of each string as soon as it is found:

```
%{
int  digstringcount=0;
}%
%%
[--]?[0-9]+      {
                    digstringcount++;
                    printf("%d %s\n",digstringcount,yytext);
                }
```

This specification matches negative digit strings, and positive strings whether or not they are preceded by a plus sign; the "?" indicates that the preceding sign is optional.

ECHO

The macro **ECHO** is a shorthand way of printing out the text of the token. The two rules in the next example have the same effect:

```
Jim|James      { ECHO; }
Jim|James      { printf("%s",yytext); }
```


The following **lex** specification draws together several of the points discussed previously.

```

1 %{
2 int subprogcount = 0;
3 int gstringcount = 0;
4 %}
5 %%
6 -[0-9]+          printf("negative integer\n");
7 "+"?[0-9]+       printf("positive integer\n");
8 -0\.[0-9]+       printf("negative real number, no whole number part\n");
9 rail[ ]+road      printf("railroad is one word\n");
10 crook            printf("Here's a crook!\n");
11 function         subprogcount++;
12 G[a-zA-Z]*       {
13                 printf("may have a G word here: %s\n ", yytext);
14                 gstringcount++;
15                 }
```

The first three rules (lines 6-8) recognize negative integers, positive integers, and negative real numbers between 0 and -1. The fourth rule (line 9) matches cases where one or more blanks intervene between the two syllables of the word “railroad”. The fifth specification (line 10) matches the word “crook” and prints a useful warning. The rule recognizing “function” (line 11) increments a counter. The last rule (lines 12-15) illustrates a multiline action, and the use of **yytext**.

Definitions section

The **lex** definitions section may contain any of several classes of items. The most important are external definitions, **#include** statements, and abbreviations.

External definitions

Recall that for legal **lex** source this section is optional, but in most cases some of these items are necessary. External definitions have the same form and function as in C. They declare that variables globally defined elsewhere (perhaps in another source file) can be accessed in your lexical analyzer.

You may want to define a variable for use only within the action associated with one rule, for example, a loop index. You can declare such a variable to be local to a block (in this case, the action sequence) as you normally would in C—by declaring it directly after the left curly brace.

#include statements

The purpose of the **#include** statement is the same as in C: to include files that are important to the lexical analyzer program. An example of this usage occurs when **lex** is used with **yacc**(CP). **yacc** is a program generator that generates parsers, programs that analyze input to ensure that it is syntactically correct. These parsers will call a lexical analyzer. When using **yacc**, the file *y.tab.h*, generated by **yacc**, should be included. This file can contain definitions for token names. **#include** statements and variable declarations must be placed between the delimiters `'%{'` and `'%{'`, for example:

```
%{
#include "y.tab.h"
extern int tokval;
int lineno;
%}
```

Abbreviations

The definitions section can also contain abbreviations for regular expressions to be used in the rules section. The purpose of abbreviations is to avoid needless repetition in writing your specifications and to provide clarity in reading them. A definition for an “identifier” would often be placed in this section.

Abbreviations must appear after the `'%{'` delimiter that ends your **#include** statements and variable declarations (if there are any).

The abbreviation appears on the left of the line and its definition appears on the right, separated by one or more spaces. When abbreviations are used in rules, they must be enclosed within curly braces. The following example illustrates how abbreviations are defined and used:

```
dig      [0-9]
ident    [a-zA-Z][a-zA-z0-9]*
%%
-{dig}+      printf("negative integer\n");
"+"?(dig)+   printf("positive integer\n");
-0\.(dig)+    printf("negative real number, no whole number part\n");
{ident}
```

The subroutines section

Subroutines can be used in a **lex** specification for the same purposes as in other C programs. Code used in actions for several rules can be written once and called when needed. Other reasons to use a subroutine are to highlight some code of interest and to simplify the rules section, even if the code is used in one rule only. As an example, consider the following routine, which is used to compile statistics on an input text:

```
%{
    int the_count = 0;
    int a_count = 0;
    int an_count = 0;
}%
article      a|an|the
%%
{article}     do_article(yytext);
    .
    .
    .
%%
do_article(s)
char *s;
{
    if (!strcmp(s,"a")) {
        a_count++;
        return 0;
    } else if (!strcmp(s,"an")) {
        an_count++;
        return 0;
    } else if (!strcmp(s,"the")) {
        the_count++;
        return 0;
    }
    printf("text not an article: %s\n",s);
    return 1;
}
```

Other examples of subroutines are programmer-defined versions of the I/O routines **input()**, **unput()**, and **output()**, which will be discussed later. Subroutines that may be exploited by many different programs should probably be stored in their own individual file or library to be called as needed.

Advanced lex usage

This section discusses advanced features of **lex** that help the user solve more complex problems that would be difficult to deal with using ordinary programming techniques.

Disambiguating rules

lex follows two rules to resolve ambiguities that may arise from the **lex** specification. These rules are:

- when a string of input text can match two or more rules in a specification, the first rule in the specification is the one which is matched, and the one whose action is executed
- if a string in the input text can match a rule in the specification, but a longer string which has the first string as its prefix will also match a rule, then the longer string matches

A situation of the type addressed by the first rule could arise if a specification had patterns to match keywords and identifiers:

```
%{
#include "y.tab.h"
}%
%%
START                { return(STARTTOK); }
BREAK                { return(BREAKTOK); }
END                  { return(ENDTOK); }
[a-zA-Z][a-zA-Z0-9]* { return(yytext); }
```

The string "START" could be matched by both the first or the fourth rule: because **START** is a reserved word, you want only the action associated with the first rule to be executed. By placing the rule for **START** and the other reserved words before the rule for identifiers, you ensure that reserved words are recognized as such.

The second kind of ambiguity could arise, for example, if the input text was coded in a programming language that had operators that were similar. Part of a **lex** specification for the C language might look like this:

```
"+"      {return(PLUS); }
"++"     {return(INC); }
```

The lexical analyzer should recognize the increment operator "++", not the addition operator "+", when it reads the following statement:

```
i++;
```

Context sensitivity

lex has a number of mechanisms that help deal with problems of context sensitivity.

Trailing context

A potential problem exists when the lexical analyzer must read characters beyond the pattern being sought because it cannot be sure it has found the pattern until some additional information is known about the “context” in which it appears. A classic example of this involves the DO statement in FORTRAN. Consider the following DO statement:

```
DO 50 k = 1 , 20, 2
```

Consider the sequence consisting of the characters preceding the comma in the statement above:

```
DO 50 k = 1
```

Because FORTRAN ignores blanks, this sequence might be equivalent to the assignment statement:

```
D050k = 1
```

It is not possible to know that the “1” is the initial value of the index “k”, and that the characters “DO” are a keyword, not the first two characters of an identifier, until the first comma is read. Therefore the lexical analyzer would not always interpret the string “DO” in the desired way if the **lex** specification contained the rule:

```
DO      {return(DOTOK);}
```

The way to handle this is to use the slash, (/) which signifies that what follows is “trailing context”. Trailing context is a second pattern that is expected to follow the token that is being searched for. The token is not matched unless the trailing context is also matched. The string that matches the trailing context is not stored in **yytext**, because it is not part of the token itself. The pattern to recognize the FORTRAN DO statement could be:

```
DO/[ ]*[0-9]+[ ]*[a-zA-Z0-9]+[a-zA-Z0-9]*=[ ]*[a-zA-Z0-9]+[a-zA-Z0-9]*[ ]*,
```

(To simplify the example, the rule accepts an index name of any length.) The “\$” operator, discussed in the section on regular expressions, is a form of trailing context. Note that this operator is not exactly the same as a newline, **\n**. Consider the specification:

```
hello\n      { printf("%s",yytext);  
world$      { printf("%s",yytext);
```

The token matched by the first rule is printed with the newline still attached as part of the token. The token matched by the second rule is printed without a newline; a newline matched by “\$”, like any trailing context, is not part of the token.

Starting state

lex allows you to set a kind of flag called a starting state that designates certain rules as applying only when that state is active. There are a number of steps that have to be followed to employ this mechanism.

For the following example, assume you are dealing with a programming language in which programs start with the keyword **#go**, end with the keyword **#stop**, and includes the keyword **while**. Assume also that the input text contains pieces of code interspersed with blocks of text. In this case, there needs to be a way for the lexical analyzer to distinguish between the word **while** as a keyword and as an ordinary word.

```

1      %{
2      #include "defs.h"
3      %}
4      %start PROG
5      %%
6      #go          { BEGIN PROG; }
7      <PROG>while  { return(WHILE); }
8      ", "        { return(','); }
9      [a-zA-Z]*   { put_in_tab(yytext); }
10     <PROG>#stop  { ECHO; BEGIN 0; }
11     .
12     .
13     .
14     %%
15     int put_in_tab()
16     {
17     .
18     .
19     .
20     }
```

The line in the declarations section beginning with **%start** (line 4) is necessary to define the state PROG. This state indicates to the lexical analyzer that it is reading code, not text.

The first rule (line 6) determines that when the analyzer reads the keyword **#go** it activates the state PROG by means of the **BEGIN** macro followed by the state name.

A rule is associated with a state by prefacing it with the state name enclosed in angle brackets, '**<**' and '**>**'. A rule that has been so designated is applied if and only if that state is active. According to rule two (line 7), if the state PROG is active, then if the character sequence **while** is seen, a token is returned indicating that it is a keyword.

A rule that is not prefaced by a state name is applied no matter which state, if any, is active. Rule three (line 8) is an example of this. Rule four (line 9) in the example is also not associated with any state. It will, not, however, match the word **while** if the state PROG is active. This is because that word will have matched the pattern in the earlier rule, rule two, according to the first disambiguating rule discussed in “Disambiguating rules” (page 385).

Rule five (line 10) in the specification deactivates the state PROG if that state is active and the keyword **#stop** is read. "BEGIN 0" deactivates the current state and does not make any other state active.

Programming techniques such as flag variables may also be used to mark context-sensitive conditions.

lex I/O routines

Some actions may require reading another character, putting a character back into the input stream, or writing a character to the standard output. **lex** supplies three functions to handle these tasks: **input()**, **unput()**, and **output()**, respectively. **input()** takes no arguments; **unput()** and **output()** take a single character-valued argument.

The following example illustrates the use of **input()** and **unput()**. The subroutine **skipcmnts()** is used to ignore comments in a language like C, where comments occur between **/*** and ***/** :

```

1      %%
2      "/*"                skipcmnts();
3      .
4      .
5      .
6      %%
7      skipcmnts()
8      {
9          for(;;)
10         {
11             while (input() != '*');
12             if (input() != '/') {
13                 unput(yytext[yytextleng-1]);
14             }
15             else
16                 return;
17         }
18     }
```

After the token **/*** is read, the lexical analyzer continues reading characters until an asterisk (*) is found. If the character after the asterisk is a **/**, the function returns. Otherwise, that character is returned to the input stream and the function keeps on reading characters. The important thing to note here is

that the analyzer does not try to match any patterns with the characters that are read by **input()**. When it resumes pattern matching, after the function returns, it starts with the first character in the input stream after the characters read by the subroutine.

There are three other things to note in this example. First, the **unput()** function (which puts back the last character read) is necessary to avoid missing the final `"/"` if the comment ends with a `*/`. In this case, having read an `"*"`, the analyzer finds that the next character is not the terminal `"/'` and must read some more. Second, the expression `yytext[yytextlen-1]` refers to the last character read. Third, this routine assumes that the comments are not nested, as is indeed the case with the C language. If, unlike in C, they are nested in the source text, after reading the first `*/` ending the inner group of comments, the lexical analyzer reads the rest of the comments as if they were part of the input to be searched for patterns.

To handle special I/O needs, such as writing to several files, standard I/O routines in C can be used to rewrite the functions **input()**, **unput()**, and **output()**. These and other programmer-defined functions should be placed in the subroutine section. The new routines will then replace the standard ones. **lex's** **input()** is equivalent to **getchar()**, and **output()** is equivalent to **putchar()**.

Routines to reprocess input

There are a number of **lex** routines that let you handle sequences of characters that are to be processed in more than one way.

yyomore

The text matching a given pattern is stored in the **yytext** array. In general, after the action associated with it is performed, the characters in **yytext** are overwritten with succeeding characters in the input stream to form the next match. The function **yyomore()** prevents this overwriting, and causes the characters matching the next pattern to be appended to those already in **yytext**. This allows you to process in the same action a set of characters associated with two (or more) successive pattern matches. It is useful when one string of characters is significant and a longer string, that includes the first, is significant as well.

Consider a character string bounded by B's and interspersed with one "B" at an arbitrary location. For example:

```
BoomBoxB
```

You may want to count the number of characters between the first and second "B" and add it to the number of characters between the second and third "B", and print the result. (The last "B" is not to be counted.)

Code to do this is:

```
%{
int flag=0;
}%
%%
B[^B]*    { if (flag == 0) {
            flag = 1;
            yymore();
        }
        else {
            flag = 0;
            printf("%d\n",yyleng);
        }
    }
```

The variable **flag** is used to distinguish the character sequence terminating just before the second “B” from that terminating just before the third. Using the example input “BoomBoxB”, the pattern first matches the string “Boom”, causing that string to be put into **yytext**. Next, the pattern matches the string “Box”, which would normally cause only that string to be put into **yytext**. However, **yymore()** was called in the action associated with the previous pattern match, so **yytext** contains “BoomBox”, and **yyleng** consequently equals 7.

yyless

The function **yyless()** resets the end point of the current token. **yyless()** takes a single integer argument: **yyless(*n*)** causes the current token to consist of the first *n* characters of what was originally matched (that is, up to **yytext[*n*-1]**). The remaining **yyleng-*n*** characters are returned to the input stream. Consider the following specification:

```
%%
[A-Z][a-z]*    {if (yyleng>5)
                yyless(5);
            }
[a-z]*         printf("%s",yytext);
```

The lexical analyzer generated from this specification removes the first 5 letters from any word that starts with an uppercase letter.

REJECT

REJECT allows the lexical analyzer to try to match the current token against the remaining patterns in the specification. Its function is the same as if **yyless(0)** were executed (that is, all the characters in the token were returned to the input stream), except that pattern matching resumes at the pattern following the current one, rather than at the first pattern in the specification. If

you want to count the number of occurrences of both the regular expression “snapdragon” and its subexpression “dragon”, the following works:

```
snapdragon    {countflowers++; REJECT;}
dragon       countmonsters++;
```

As an example of one pattern overlapping another, the following counts the number of occurrences of the expressions “comedian” and “diana”, even where the input text has sequences such as “comediana”:

```
comedian      {comiccount++; REJECT;}
diana         princesscount++;
```

Note that the actions here may be considerably more complicated than simply incrementing a counter.

End-of-file processing with *yywrap*

The routine **yywrap()** is used to deal with end-of-file processing. In its default form, **yywrap()** simply returns 1 if the end-of-file has been reached, and 0 otherwise. A user-defined **yywrap()** may be substituted to provide some other action at the end of input. This routine should be linked before the *lex* library.

Using *lex* with *yacc*

If you work on a compiler project or develop a program to check the validity of an input language, you may want to use the UNIX system program tool **yacc**(CP). *lex* can conveniently be used with **yacc** to develop compilers. For a more complete discussion, see Chapter 13, “*yacc*” (page 397). Whether or not you plan to use *lex* with **yacc**, this part is useful because it covers information of interest to all *lex* programmers.

yylex()

No matter what kind of parser you are using, the lexical analyzer generated by *lex* is invoked through a call to the function **yylex()**. **yylex** is an integer-valued function. This name is convenient because the built-in function call that **yacc**-generated parsers use to invoke the lexical analyzer is also **yylex()**.

Returning tokens

If your lexical analyzer is to be used along with a parser, each *lex* action should end with a **return** statement of the form:

```
return token;
```

or

```
return(token);
```

Here, **token** is an integer-valued variable, literal, or macro, or a character enclosed in single quotes. The value returned indicates to the parser what kind of token the lexical analyzer has found. The parser then resumes control

and later makes another call to the lexical analyzer (via a call to `yylex()`) when it needs another token.

The different return values indicate to the parser whether the token is a reserved word, identifier, constant, arithmetic operator, relational operator, or other token type. In some cases, such as reserved words, and sometimes operators, the return value indicates which token was recognized: for example, a value of 16 might indicate the reserved word **else** in a C compiler. In the other cases, especially identifiers and constants, there needs to be another mechanism for telling the parser the exact value of the token. Consider the following portion of **lex** source for a lexical analyzer for a hypothetical programming language:

```

1      %{
2      include "token.defs"
3      extern int tokval;
4      %}
5      %%
6      begin                return(BEGIN);
7      end                  return(END);
8      while                return(WHILE);
9      if                   return(IF);
10     package              return(PACKAGE);
11     reverse               return(REVERSE);
12     loop                 return(LOOP);
13     "("                  return('(');
14     ")"                  return(')');
15     [a-zA-Z][a-zA-Z0-9]* { tokval = put_in_tabl();
16                           return(IDENTIFIER); }
17     [0-9]+               { tokval = put_in_tabl();
18                           return(INTEGER); }
19     \+                   { tokval = PLUS;
20                           return(ARITHOP); }
21     -                    { tokval = MINUS;
22                           return(ARITHOP); }
23     >                    { tokval = GREATER;
24                           return(RELOP); }
25     >=                   { tokval = GREATEREQ;
26                           return(RELOP); }
27     %%
28     put_in_tabl()
29     {
30         .
31         .
32         .
33     }
```

The identifiers **BEGIN**, **END**, and **WHILE**, used as return values would typically be integer-valued macros defined in some header file: in this case, *token.defs* (see line 2). The header file would look something like this:

```
#define BEGIN 1
#define END 2
#define WHILE 3
.
.
.
#define RELOP 12
```

If it becomes necessary to change the value of some token type, then a **#define** statement in the header file is changed. If using **yacc** to generate the parser, you may insert the following statement into the definitions section of your **lex** source:

```
#include "y.tab.h"
```

NOTE **yacc**, with the **-d** option, generates *y.tab.h* on the basis of the **yacc** specification, which includes token declarations. If **lex** and **yacc** are used together, presumably the same token set is used with both. If the file *y.tab.h* is included in the **lex** source, then **yacc** must be run before the **lex**-generated file is compiled.

To indicate the reserved words in the example, the returned integer values suffice. If a token consists of a single character, like the "(" and ")" tokens in lines 13 and 14 of the example, its literal value may be passed to the parser in the "return" statement. For the other token types (lines 15-26), the integer value identifying the token is stored in the programmer-defined variable **tokval**. This variable is globally defined so that the parser as well as the lexical analyzer can access it. If the parser has been generated by **yacc**, the variable **yyval** should be used for this purpose. Parsers generated by **yacc** look in that variable for the value of the token. **yyval** is defined in the file *y.tab.h*. It is an integer variable by default, but it can be redefined as a C union in the **yacc** specification.

Using a symbol table

This example illustrates the use of a symbol table. The symbol table is a data structure that is globally accessible (or at least accessible by the parser); it stores the text of most or all of the tokens found in a particular input and associates them with identifying "indexes". The function **put_in_tabl()** helps maintain the symbol table. This function may be defined in the subroutines section of the **lex** specification, or in an included file. The implementation details of this function are not important here, but it may be understood to carry out the following tasks:

- When a token is recognized, **put_in_tabl()** takes the value in **yytext** and compares it with all the strings currently in the symbol table.

- If it finds that value already in the table, **put_in_tabl()** returns the index of the value.

The index is a unique identifying integer: no other identifier may have the same index. The index could be an array subscript, but the details of this are unimportant to us as long as the value is unique.

- If the current token does not already appear in the symbol table, a new entry is created for it and an index is generated. (The function may store some other information in the symbol table such as whether the token is an identifier, constant, or other token type.) **put_in_tabl()** then returns the index.

Different parts of the program know about the text of an identifier by way of the index stored in **tokval**. For example, when the parser receives a value from the lexical analyzer indicating that an identifier (line 15) has been recognized, it can find the string in the symbol table that has an index equal to **tokval**.

Note that the example shows two ways to assign a value to **tokval**. **put_in_tabl()** may assign a symbol table index to **tokval**. In the last few actions of the example (line 15-26), **tokval** is assigned a specific integer (in the form of a defined macro) indicating which operator the analyzer recognized. For example, in the last rule in the specification (lines 25 and 26), the lexical analyzer indicates the general class of operator by returning the value **RELOP** to the parser. This enables the parser to check the input for syntactic correctness. Further, the lexical analyzer indicates the specific operator by assigning the value **GREATEREQ** to **tokval**. The parser has some way of associating this value with the string **>=**. Quite possibly the symbol table will have a set of base entries that includes **>=**. The definition of **GREATEREQ**, and the other predefined token indexes will not necessarily appear in the same set of definitions as the token types.

Using lex under UNIX systems

This section explains how to run **lex** to create C code from **lex** specifications, then how to compile and run the lexical analyzer program.

Running lex

If *lex.l* is the file containing the **lex** specification, the C source for the lexical analyzer is produced by running **lex** with the following command:

```
lex lex.l
```

lex produces a C file called *lex.yy.c*.

Options

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument.

The **-t** option sends **lex**'s output to the standard output rather than to the file *lex.yy.c*.

The **-v** option prints out a small set of statistics describing the so-called finite automata that **lex** produces with the C program *lex.yy.c*. (For a detailed account of finite automata and their importance to **lex**, see the Aho, Sethi, and Ullman book, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.)

lex uses a table (a two-dimensional array in C) to represent its finite automaton. The maximum number of states that the finite automaton requires is set by default to 4000. If your **lex** source has a large number of rules or the rules are very complex, this default value may be too small. You can enlarge the value by placing the following entry in the definitions section of your **lex** source:

```
%n 6000
```

This entry tells **lex** to make the table large enough to handle as many as 6000 states. (The **-v** option indicates how large a number you should choose.) If you have need to increase the maximum number of state transitions beyond 16000, the designated parameter is **a**, for example:

```
%a 20000
```

See **lex(CP)** for a concise description of the command and a list of all the options available.

Compiling a lexical analyzer

The file *lex.yy.c* may be compiled and linked in the same way as any C program. The **-ll** option is used to link the object file created from this C source with **lex** library:

```
cc lex.yy.c -ll
```

The **lex** library provides a default **main()** program that calls the lexical analyzer under the name **yylex()**, so you do not have to supply your own **main()**.

If you have the **lex** specification spread across several files, you can run **lex** on each of them individually, but be sure to rename or move each *lex.yy.c* file before you run **lex** on the next one. Otherwise, each file overwrites the previous one. Once you have generated all the C files, you can compile all of them in one command line.

To compile and link the output files produced by **lex** and **yacc**, run:

```
cc lex.yy.c y.tab.c -ly -ll
```

Note that the **yacc** library is linked (with the **-ly** option) before the **lex** library (with the **-ll** option) to ensure that the **main()** program supplied will call the **yacc** parser.

Running the lexical analyzer

By default, the lexical analyzer takes input from the standard input. To have it take input from a file, use redirection; for example:

```
a.out < text.in
```

a.out is the executable lexical analyzer.

Output is sent to the standard output. You can redirect this as well:

```
a.out < text.in > text.out
```

Using make with lex

The **make**(CP) utility can be used to maintain programs that involve **lex**. **make** assumes that a file that has an extension of *.l* is a **lex** source file. It knows how such a file must be processed to create an object file.

Suppose that a list of dependencies in a makefile contains a filename *x.o*, and there exists a file *x.l*. If *x.l* was modified later than the file *x.o* (or if *x.o* does not exist), then **make** will cause **lex** to be run on *x.l*, and then cause the object file *x.o* to be created from the resulting *lex.yy.c*. The **make** internal macro **LFLAGS** can be used to specify **lex** options to be invoked automatically by **make**. For more information, see Chapter 10, “make” (page 315).

Chapter 13

yacc

yacc(CP) is a “parser generator”. It generates C code that can be compiled to produce a parser, which is a program that examines the input stream and, possibly among other things, checks whether it is syntactically correct according to a given grammar.

yacc’s input typically consists of:

- a grammar — a set of rules describing the expected syntactic structure of the input to the parser
- actions — some C code to be invoked when a rule is recognized
- auxiliary declarations and subroutines

A **yacc**-generated parser calls a low-level input scanner, called a lexical analyzer. This routine reads the input stream and separates it into items called “tokens”. The sequence of tokens that the parser receives from the lexical analyzer is compared against the grammar rules. When a rule is recognized, an action (code that the user has supplied for this rule) is executed. Actions can return values, use values returned by previous actions, and carry out any other of the operations possible in C.

The nucleus of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, the following rule defines a symbol “date” in terms of other symbols “month”, “day”, and “year”:

```
date : month day ',' year ;
```

Input such as the following would be matched by this rule:

```
April 16, 1961
```


The symbols to the right of the colon have been defined as tokens, defined in other rules in the specification, or else are literals such as the comma in the rule above. In the example, the comma is enclosed in single quotes, indicating that the comma is to appear literally in the input. The colon and semicolon serve as punctuation in the rule and have no significance in evaluating the input.

This chapter discusses the following topics:

- yacc specifications
- the yacc environment and special features of the parsers yacc produces
- lexical analysis
- the parser operation
- ambiguities
- operator precedences in arithmetic expressions
- error detection and recovery
- suggestions to improve the style and efficiency of the specifications
- examples of yacc usage

yacc specifications

A yacc specification consists of a mandatory rules section, and optional sections for definitions and user subroutines.

The declarations section for definitions, if present, must be the first section in the yacc program. The mandatory rules section follows the definitions; if there are no definitions, then the rules section is first. In both cases, the rules section must start with the delimiter `%%`. If there is a subroutines section, it follows the rules section and is separated from the rules by another `%%` delimiter. If there is no second `%%` delimiter, the rules section continues to the end of the file.

When all sections are present, a specification file has the format:

```

declarations
%%
rules
%%
subroutines
```

The example that follows is a complete yacc specification. The sample program generates a parser which takes input in the form:

```
month day , year
```

This input is converted to output in the form:

```
day month year
```

In the example, the declarations section defines a data structure used to hold the values associated with tokens, and declares all the token names used in the rules section. The rules section contains one rule and an action associated with it. The subroutines section defines a function that is called in the action.

```
%union
{
char *text;
int ival;
}
%token t_DAY
%token t_MONTH
%token t_YEAR
%%
date :      t_MONTH t_DAY ',' t_YEAR
      { print_date ($2,$1,$4); };
%%
void print_date(d,m,y)
char *m;
int d,y;
{
    printf("%d %s %d\n",d,m,y);
}
```

The parser uses a lexical analyzer that can return the tokens *t_DAY*, *t_MONTH*, and *t_YEAR*, and also can associate each token with some value. The mechanisms by which this may be implemented are discussed later in this chapter, and also in Chapter 12, “lex” (page 375).

NOTE While the lexical analyzer may be included as a routine defined in the specification file, it is more usual to define it elsewhere.

Blanks, tabs, and newlines are ignored in a yacc specification, but they cannot appear in names or in multicharacter reserved symbols. Comments can appear wherever a symbol is legal. They are enclosed in */* ... */*, as in C.

Terminal and non-terminal symbols

The tokens that are recognized by the lexical analyzer and passed on to the parser are referred to as “terminal symbols”, because they cannot be broken down into smaller units. The terms “token” and “terminal symbol” are essentially synonymous. By contrast, symbols that can be broken down into other symbols are called “non-terminal symbols”. Symbols of each type are also called “names”.

Rules

The rules section is made up of one or more grammar rules, each of which has the form:

```
lname : definition ;
```

The purpose of the rule is to define *lname* in terms of other symbols. In the example, *lname* is a non-terminal symbol, as are all symbols that appear to the left of the colon in a rule. The definition of *name* can consist of a sequence of terminal symbols, a sequence of other non-terminal symbols, or a sequence of both terminal and non-terminal symbols. Non-terminal symbols that appear in the definition of another symbol are still regarded as non-terminal symbols in that context. The colon and the semicolon are yacc delimiters: the colon separates the non-terminal symbol on the left from its definition and the semicolon must be the last character in the rule.

If actions (page 402) are associated with the rule, they can appear between the colon and the semicolon.

Symbols can be any length and are composed of letters, dots, underscores, and digits, although a digit cannot be the first character of a symbol. Upper-case and lowercase letters are distinct. The NULL character (\0 or 0) should never be used in a grammar rule.

Literals and escape sequences

A literal in a definition consists of a character enclosed in single quotes ('). Literal characters must be passed to the parser by the lexical analyzer, and are considered to be tokens. As in C, the backslash (\) is an escape character within literals; all the C escape sequences are recognized. The following escape sequences are understood by yacc:

<code>'\n'</code>	newline
<code>'\r'</code>	return
<code>'\''</code>	single quote (')
<code>'\\'</code>	backslash (\)
<code>'\t'</code>	tab
<code>'\b'</code>	backspace
<code>'\f'</code>	form feed
<code>'\nnn'</code>	a character in octal notation

Alternation in grammar rules

If there are several grammar rules with the same left-hand side, the vertical bar (|) can be used to combine them, thereby eliminating the need to rewrite the left-hand side. The semicolon at the end of a rule is dropped before a vertical bar. The following constructions are equivalent:

```
A : B C D ;
A : E F ;
A : G ;
```

and

```
A : B C D
   | E F
   | G
   ;
```

It is not necessary for all grammar rules with the same left side to appear together in the grammar rules section, although if they do, the input is more readable and easier to change.

Empty rules

If a non-terminal symbol matches the empty string, this can be indicated by an empty definition, such as the following:

```
epsilon : ;
```

Representing constructs as tokens or non-terminals

Depending on the situation, you can recognize constructs using either the lexical analyzer or grammar rules. For example, the following rules can be used to define the symbol *month*:

```
month : 'J' 'a' 'n' ;
month : 'F' 'e' 'b' ;
.
.
.
month : 'D' 'e' 'c' ;
```

In this example, *month* is a non-terminal symbol. The lexical analyzer only needs to recognize individual letters, and therefore may be very simple. However, such low-level rules are wasteful and result in a complicated specification. To avoid this problem, have the lexical analyzer recognize strings such as “January” and return an indication that a “month” token was seen. In that case, “month” is a terminal symbol and the detailed rules are not needed.

Erroneous specifications

In some cases, **yacc** fails to produce a parser when given a set of specifications. The specifications may be self-contradictory or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter often can be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules.

Actions

An action is a set of C statements enclosed in curly braces, '{' and '}'. The user can associate one or more actions with each grammar rule; these actions are performed when the rule is recognized. An action can appear anywhere in the list of symbols defining a rule, including before the first symbol; usually, actions follow the final symbol in the definition. Actions can return values, obtain the values returned by previous actions, and use values for tokens returned by the lexical analyzer. They can also carry out other tasks that can be programmed in C, such as input and output operations, calls to subroutines, and modification of variables.

The following are examples of grammar rules with actions:

```
A      :  '(' B  ') '
      {
          hello( 1, "abc" );
      }
      ;
```

and:

```
XXX    :  YYY ZZZ
      {
          printf("a message\n");
          flag = 25;
      }
      ;
```

Values of symbols

Each symbol in a grammar rule, including the one to the left of the colon, may have some value associated with it. For a terminal symbol, this value can be assigned by the lexical analyzer (for example, the literal value of an identifier). Non-terminal symbols recognized by the parser can have values associated with them by parser actions. These values can be numbers, text, or another kind of data structure.

The dollar-sign symbol (\$) is used in actions to access the value of a symbol. The pseudo-variable \$\$ represents the value returned by the action. For example, the following action returns the value '1'.

```
{ $$ = 1; }
```

If the action follows the final symbol in the definition, then \$\$ is the value associated with the symbol to the left of the colon, and is that symbol's value when it appears to the right of the colon in another grammar rule.

To obtain the values returned by previous actions and the lexical analyzer, the action can use the pseudo-variables \$1, \$2, ... \$n. \$n refers to the value of the nth symbol or action to the right of the colon. In the following rule, \$2 is the value returned by C, and \$3 is the value returned by D:

```
A : B C D ;
```

Consider the rule:

```
expr : '(' expr ')' ;
```

It is expected that the value returned by this rule is the value of the *expr* within the parentheses. Since the first component of the action is the literal left parenthesis, the desired result can be obtained with the following action:

```
expr : '(' expr ')'
{
    $$ = $2 ;
}
```

Default action

By default, the value of a rule (that is, the value assigned to the symbol to the left of the colon) is the value of the first element in the definition, (\$1). Thus, grammar rules such as the following example, which has only one symbol to the right of the colon, often need not have an explicit action:

```
A : B ;
```

This example is equivalent to:

```
A : B
{
    $$ = $1;
}
;
```

Actions in the middle of rules

In previous examples, all actions come at the end of rules. Sometimes it is desirable to have an action take place before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well. This action can return a value that is accessible by the actions to the right of it through the usual **\$** mechanism. In turn, it can access the values returned by the symbols or actions to its left. The following example of such a rule sets *x* to 1, and sets *y* to the value returned by *C*.

```
A      :   B
        {
            $$ = 1;
        }
        C
        {
            x = $2;
            y = $3;
        }
    ;
```

The first action is given a value by the assignment to **\$\$**. Because that action is the second component of the list to the right of the colon, its value is referred to in subsequent actions as **\$2**. The value returned by *C*, which would normally have been **\$2**, is now **\$3**.

yacc treats the previous example as if it were written as follows, where *ACT* is an empty action:

```
ACT    :   /* empty */
        {
            $$ = 1;
        }
    ;

A      :   B ACT C
        {
            x = $2;
            y = $3;
        }
    ;
```

Accessing left-context symbols

The following discussion is somewhat advanced and therefore should be given careful examination.

An action associated with the left-hand symbol in a rule may need to refer to values associated with symbols that occurred before the current left-hand symbol. These values are referred to as *left-context values*, because they are associated with symbols that appeared to the left of the current left-hand symbol in another rule in the specification. Consider the following yacc specification for a grammar that recognizes dates:

```
%token t_MONTH t_DAY t_YEAR
%union{
    char *text;
    int ival;
};
%%
date: year day month

month : t_MONTH
      { if (!strcmp($1,"February"))
        if ($0==29 && ($-1)%4!=0)
            printf("Too many days!\n");
      }
      ;

day   : t_DAY
      {
          $$ = $1;
      }

year  : t_YEAR ;
      {
          $$ = $1;
      }
```

In this example the lexical analyzer routine associates an integer value with the tokens *t_DAY* and *t_YEAR*, and a character string with the token *t_MONTH*.

The action associated with the symbol “month” checks whether “February 29” occurs in a non-leap year. To do so, it needs to know what values are associated with the “day” and “year” symbols. These symbols appear to the left of “month” in the first rule in the specification, and so their values are left-context values with respect to the symbol “month”.

There are two constructions for accessing left-context values. The value associated with the symbol immediately to the left of the current left-hand symbol is referred to as **\$0**. Values farther to the left are referred to by constructions

of the form $\$-n$. For example, the pseudo-variable $\$-1$ refers to the value associated with the symbol that is two symbols to the left of “month”. In general, the pseudo-variable $\$-n$ refers to the value associated with the symbol that is $n-1$ symbols to the left of the current symbol.

In the action associated with “month” in the example, $\$0$ refers to the value associated with “day”, and $\$-1$ refers to the value associated with “year”.

Parse trees

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. The following example shows a C function `node` written so that the following call creates a node with label `L` and descendants `n1` and `n2`, and returns a pointer to the newly-created node:

```
node( L, n1, n2 )
```

Then a parse tree can be built by supplying actions such as:

```
expr      :   expr '+' expr
          {
              $$ = node( '+', $1, $3 );
          }
```

Declarations

The declarations section is used to declare and describe constructs that are needed by the parsing mechanism and the actions associated with rules in the rules section.

Tokens

Any token that appears in a rule in the rules section must be declared. There are several ways of doing this: the most common way is by using a `%token` statement. This has the form:

```
%token name1 name2 ...
```

Each name that appears after the keyword `%token` is thereby declared as a token. You can declare several tokens on the same statement, and have several such statements. Tokens can be declared in a similar way using the `%left`, `%right` and `%nonassoc` keywords, discussed in “Precedence” (page 421).

Every name that appears in the rules section, but is not defined in the declarations section, is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Start symbol

The **start** symbol is the top-level non-terminal symbol in the grammar. By default, the start symbol is the left-hand side of the first grammar rule in the rules section. You can declare a start symbol explicitly in the declarations section, using the **%start** keyword:

```
%start somename
```

C declarations

C code to be used by the parser can appear in the declarations section, enclosed between the delimiters `'%{'` and `'%}'`. Declarations made here have global scope, so they are known to the action statements and can be made known to the lexical analyzer. This section is usually used for variable declarations and **#include** statements, though other C code can appear here, as shown in the following example:

```
%{
#include "global.h"
int ival = 0;
%}
```

Names beginning with **yy** should be avoided, because internal variables used by the parser begin with these characters.

Support for arbitrary value types

By default, the values that parser actions associate with symbols are integers. **yacc** can also support values of other types, including structures. You can declare a C union that holds the different kinds of values that symbols can have. The parser maintains a data structure called value stack that is declared to be of this union type. To declare the union, use a declaration in the following form:

```
%union
{
    ... body of union ...
};
```

For example:

```
%union
{
    char *text;
    int ival;
    double dub;
};
```

In addition to the value stack, the external variables **yylval** and **yyval** are declared to have type equal to this union. If **yacc** is invoked with the **-d** option, the union declaration is defined under the name **YYSTYPE** in the file *y.tab.h*.

Once **YYSTYPE** is defined, the union member names must be associated with the various terminal and non-terminal names. This enables **yacc** to automatically associate the right type with the pseudo-variables used in actions so that the resulting parser is type-checked. For non-terminal symbols, this association is done using the **%type** keyword. The following declarations associate symbols with the members of the union in the example above:

```
%token <text> s1 s2
%token <ival> s3
%token <dub> s4
```

To associate a terminal symbol (**token**) with a union member name, the **%token** keyword is normally used. The following declaration associates the tokens *s5* and *s6* with the union member *ival*.

```
%token <ival> s5 s6
```

In some cases, these mechanisms are insufficient. For example, there is no default type for the value returned by an action that occurs in the middle of a rule. Similarly, **yacc** must be told explicitly about the type of left-context values such as **\$0**. In such cases, a type can be imposed by inserting a union member name between angle brackets, '**<**' and '**>**', immediately after the first "**\$**" in a pseudo-variable. The following example shows this usage.

```
stat      :  A
           {
               $<intval>$ = 3;
           }
          B
           {
               fun( $<intval>2, $<other>0 );
           }
          ;
```

Other declarations

The keywords **%left**, **%right**, and **%nonassoc** can replace **%token** in the preceding examples. However, these keywords are used principally to deal with operator precedence and associativity. An understanding of precedence and associativity relies heavily on the discussions which follow, and hence consideration of these topics is delayed until "Precedence" (page 421).

Subroutines

This section contains user-defined routines. Usually the routines appearing here are used by actions that appear in the rules section. Additionally, a parser generated by **yacc** must be supplied with three routines: *yylex* to perform lexical analysis, *yyerror* to deal with error messages, and a *main()* routine. These routines can be supplied in the subroutines section.

The subroutines section is available as a convenience, because any functions that appear there can also be included using an **include** directive in the declarations section, or else linked to the parser module.

The yacc environment

When **yacc** processes a specification, the output is a file of C code, named *y.tab.c*. The function produced by **yacc** is called *yyparse()*, and it is an integer-valued function. When *yyparse* is called, it repeatedly calls *yylex()*, the lexical analyzer supplied to obtain input tokens. If the lexical analyzer returns the end-marker token and the parser accepts the input, *yyparse()* returns the value 0. If an error is detected, *yyparse()* returns the value 1, and no error recovery is possible.

A *main()* routine that calls *yyparse()* must be defined. In addition, a routine called *yyerror()* is needed to print a message when a syntax error is detected.

The user must supply these two routines. A library has been provided with default versions of *main()* and *yyerror()*. The library is accessed by using the **-ly** option to **cc**(CP) or **ld**(CP). The following source code shows the simplicity of these default programs:

```
main()
{
    return (yyparse());
}
and
#include <stdio.h>

yyerror(s)
    char *s;
{
    (void) fprintf(stderr, "%s\n", s);
}
```

The argument to *yyerror()* is a string containing an error message, usually the string `syntax error`. An application may require more sophisticated error reporting. The program should keep track of the input-line number and print it with the message when a syntax error is detected. The external integer variable **yychar** contains the look-ahead token number at the time the error was detected. This may be useful in giving better diagnostics.

The external integer variable **yydebug** is normally set to 0. If not, the parser outputs a verbose description of its actions, including the input symbols read and the parser actions. You can set this variable by using a debugger (**adb**, **sdb**, or **dbxtra**).

Lexical analysis

The user must supply a lexical analyzer to read the input stream and pass tokens to the parser. This analyzer may also need to associate values with the tokens and make these values known to the parser. A parser generated by **yacc** calls an integer-valued function called *yyllex* to provide tokens. The function returns an integer, the “token number”, that represents the kind of token being read. If value is associated with that token, the lexical analyzer should assign it to the external variable **yylval**. A parser generated by **yacc** is designed to look in this variable for the values of symbols that are defined as tokens. The variable **yylval** takes an integer value by default. It can be redefined to take on other kinds of values, as explained in “Support for arbitrary value types” (page 407).

The parser and the lexical analyzer must agree on these token numbers before they can communicate. The numbers can be chosen by **yacc** or the user.

lex and yylval

The **lex**(CP) utility is a program generation tool for constructing lexical analyzers. Lexical analyzers produced by **lex** are designed to work with **yacc** parsers. **lex** generates a lexical analyzer, which is called through a call to a function **yyllex**.

The following example of a **lex** specification for a lexical analyzer can be used with either of the parsers for recognizing dates, generated from the **yacc** specifications given earlier in this chapter.

```

%{
#include <stdlib.h>
#include "y.tab.h"
char *p;
}%
mon      January|February|March|April|May|June|July|August|
September|October|November|December
%%
{mon}      {
                p=(char *)calloc(strlen(yytext)+1,sizeof(char));
                strcpy(p,yytext);
                yylval.text=p;
                return(t_MONTH);
            }
[0-9]{1,2}  {
                yylval.ival=atoi(yytext);
                return(t_DAY);
            }
[0-9]{4}    {
                yylval.ival=atoi(yytext);
                return(t_YEAR);
            }
\,         {
                return ',';
            }

```

The analyzer returns a token *t_MONTH*, *t_DAY*, or *t_YEAR* when it recognizes the corresponding sequence of characters. The lexical analyzer associates a *t_DAY* or *t_YEAR* token with an integer value and a *t_MONTH* token with a character string. The tokens are declared in the **yacc** specification and subsequently defined in the file *y.tab.h*, generated by **yacc** with the **-d** option.

The previous example illustrates the use of **yylval**. This variable is defined as a C union having a member called **text** to point to character strings and a member **ival** to hold an integer value. This definition was performed in the **yacc** specification.

In the action for the first pattern, the lexical analyzer puts the value of the string that it matches into the array **yytext**. A copy is made of **yytext**, and **yylval.text** is assigned a pointer to this value. If a pointer to **yytext** had been assigned to **yylval.text**, a problem could arise because the value in **yytext** could get overwritten by the lexical analyzer by the time **yylval** was used by the parser.

The actions for the second and third patterns convert the matched string to an integer and assign **yylval.ival** this value.

Token numbers

By default, token numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. These definitions are placed in the file *y.tab.h*.

If the user prefers to assign the token numbers, the first appearance of the token name or literal in the declarations section must be followed immediately by a non-negative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. Since the potential for duplication exists here, care must be taken to make sure that all token numbers are distinct.

End-marker token

The end of the input to the parser is signaled by a special token, called the “end-marker”. The end-marker is represented by either a zero or a negative number. Thus, every lexical analyzer should be prepared to return 0 or a negative number as a token upon reaching the end of its input. If the tokens up to, but not including, the end-marker form a construct that matches the start symbol, the parser function accepts the input and returns to its caller after the end-marker is seen. If the end-marker is seen in any other context, it is an error.

It is the responsibility of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end-of-file or end-of-record. The programmer does not have to deal with this if using **lex**, which takes care of this job automatically.

Reserved token names

Avoid using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively. One technique for avoiding this kind of difficulty is to preface all token names with some string, such as **t_**, that is unlikely to appear in any reserved word.

Parsing

This section explains how to create and run the parser once **yacc** is run to produce the C code, and also discusses parser operation.

Compiling

The file *y.tab.c* produced by **yacc** can be compiled with **cc** as follows:

```
cc y.tab.c -ly
```

The **-ly** option causes the object file generated to be linked with the **yacc** library. If you supply your own versions of **main()** and **yyerror()**, link the module containing these routines before the library.

A routine named **yylex()** must be provided to do lexical analysis and return tokens to the parser. This routine can be supplied by the user or generated from a **lex** specification. A lexical analyzer generated **lex** can be linked to the parser using the following command line:

```
cc y.tab.c lex.yy.c -ly -ll
```

lex.yy.c is the C file that **lex** produces. If you are using the library version of **main()**, the **-ly** option must appear before **-ll**, so that the version in the **yacc** library is used.

Using make

The **make** facility can be used to help maintain the parser code. **make** interprets files with the extension *.y* as **yacc** source files. Suppose a list of dependencies in a *makefile* contains a filename *x.o*, and there exists a file *x.y*. If *x.y* has been modified later than *x.o*, or if *x.o* does not exist, then **make** will cause **yacc** to be run on *x.y*, and cause the object file *x.o* to be created from the resulting *y.tab.c*.

Running the parser

The resulting executable file accepts input from the standard input. If, for example, the data to be parsed is in a file called *textin*, it can be processed using the following command line:

```
a.out < textin
```

The parser's output is sent to the standard output. This can be captured in a file.

Parser operation

The algorithm **yacc** uses to go from the specification to the C code for the parser is complex and is not discussed here. The parser itself is relatively simple and an understanding of how it works, while not essential, makes treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite-state machine with a state stack. The parser is also capable of reading and remembering the next input token, called the “look-ahead token”. The current state is always the one on the top of the stack. The states of the finite-state machine are given small integer labels. In addition to the state stack, a parallel stack, the *value stack*, holds the values returned by the lexical analyzer and the actions. Initially, the machine is in state 0 (that is, the state stack contains only state 0) and no look-ahead token has been read.

The machine has only five actions available: **shift**, **reduce**, **goto**, **accept**, and **error**. The **goto** is always performed as a component of a **reduce** action. The parser operates in the following manner:

1. Based on its current state, the parser decides whether it needs a look-ahead token to choose the action to be taken. If so, it calls *yylex()* to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto or popped off the stack, and in the look-ahead token being processed or left alone.

When **yacc** is invoked with the **-v** option, a file called *y.output* is produced with a human-readable description of the parser. The actions referred to in the discussion that follows are taken from such a description file.

Shift action

The **shift** action is the most common one the parser takes. A shift occurs whenever a token is recognized. Whenever a shift action is taken, there is always a look-ahead token. A shift is represented in *y.output* as follows (assume that the current state is 56):

```
LOOP  shift 34
```

This says that in state 56, if the look-ahead token is **LOOP**, then the current state (56) is pushed down on the stack, and state 34 becomes the current state, that is, it is pushed on the stack. In addition, the look-ahead token is cleared, and the variable **yyval** is pushed on to the value stack.

Reduce and goto actions

A **reduce** action takes place when the parser determines that all the items on the right-hand side of some grammar rule have been seen. In a reduce action, all the states that were put on the stack while the right-hand side of the rule was being recognized are popped from the stack. Then a new state is put on the stack, based on the symbol to the left of the rule, the state that is currently at the top of the stack, and sometimes the look-ahead token. Suppose the following rule is being reduced:

A : x y z ;

The **reduce** action depends on the symbol to the left of the colon and the number of symbols on the right-hand side (in this case, three). This reduction first involves popping three states off the top of the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z* and no longer serve any useful purpose. After these states have been popped, the state that is on top of the stack is the one the parser was in before it recognized any of the symbols on the right side of the rule. Next, something similar to a shift of 'A' using this uncovered state is performed. A new state is obtained and pushed onto the stack, and parsing continues. This action is called a **goto**. There are differences between a **goto** and an ordinary shift of a token. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. Sometimes, but not usually, it is necessary for the parser to refer to the look-ahead token to decide whether or not to **reduce**.

In effect, the **reduce** action turns back the clock in the parse, popping the states off the stack so that the stack has the same contents as before the symbols on the right side of the rule were recognized. The parser then treats the symbol on the left side of the rule as if it were an input token and performs an action accordingly. Note that if the rule has an empty right-hand side, no states are popped off the stack.

The **reduce** actions are associated with individual grammar rules. In the *y.output* file, these rules are given small integer numbers, which could lead to some confusion. The following action refers to "grammar rule" 18:

. reduce 18

This action refers to *state* 34:

LOOP shift 34

In any case, the *state*, which is uncovered when symbols are popped off the stack on a **reduce**, will contain an entry such as the following:

A goto 20

If the left side of the current rule consists of the symbol "A", this action causes *state* 20 to be pushed onto the stack.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. When a shift takes place, the external variable **yyval** is copied onto the value stack. A reduction takes place after the action code associated with the rule is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudo-variables **\$1**, **\$2**, and so on, refer to the value stack.

Accept action

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker, and indicates that the parser has done its job.

The occurrence of **accept** can be simulated in an action by use of the macro **YYACCEPT**. The **YYACCEPT** macro causes **yyparse()** to return the value 0, indicating a successful parse. Here is an example of its use:

```
quest    :    wealth
          |    love
          |    holygrail
          {
              YYACCEPT;
          }
        ;
```

Error action

The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. Error recovery is discussed later.

The **error** parser action can be simulated in a action code by use of the macro **YYERROR**. **YYERROR** causes the parser to behave as if the current input symbol had been a syntax error. The function **yyerror()** is called and error recovery takes place. Here is an example of such usage:

```
seq:      first second third
        {
            if ($1<$2 || $2<$3) {
                printf("Values out of order!\n");
                YYERROR;
            }
        }
```

Interpreting the y.output file

Consider the following yacc specification:

```
%token DING DONG DELL
%%
rhyme  :  sound place
        ;
sound  :  DING DONG
        ;
place  :  DELL
        ;
```

The *y.output* file corresponding to the preceding grammar (with some statistics stripped off the end) is:

```
state 0
    $accept :  _rhyme $end
    DING shift 3
    . error
    rhyme goto 1
    sound goto 2

state 1
    $accept :  rhyme_$end
    $end accept
    . error

state 2
    rhyme :  sound_place
    DELL shift 5
    . error
    place goto 4

state 3
    sound :  DING_DONG
    DONG shift 6
    . error

state 4
    rhyme :  sound place_ (1)
    . reduce 1

state 5
    place :  DELL_ (3)
    . reduce 3

state 6
    sound :  DING DONG_ (2)
    . reduce 2
```

The actions for each state are specified, with a description of the parsing rules processed in each state. The underscore (`_`) character in a rule separates the symbols that have been seen from those that are expected to follow. The following input can be used to track the operations of the parser:

DING DONG DELL

The input is processed in the following steps:

1. Initially, the current state is state 0. The parser needs to refer to the input to decide between the actions available in state 0, so the first token, DING, is read and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, so state 3 is pushed onto the stack and the look-ahead token is cleared.
2. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, so state 6 is pushed onto the stack and the look-ahead is cleared. The stack now contains 0, 3, and 6.
3. In state 6, without even consulting the look-ahead, the parser reduces by rule 2:

sound : DING DONG

In the reduction, two states, 6 and 3, are popped off the stack, uncovering state 0.

4. Now *sound*, the left side of rule 2, has just been recognized. Consulting the description of state 0, we see that there is a *goto* on *sound*:

sound goto 2

This causes state 2 to be pushed onto the stack and become the current state.

5. In state 2, the next token, DELL, is read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it. The look-ahead token is cleared.
6. In state 5, the only action is to reduce by rule 3:

place : DELL

This rule has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered.

7. There is a **goto** on *place* in state 2; this causes the state to become 4. Now, the stack contains 0, 2, and 4.
8. In state 4, the only action is to reduce by rule 1:

rhyme : sound place

There are two symbols on the right, so the top two states are popped off, uncovering state 0.

9. In state 0, there is a **goto** on *rhyme*, causing the parser to enter state 1.
10. In state 1, the end-marker, indicated by **\$end**, is obtained when the input is read. The accept action in state 1 (when the end-marker is seen) successfully ends the parse.

Ambiguity and conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the following grammar rule is a natural way of expressing the fact that one way of forming an arithmetic expression is to join two other expressions with a minus sign.

```
expr  :  expr '-' expr
```

Unfortunately, this grammar rule does not completely specify how all complex inputs should be structured.

The **yacc** program detects such ambiguities when it is attempting to build the parser.

Consider the problem that confronts the parser when it is given the input:

```
expr - expr - expr
```

After the parser reads the second *expr*, the visible input is:

```
expr - expr
```

This matches the right side of the preceding grammar rule. The parser can reduce the input by the rule. After applying this rule, the input is reduced to *expr* (the left side of the rule). The parser then reads the final part of the input:

```
- expr
```

A similar reduction then takes place. This interpretation of the rule is called the “left associative” interpretation.

Alternatively, the parser could defer the immediate application of the rule when it has seen:

```
expr - expr
```

Instead, the parser can continue reading until it sees the whole input:

```
expr - expr - expr
```

It can then apply the rule to the rightmost three symbols, reducing them to *expr*, leaving:

```
expr - expr
```

Now the rule can be reduced once more. This interpretation is called “right associative”.

Depending on which interpretation is used, the parser can do either a shift or a reduction after reading:

```
expr - expr
```

This is called a **shift-reduce** conflict. The parser can also have a choice of two legal reductions, called a **reduce-reduce** conflict. There are no **shift-shift** conflicts.

Disambiguating rules

yacc uses “disambiguating rules” so that it can produce a parser when **shift-reduce** or **reduce-reduce** conflicts occur. The two default disambiguating rules are:

- where there is a **shift-reduce** conflict, perform the shift
- where there is a **reduce-reduce** conflict, reduce by the grammar rule that appears earlier in the **yacc** specification

Rule 1 means reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user further control over the behavior of the parser when there are **reduce-reduce** conflicts; these should be avoided whenever possible.

In general, when it is possible to apply disambiguating rules to a grammar to produce a correct parser, it is also possible to write equivalent grammar rules without the ambiguities.

yacc always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by the two disambiguating rules. The conflict messages of **yacc** are best understood by examining the *y.output* file. Here is an example that describes the conflicts that arise from an **if-else** construct, a common source of ambiguity in programming language grammars:

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE
```

```
state 23  '
    stat : IF ( cond ) stat_          (18)
    stat : IF ( cond ) stat_ELSE stat
    ELSE      shift 45
    .         reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The state description gives the grammar rules active in the state and the parser actions. In state 23, the parser has seen input corresponding to:

```
IF ( cond ) stat
```

Two grammar rules are active at this time, and so the parser can do one of two things. If the input symbol is **ELSE**, it will shift into state 45. State 45 will

have the following line as part of its description, indicating that ELSE has been shifted to arrive at this state.

```
stat : IF ( cond ) stat ELSE_stat
```

In state 23, the default action (designated by a dot, `(.)`) will be performed if the input symbol is something other than ELSE. In that case, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

The fact that the action that occurs when ELSE is read appears first in *y.output* indicates that the **shift** is the favored action.

Users who encounter unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

The disambiguating rules given above are not sufficient in resolving the conflicts that arise in the parsing of arithmetic expressions. These situations require that the parser be given some information about “precedence” and “associativity”. Most of the commonly-used constructions for arithmetic expressions can be described naturally by the notion of precedence levels for operators, together with left and right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than those constructed from unambiguous grammars.

Grammar rules for binary operators are typically written in the form:

```
expr : expr OP expr
```

Rules for unary operators typically look like:

```
expr : UNARYOP expr
```

These create a very ambiguous grammar with many parsing conflicts. To avoid ambiguity, the user can specify the precedence of all the operators and the associativities of the binary operators. This information is sufficient in allowing *yacc* to resolve the parsing conflicts and construct a parser that implements the precedences and associativities.

In an expression where there is a choice of two operators to evaluate, “precedence” determines which of the two is evaluated. For example, the following expression could be evaluated to 23 or 35, depending on whether ‘+’ or ‘*’ is evaluated first:

```
3+4*5
```

The operator evaluated first is said to have higher precedence.

“Associativity” determines which side of an expression involving a particular operator should be evaluated first. The following example could be evaluated to 2 or 4:

6-3-1

The result depends on whether the logical grouping is:

(6-3)-1

or:

6-(3-1)

If an operator is “left associative”, then the expression to the left of the operator is evaluated first, as in the first case above. If it is “right associative”, then the right side is evaluated first, as in the second case above.

The precedences and associativities are attached to tokens in the declarations section of the **yacc** specification. This is done with a series of lines starting with one of the **yacc** keywords **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All of the tokens on the same line have the same precedence level and associativity; the lines occur in order of increasing precedence. The following lines describe the precedence and associativity of the four arithmetic operators:

```
%left '+' '-'
%left '*' '/'
```

Plus and minus are left-associative and have lower precedence than star and slash, which are also left-associative. The keyword **%right** is used to indicate right-associative operators, and the keyword **%nonassoc** is used to describe operators like **.LT.** in FORTRAN that may not associate with themselves. The following statement is illegal in FORTRAN, and therefore the **.LT.** operator should be described with the keyword **%nonassoc**:

```
A .LT. B .LT. C
```

Here is a **yacc** specification for expressions involving operators:

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NAME
    ;
```

Now consider the following input tokens:

```
a = b = c*d - e - f*g
```

A parser that follows the specification just given structures the input as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

Assigning a precedence to grammar rules

In general, unary operators must be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is the unary and binary minus, (−). Unary minus often has the same strength as multiplication, or even higher, while binary minus will have a lower strength than multiplication.

Since the representations of the two operators are the same, the keywords **%left** and **%right** cannot be used to set the precedences for both. Rather, the precedence for the unary minus is associated with the “grammar rule” in which it appears. The keyword **%prec** is used to change the precedence level associated with a particular grammar rule. Here is a grammar that involves unary and binary minuses:

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
    ;
```

The keyword **%prec** appears immediately after the body of the grammar rule, before an action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. In this example, it is used to give unary minus the same precedence as the multiplication operator, (*).

If a token appears after one of the keywords **%left**, **%right**, and **%nonassoc**, it does not need to be declared by **%token** as well, although it is not incorrect to do so.

Precedences and associativities used by **yacc** to resolve parsing conflicts give rise to the following disambiguating rules:

- The precedence and associativity associated with a grammar rule is that of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
- When there is a **reduce-reduce** conflict, or when there is a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used and the conflicts are reported.
- If there is a **shift-reduce** conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (**shift** or **reduce**) associated with the higher precedence. If precedences are equal, then associativity is used. Left-associativity implies **reduce**, right-associativity implies **shift**, and nonassociativity implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts that **yacc** reports. This means that mistakes in the precedence specification may disguise errors in the input grammar. It is a good idea to be sparing with precedence and use them in a cookbook fashion until you are experienced in using it. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

Error handling

The input to the parser will sometimes not conform to the specifications. If no provision has been made to deal with errors, the processing halts when an error is detected. Instead of stopping all processing when an error is found, it is often more useful to continue scanning the input to find other syntax errors. Some of the problems associated with error handling are semantic ones. When an error is found, for example, it may be necessary to reclaim parse-tree storage, delete or alter symbol-table entries, or set switches to avoid generating any further output.

Error handling mechanisms can be provided as part of the input specifications. This permits the reentry of data after bad data has been seen, or the continuation of the input process after skipping over the bad data. This leads to the problem of where parser should resume parsing after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

Error token

The token name **error** is provided by **yacc** to allow the user some control over this process. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. If an error occurs, the parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token, and performs the action encountered. The look-ahead token is then reset to the token that caused the error.

In order to prevent a cascade of error messages, after detecting an error, the parser remains in an error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is ignored.

As an example, the following rule means that on a syntax error the parser will pop its state stack until the symbol *stat* is valid. It will then act as if *stat* has been seen and perform the code associated with the token **error**:

```
stat      :  error
           {
             ... action code ...
           }
| ...
.
.
```

Error rules such as this are very general but difficult to control. Rules like the following are somewhat easier:

```
stat      :  error '.'
```

Here, when there is an error, the parser attempts to skip over the statement to the next period. Tokens following the error and preceding the next period cannot be shifted and are discarded. When the period is seen, this rule will be reduced and any action associated with it performed.

Interactive error recovery

Another form of **error** rule can be used in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example illustrates one way to do this:

```
input      :      error  '\n'
            {
                (void) printf( "Reenter last line: " );
            }
            input
            {
                $$ = $4;
            }
            ;
```

There is one potential difficulty with this approach: the parser must correctly process three input tokens before it considers that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is unlikely to be acceptable, and so there is a mechanism that can force the parser to believe that error recovery has been accomplished. The following statement in an action resets the parser to its normal mode:

```
yyerrok ;
```

The last example can be rewritten as follows:

```
input:  error  '\n'
        {
            yyerrok;
            (void) printf( "Reenter last line: " );
        }
        input
        {
            $$ = $4;
        }
        ;
```

As previously mentioned, the next token seen after the **error** symbol is the same token that was seen when **error** was discovered. Sometimes this is inappropriate: for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. This can be done with the statement:

```
yyclearin ;
```

For example, suppose that the action after **error** is to call some sophisticated resynchronization routine that attempts to advance the input to the beginning of the next valid statement. After this routine is called, the next token

returned by the lexical analyzer is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset. An action similar to the following can perform this:

```

stat      :      error
          {
            resynch();
            yyerrok  ;
            yyclearin;
          }
          ;

```

Hints for preparing specifications

This part contains miscellaneous hints on preparing clear, efficient, and easily changeable specifications.

Input style

It is difficult to provide rules with substantial actions and still have a readable specification file. Here are a few style hints:

- Use all uppercase letters for token names and all lowercase letters for non-terminal names. This helps in debugging.
- Preface token names with an unusual sequence of characters, such as **t_**, to ensure that there are no conflicts with C reserved words.
- Put grammar rules and actions on separate lines to make the specification easier to read and edit.
- Put all rules with the same left side together. The left side should appear only once and each rule after the first one should begin with a vertical bar. Put the semicolon on a separate line after the last rule. This allows new rules to be added easily.
- Indent the body of a rule by one tab stop and the body of an action by two tab stops.
- Put complicated actions into subroutines.

Left recursion

The algorithm used by the **yacc** parser encourages left-recursive grammar rules. Rules of the following form match this algorithm:

```
name : name rest_of_rule ;
```

Rules like this arise frequently during the writing of specifications for sequences and lists. For example:

```
list : item
     | list ',' item
     ;
```

and:

```
seq : item
     | seq item
     ;
```

The first rule in each group will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules such as the following, the parser is somewhat bigger, and the items are seen and reduced from right to left:

```
seq : item
     | item seq
     ;
```

A more serious problem is that an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion whenever possible.

It is worth considering whether a sequence with zero elements has any meaning; if it has, consider writing the sequence specification using an empty rule:

```
seq : /* empty */
     | seq item
     ;
```

Once again, the first rule is always reduced exactly once before the first item is read, and then the second rule is reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts may arise if the parser is asked to decide between empty sequences that can satisfy more than one rule, when not enough input has been seen to know which one is appropriate.

Lexical tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might normally want to delete blanks, but not within quoted strings; names might be entered into a symbol table in declarations, but not in expressions. One way of handling these situations is to create a global flag that is set by actions in the parser but which can be accessed by the lexical analyzer. Then the lexical analyzer can make decisions on the basis of this variable's value. The following specification specifies a program that consists of zero or more declarations followed by zero or more statements:

```
%{
    int dflag;
}%
... other declarations ...
%%

prog  :  decls  stats
      ;

decls :  /* empty */
      {
          dflag = 1;
      }
      |  decls  declaration
      ;

stats :  /* empty */
      {
          dflag = 0;
      }
      |  stats  statement
      ;

... other rules ...
```

The flag **dflag** is set to 0 when reading statements and 1 when reading declarations. The first token in the first statement must be seen by the parser before it can tell that the declarations section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

Reserved words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass the lexical analyzer information telling it one instance of **if** is a keyword and another instance is a variable. The user can attempt to implement this using the mechanism described in the last subsection, but this is not trivial. It is better that the keywords be reserved, that is, forbidden for use as variable names.

Examples of using yacc

This section contains two extended examples. They display many of the features of **yacc** discussed in this chapter.

Simple example

This example gives the complete **yacc** applications for a small desk calculator; the calculator has 26 registers, labeled **a** through **z**, and accepts arithmetic expressions made up of the operators:

```
+, -, *, /, % (mod operator), & (bitwise and),
| (bitwise or), and assignments.
```

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator shows how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

```

%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */

%%          /* beginning of rules section */

list       : /* empty */
            | list stat '\n'
            | list error '\n'
            {
                yyerrok;
            }
            ;

stat       : expr
            {
                (void) printf( "%d\n", $1 );
            }
            | LETTER '=' expr
            {
                regs[$1] = $3;
            }
            ;

```

```

expr      : '(' expr ')'
          {
              $$ = $2;
          }
          | expr '+' expr
          {
              $$ = $1 + $3;
          }
          | expr '-' expr
          {
              $$ = $1 - $3;
          }
          | expr '*' expr
          {
              $$ = $1 * $3;
          }
          | expr '/' expr
          {
              $$ = $1 / $3;
          }
          | expr '%' expr
          {
              $$ = $1 % $3;
          }
          | expr '&' expr
          {
              $$ = $1 & $3;
          }
          | expr '|' expr
          {
              $$ = $1 | $3;
          }
          | '-' expr %prec UMINUS
          {
              $$ = -$2;
          }
          | LETTER
          {
              $$ = regs[$1];
          }
          | number
          ;

```

```

number      :  DIGIT
             {
                 $$ = $1; base = ($1==0) ? 8 : 10;
             }
             |  number DIGIT
             {
                 $$ = base * $1 + $2;
             }
             ;

%%          /* beginning of subroutines section */

int yylex( ) /* lexical analysis routine */
{
    /* return LETTER for lowercase letter, */
    /* yylval = 0 through 25 */
    /* returns DIGIT for digit, yylval = 0 through 9 */
    /* all other characters are returned immediately */
    int c;
        /*skip blanks*/
    while ((c = getchar()) == ' ')
        ;

        /* c is now nonblank */

    if (islower(c))
    {
        yylval = c - 'a';
        return (LETTER);
    }
    if (isdigit(c))
    {
        yylval = c - '0';
        return (DIGIT);
    }

    return (c);
}

```

Advanced example

This section gives an example of a grammar using some of the advanced features. The desk calculator example is modified to provide a desk calculator that does floating-point interval arithmetic. The calculator understands floating-point constants, the arithmetic operations $+$, $-$, $*$, $/$, and unary minus $(-)$, and; registers labeled **a** through **z**. It also understands intervals written in the following format, where X is less than or equal to Y :

(X, Y)

There are 26 interval-valued variables, **A** through **Z**, that may also be used.

The usage is similar to that in the previous example: assignments return no value and print nothing, while expressions print the floating or interval value.

Intervals are represented by a structure consisting of the left and right end-point values stored as doubles. This structure is given a type name, `INTERVAL`. The `yacc` value stack can also contain floating point scalars, and integers used to index into the arrays holding the variable values.

`YYERROR` is used to handle error conditions. The errors dealt with in this way are division by an interval containing 0, and an interval presented in the wrong order. The error recovery mechanism of `yacc` is used to discard the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is processed by `yacc`: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines:

```
2.5 + (3.5 - 4.)
```

and:

```
2.5 + (3.5, 4)
```

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is avoided by having two rules for each binary interval valued operator—one when the left operand is a scalar, and one rule when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this, there are still many cases where the conversion may be applied or not, leading to the above conflicts. These are resolved by listing the rules that yield scalars first in the specification file: in this way, the conflict will be resolved in favor of keeping scalar valued expressions scalar valued, until they are forced to become intervals.

This way of handling multiple types is very instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more so. Thus, while this example is instructive, it is better practice in a more normal programming language environment to maintain the type information at the lexical level, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating-point constants. The C library routine **atof()** is used to do the actual conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser and then error recovery.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval
{
    double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

}%

%start line

%union
{
    int ival;
    double dval;
    INTERVAL vval;
}

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST      /* floating point constant */

%type <dval> dexp        /* expression */

%type <vval> vexp        /* interval expression */

    /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS /* precedence for unary minus */
```

```

%%                               /* beginning of rules section */
lines : /* empty */
      | lines line
      ;
line  : dexp '\n'
      {
          (void) printf("%15.8f\n", $1);
      }
      | vexp '\n'
      {
          (void) printf("(%15.8f, %15.8f)\n", $1.lo, $1.hi);
      }
      | DREG '=' dexp '\n'
      {
          dreg[$1] = $3;
      }
      | VREG '=' vexp '\n'
      {
          vreg[$1] = $3;
      }
      | error '\n'
      {
          yyerrok;
      }
      ;

dexp  : CONST
      | DREG
      {
          $$ = dreg[$1];
      }
      | dexp '+' dexp
      {
          $$ = $1 + $3;
      }
      | dexp '-' dexp
      {
          $$ = $1 - $3;
      }
      | dexp '*' dexp
      {
          $$ = $1 * $3;
      }
      | dexp '/' dexp
      {
          $$ = $1 / $3;
      }

```

```

: '-' dexp %prec UMINUS
{
    $$ = -$2;
}
| '(' dexp ')'
{
    $$ = $2;
}
;

vexp : dexp
{
    $$ .hi = $$ .lo = $1;
}
| '(' dexp ',' dexp ')'
{
    $$ .lo = $2;
    $$ .hi = $4;
    if( $$ .lo > $$ .hi )
    {
        (void) printf("interval out of order \n");
        YYERROR;
    }
}
| VREG
{
    $$ = vreg[$1];
}
| vexp '+' vexp
{
    $$ .hi = $1 .hi + $3 .hi;
    $$ .lo = $1 .lo + $3 .lo;
}
| dexp '+' vexp
{
    $$ .hi = $1 + $3 .hi;
    $$ .lo = $1 + $3 .lo;
}
| vexp '-' vexp
{
    $$ .hi = $1 .hi - $3 .lo;
    $$ .lo = $1 .lo - $3 .hi;
}
| dexp '-' vexp
{
    $$ .hi = $1 - $3 .lo;
    $$ .lo = $1 - $3 .hi;
}

```



```

| vexp '*' vexp
{
    $$ = vmul( $1.lo,$1.hi,$3 );
}
| dexp '*' vexp
{
    $$ = vmul( $1, $1, $3 );
}
| vexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1, $1, $3 );
}
| dexp '/' vexp
{
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1.lo, $1.hi, $3 );
}
| '-' vexp %prec UMINUS
{
    $$hi = -$2.lo; $$lo = -$2.hi;
}
| '(' vexp ')'
{
    $$ = $2;
}

;

%%
/* beginning of subroutines section */
# define BSZ 50 /* buffer size for floating point number */

/* lexical analysis */
int yylex( )
{
    register int c;

    /* skip over blanks */
    while ((c = getchar()) == ' ')
        ;
    if (isupper(c))
    {
        yylval.ival = c - 'A';
        return (VREG);
    }
    if (islower(c))
    {
        yylval.ival = c - 'a';
        return(DREG);
    }
}

```

```

        /* gobble up digits. points, exponents */

if (isdigit(c) || c == '.')
{
    char buf[BSZ+1], *cp = buf;
    int dot = 0, exp = 0;

    for(; (cp - buf) < BSZ ; ++cp, c = getchar())
    {
        *cp = c;
        if (isdigit(c))
            continue;
        if (c == '.')
        {
            if (dot++ || exp)
                return ('.'); /* will cause syntax error */
            continue;
        }
        if (c == 'e')
        {
            if (exp++)
                return ('e'); /* will cause syntax error */
            continue;
        }
        /* end of number */
        break;
    }

    *cp = ' ';
    if (cp - buf >= BSZ)
        (void) printf("constant too long - truncated\n");
    else
        ungetc(c, stdin); /* push back last char read */
    yylval.dval = atof(buf);
    return (CONST);
}
return (c);
}

```

```

INTERVAL
hilo(a, b, c, d)
    double a, b, c, d;
{
    /* returns the smallest interval containing a, b, c, and d */

    /* used by *,/ routine */
    INTERVAL v;

    if (a > b)
    {
        v.hi = a;
        v.lo = b;
    }
    else
    {
        v.hi = b;
        v.lo = a;
    }
    if (c > d)
    {
        if (c > v.hi)
            v.hi = c;
        if (d < v.lo)
            v.lo = d;
    }
    else
    {
        if (d > v.hi)
            v.hi = d;
        if (c < v.lo)
            v.lo = c;
    }
    return (v);
}

INTERVAL
vmul(a, b, v)
    double a, b;
    INTERVAL v;
{
    return (hilo(a * v.hi, a * v.lo, b * v.hi, b * v.lo));
}

```

```

dcheck(v)
    INTERVAL v;
{
    if (v.hi >= 0. && v.lo <= 0.)
    {
        (void) printf("divisor interval contains 0.\n");
        return (1);
    }
    return (0);
}
INTERVAL
vdiv(a, b, v)
    double a, b;
    INTERVAL v;
{
    return (hilo(a / v.hi, a / v.lo, b / v.hi, b / v.lo));
}

```


Chapter 14

Shared libraries

Efficient use of disk storage space, memory, and computing power is becoming increasingly important. A shared library can offer savings in all three areas. For example, if constructed properly, a shared library can make *a.out* files (executable object files) smaller on disk storage and processes (*a.out* files that are executing) smaller in memory.

The section “What is a shared library?” (this page), describes what a shared library is and how to use one to build *a.out* files. It also offers suggestions about when and when not to use a shared library and how to determine whether an *a.out* uses a shared library.

The section “Building a shared library” (page 454), describes how to build a shared library. Specifically, this section describes how to use **mkshlib**(CP) and how to write C code for shared libraries. Also described is how to use the tool **chkshlib**(CP), which checks the compatibility of versions of shared libraries.

What is a shared library?

A shared library is a file containing object code that several *a.out* files may use simultaneously while executing. When a program is link edited with a shared library, the library code that defines the program’s external references is not copied into the program’s object file. Instead, a special section called **.lib** that identifies the library code is created in the object file. When the UNIX system executes the resulting *a.out* file, it uses the information in this section to bring the required shared library code into the address space of the process.

The implementation behind these concepts is a shared library with two pieces. The first, called the host shared library, is an archive that the link editor searches to resolve user references and to create the **.lib** section in *a.out* files.

The structure and operation of this archive is the same as any archive without shared library members. It must be present on the system when the *a.out* files are link edited.

The second part of a shared library is the target shared library. This is the file that the UNIX system uses when running *a.out* files built with the host shared library. It contains the actual code for the routines in the library. It must be present on the the system where the *a.out* files will be run.

A shared library offers several benefits by not copying code into *a.out* files. It can:

- Save disk storage space. Shared library code is not copied into all the *a.out* files that use that code. *a.out* files are smaller and use less disk space.
- Save memory. By sharing library code at run time the dynamic memory needs of the processes are reduced.
- Make executable files using library code easier to maintain.

At run time shared library code is brought into the processes' address space. Therefore, updating a shared library effectively updates all executable files that use the library. If an error in shared library code is fixed, all processes automatically use the corrected code.

Non-shared libraries cannot offer this benefit: changes to archive libraries do not affect executable files, because code from the libraries is copied to the files during link editing, not during execution.

Shared libraries examples

The C and networking libraries are available as shared libraries.

Shared Library	Host Library Command Line Option	Target Library Pathname
C Library	<code>-lc_s</code>	<code>/shlib/libc_s</code>
Networking Library	<code>-lnsl_s</code>	<code>/shlib/libnsl_s</code>

The `_s` suffix is used to indicate that the library is shared and statically linked.

Building an a.out file

The standard (non-shared) C library is still available with releases of the C Programming Language Utilities; this library is searched by default during the compilation or link editing of C programs.

To build *a.out* files, a shared library's name must be directly referenced by using the `-l` option with the C compiler. You direct the link editor to search a shared library the same way you direct a search of any library on the UNIX system:

```
cc file.c -o file ... -l library_file ...
```

To direct a search of the networking library, for example, you use the following command line.

```
cc file.c -o file ... -l nsl_s ...
```

And to link all the files in your current directory together with the shared C library you'd use the following command line:

```
cc *.c -lc_s
```

Including the `-lc_s` argument after all other `-l` arguments on a command line, indicates that the shared C library will be treated like the relocatable C library, which is searched by default after all other libraries specified on a command line.

A shared library might be built with references to other shared libraries. That is, the first shared library might contain references to symbols that are resolved in a second shared library.

For example, if the shared library *libX_s.a* references symbols in the shared C library, the command line would be as follows:

```
cc *.c -lX_s -lc_s
```

Notice that the shared library containing the references to symbols must be listed on the command line before the shared library needed to resolve those references. For more information on inter-library dependencies, see "Referencing symbols in a shared library from another shared library".

Deciding whether to use a shared library

The decision to use a shared library, should be based on whether it saves space in disk storage and memory. A well-designed shared library almost always saves space. As a general rule, use a shared library when it is available.

To determine what savings are gained from using a shared library, try building the same application with both a non-shared and a shared library, assuming both kinds are available. Then compare the two versions of the application for size and performance. For example:

```
$ cat hello.c
main()
{
    printf("Hello\n");
}
$ cc -o unshared hello.c
$ cc -o shared hello.c -lc_s
$ size unshared shared
unshared: 8680 + 1388 + 2248 = 12316
shared: 300 + 680 + 2248 = 3228
```

If the application calls only a few library members, it is possible that using a shared library could take more disk storage or memory. For a more detailed discussion see “Space considerations” (this page).

Space considerations

The following section outlines the benefits and drawbacks of using a shared library. It covers:

- how shared libraries save space that non-shared libraries cannot
- how shared libraries might increase space usage

Saving space

To better understand how a shared library saves space, compare the space used by a shared library and a non-shared library.

A host shared library resembles a non-shared library in three ways. First, both are archive files. Second, the object code in the library typically defines commonly used text symbols and data symbols. The symbols defined inside, and made visible outside, the library are external symbols. Note that the library may also have imported symbols: symbols that it uses but does not define.

Third, the link editor searches the library for these symbols when linking a program to resolve its external references. By resolving the references, the link editor produces an executable version of the program, the *a.out* file.

NOTE The link editor on the UNIX system is a static linking tool; static linking requires all symbolic references in a program be resolved before the program may be executed. The link editor uses static linking with both a non-shared library and a shared library.

Although these similarities exist, a shared library differs significantly from a non-shared library. The major differences are related to how the libraries are handled to resolve symbolic references.

To produce an *a.out* file using a non-shared library, the link editor copies the library code that defines a program's unresolved external reference from the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an *a.out* file using a shared library, the link editor copies from the shared library into the program's object file only a small amount of code for initialization of imported symbols. For more details on imported symbols, see "Importing symbols" (page 464). For the bulk of the library code, it creates a special section called **.lib** in the file that identifies the library code needed at run time and resolves the external references to shared library symbols with their correct values. When the UNIX system executes the resulting *a.out* file, it uses the information in the **.lib** section to bring the required shared library code into the address space of the process.

Figure 14-1, “*a.out* files created using a non-shared library and a shared library” (this page) depicts the *a.out* files produced using a non-shared version and a shared version of the standard C library to compile the following program:

```
main()
{
    ...
    printf( "How do you like this manual?\n" );
    ...
    result = strcmp( "I do.", answer );
    ...
}
```

Notice that the shared version is smaller. Figure 14-2, “Processes using an archive and a shared library” (page 449) depicts the process images in memory of these two files when they are executed.

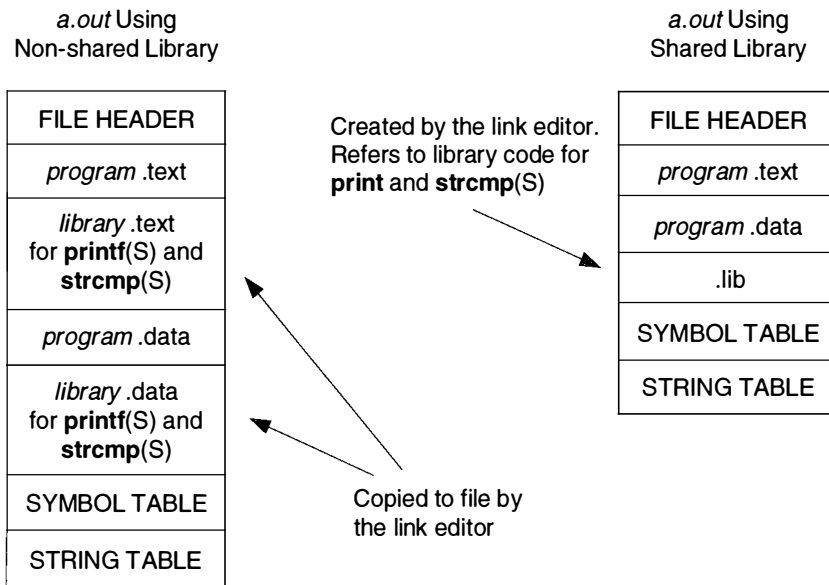


Figure 14-1 *a.out* files created using a non-shared library and a shared library

Now consider what happens when several *a.out* files need the same code from a library. When using a non-shared library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory

when the *a.out* files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the *a.out* files, as indicated in Figure 14-2, “Processes using an archive and a shared library” (this page). This separation enables all processes using the same shared library to reference a single copy of the code.

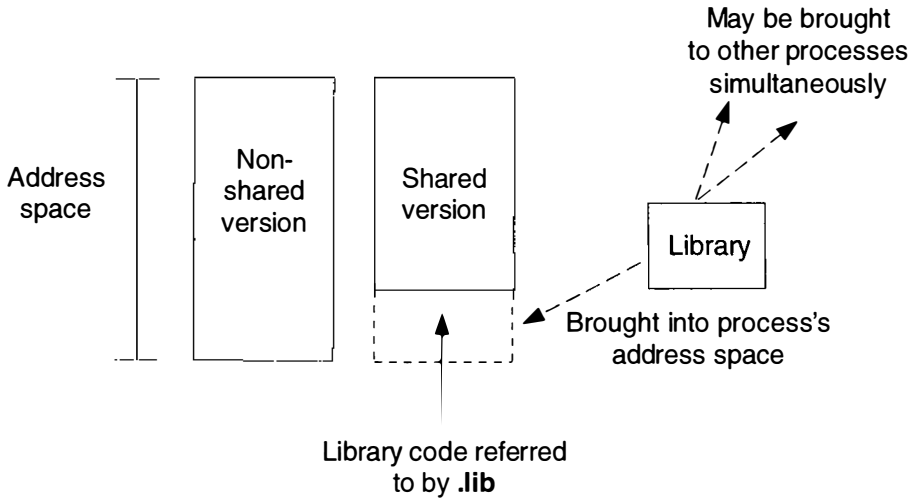


Figure 14-2 Processes using an archive and a shared library

Increase space usage in memory

A target library might add space to a process. A shared library's target file may have both text and data regions connected to a process. The text region is shared by all processes that use the library, the data region is not. Every process that uses the library gets its own private copy of the entire library data region. This adds to the process's memory requirements. As a result, if an application uses only a small part of a shared library's text and data, executing the application might require more memory with a shared library than without one. For example, using the shared C library to access only `strcmp(S)` saves disk storage and memory. However, the memory cost for sharing all the shared C library's private data region outweighs the savings. The non-shared version of the library would be more appropriate.

Coding an application

Application source code in C or assembly language is compatible with both non-shared and shared libraries. As a result, there should be no coding changes required in applications that currently use a shared library. When coding a new application for use with a shared library, observe standard coding conventions.

Observe the following two points when using either a non-shared or a shared library:

- Don't define application symbols with the same names as those in a library.
Although exceptions exist, avoid redefining standard library routines, such as **printf(S)** and **strcmp(S)**. Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.
- Don't use undocumented archive routines.

Use only the functions and data mentioned on the manual pages describing the routines in Section (S) of the *Programmer's Reference*.

Identifying a.out files that use shared libraries

Use the **dump(CP)** command to look at the section headers for the file:

```
dump -hv a.out
```

If the file has a **.lib** section, a shared library is needed. If the *a.out* does not have a **.lib** section, it does not use shared libraries.

To display the shared libraries used by *a.out*, use the **-L** option as shown in the following example:

```
dump -L a.out
```

Debugging a.out files that use shared libraries

dbxtra reads the shared libraries' symbol tables and performs as documented (in the *Programmer's Reference*) using the available debugging information. The branch table is hidden so that functions in shared libraries can be referenced by their names.

Shared library data is not dumped to core files, however. If an error is encountered resulting in a core dump and does not appear in the application's code, debugging might be easier if the application is rebuilt with the non-shared version of the library used.

Implementing shared libraries

The following section describes host and target shared libraries and the branch table.

The host library and target library

Every shared library has two parts: the host library used for linking that resides on the host machine and the target library used for execution that resides on the target machine. The host machine is the machine on which an *a.out* file is built; the target machine is the machine on which the file is run. The host and target may be the same machine, but it is not a requirement.

The host library is just like a non-shared library. Each of its members (typically a complete object file) defines text and data symbols in its symbol table. The link editor searches this file when a shared library is used during the compilation of a program.

The search is for definitions of symbols referenced in the program but not defined there. However, the link editor does not copy the library code defining the symbols into the program's object file. Instead, it uses the library members to locate the definitions and then places symbols in the file that tell where the library code is. The result is the special section in the *a.out* file shown in Figure 14-2, "Processes using an archive and a shared library" (page 449) as **.lib**.

The target library, used for execution, resembles an *a.out* file. During execution, if a process needs a shared library, this file is read. The **.lib** section in the *a.out* file tells which shared libraries are needed. When the UNIX system executes the *a.out* file, it uses this section to bring the appropriate library code into the address space of the process. In this way, before the process starts to run, all required library code has been made available.

Shared libraries enable the sharing of **.text** sections in the target library, which is where text symbols are defined. Although processes that use the shared library have their own virtual address spaces, they share a single physical copy of the library's text among them.

The target library cannot share its **.data** sections. Each process using data from the library has its own private data region that mirrors the **.data** section of the target library. Processes that share text do not share data and stack area in order that they do not interfere with one another.

As suggested above, the target library is a lot like an *a.out* file, which can also share its text, but not its data. Processes must have execute permission for a target library in order to execute an *a.out* file that uses the library.

The branch table

When the link editor resolves an external reference in a program, it gets the address of the referenced symbol from the host library. A static linking loader like `ld` binds symbols to addresses during link editing. In this way, the *a.out* file for the program has an address for each referenced symbol.

If non-shared library code is updated and the address of a symbols changes, the *a.out* file will still run, since that file already has a copy of the code defining the symbol. However, that type of change can adversely affect an *a.out* file built with a shared library. This file has only a single symbol telling where the required library code is. Therefore, if the *a.out* file runs after a change, the operating system could bring in the wrong code.

To avoid this, two options are available. The first is to recompile the library after each update. The second is to implement the shared library with a branch table. A branch table associates text symbols with absolute addresses. Each address labels a jump instruction to the address of the code defining a symbol. Instead of being directly associated with the addresses of code, text symbols have addresses in the branch table.

Figure 14-3, “A branch table in a shared library” (this page) shows two *a.out* files executing a call to **printf(S)**. The process on the left was built using a non-shared library. It already has a copy of the library code defining the **printf** symbol. The process on the right was built using a shared library. This file references an absolute address (10) in the branch table of the shared library at run time; at this address, a jump instruction references the needed code.

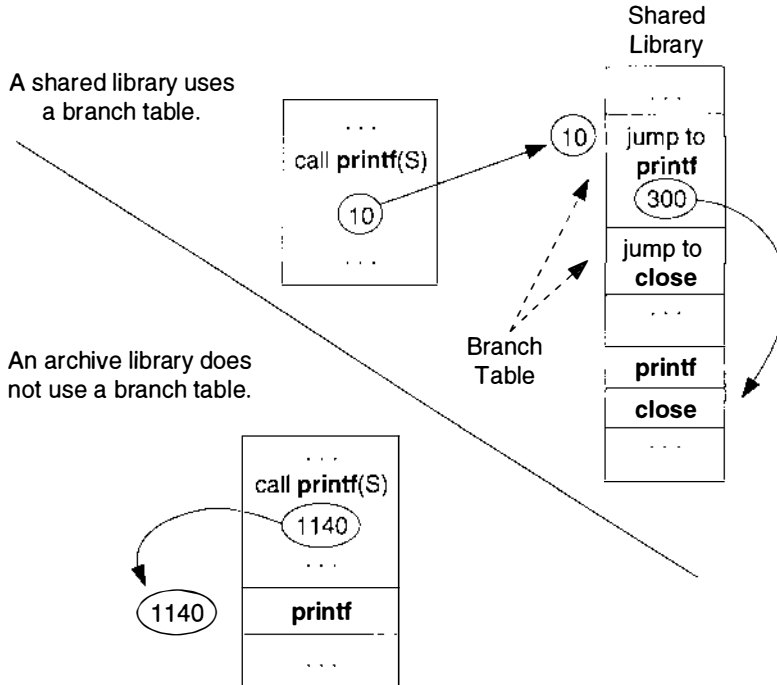


Figure 14-3 A branch table in a shared library

Data symbols do not have a mechanism to prevent a change of address between shared libraries. The tool **chkshlib(CP)** compares *a.out* files with a shared library to check compatibility and help you decide if the files need to be recompiled. See “Checking versions of shared libraries using **chkshlib(CP)**” (page 473).

Building a shared library

This section of the chapter explains how to build a shared library. It covers the major steps in the building process, the use of the UNIX system tool **mkshlib**(CP) that builds the host and target libraries, and some guidelines for writing shared library code. An example is provided to demonstrate the building process.

This section makes the following assumptions:

- you are an advanced C programmer
- you are familiar with the archive library building process.
- familiar with basic operating system concepts

The building process

To build a shared library the following six steps must be completed:

- choosing region addresses
- choosing the pathname for the shared library target file
- selecting the library contents
- rewriting existing library code to be included in the shared library
- writing the library specification file
- using the **mkshlib** tool to build the host and target libraries

Step 1: choosing region addresses

Choose region addresses for your shared library.

Shared library regions on the 386-based computer correspond to memory management unit (MMU) segment sizes, each of which is 4 MB. The following table gives a list of the segment assignments on the 386-based computer (as of the copyright date for this guide) and shows what virtual addresses are available for libraries.

Start Address	Contents	Target Pathname
0xA0000000 . . .	Reserved for AT&T Shared C Library AT&T Networking Library	/shlib/libc_s /shlib/libnsl_s
0xA3C00000 0xA4000000 0xA4400000 0xA4800000 0xA4C00000	Generic Database Library	Unassigned
0xA5000000 0xA5400000 0xA5800000 0xA5C00000	Generic Statistical Library	Unassigned
0xA6000000 0xA6400000 0xA6800000 0xA6C00000	Generic User Interface Library	Unassigned
0xA7000000 0xA7400000 0xA7800000 0xA7C00000	Generic Screen Handling Library	Unassigned
0xA8000000 0xA8400000 0xA8800000 0xA8C00000	Generic Graphics Library	Unassigned
0xA9000000 0xA9400000 0xA9800000 0xA9C00000	Generic Networking Library	Unassigned
0xAA000000 0xAB000000 . . . 0xAFC00000	Scan Code library Libprot library	/shlib/libsc_s /shlib/protlib_s
0xB0000000 . . . 0xBF000000	For private use	Unassigned

If a shared library is built within a reserved address region, there is a risk of conflicting with future products.

A number of segments are allocated for shared libraries that provide various services such as graphics, database access, and so on. These categories are intended to reduce the chance of address conflicts among commercially available libraries. Although two libraries of the same type may conflict, that does not matter. A single process should not usually need to use two shared libraries of the same type. If the need arises, a program can use one shared library and one non-shared library instead.

NOTE Any number of libraries can use the same virtual addresses, even on the same machine. Conflicts occur only within a single process, not among separate processes. Thus two shared libraries can have the same region addresses without causing problems, as long as a single *a.out* file does not need to use both libraries.

Several segments are reserved for private use. If you are building a large system with many *a.out* files and processes, shared libraries might improve its performance. As long as you do not intend to release the shared libraries as separate products, you should use the private region addresses. You can put your shared libraries into these segments and avoid conflicting with commercial shared libraries. You should also use these segments when you will own all the *a.out* files that access your shared library. Don't risk address conflicts.

NOTE If you plan to build a commercial shared library, you are strongly encouraged to provide a compatible, relocatable archive as well.

Step 2: choosing the target library pathname

After selecting the region addresses, choose the pathname for the target library. For example */shlib/libc_s* was chosen for the shared C library and */shlib/libnsl_s* for the networking library. To choose a pathname for your shared library, consult the established list of names for your computer or see your system administrator.

NOTE Shared libraries required to boot a UNIX system should be located in */shlib*; other application libraries normally reside in */usr/lib* or in private application directories.

Step 3: selecting library contents

The most important task in the building process is selecting the contents for the shared library. Some routines are prime candidates for sharing; others are not. For example, it is a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that are not used as much.

For general guidelines see “Choosing library members” (page 458). See also the guidelines in “Importing symbols” (page 464), “Referencing symbols in a shared library from another shared library” (page 469), and “Tuning the shared library code” (page 471).

Step 4: rewriting existing library code

If you choose to include some existing code from an archive library in a shared library, changing some of the code will make the shared code easier to maintain. See “Changing existing code for the shared library” (page 459).

Step 5: writing the library specification file

After you select and edit all the code for your shared library, you have to build the shared library specification file. The library specification file contains all the information that **mkshlib** needs to build both the host and target libraries. See “Writing the specification file” (page 482) for a sample specification file. Also, see “Using the specification file for compatibility” (page 462). The contents and format of the specification file are given by the directives list. For a description of each directive, see **mkshlib**(CP). References to these directives are made throughout this chapter. Prior review of **mkshlib**(CP) is recommended.

Step 6: using **mkshlib** to build the host and target

The **mkshlib**(CP) command builds both the host and target libraries. **mkshlib** invokes other tools such as the assembler, **as**(CP), and link editor, **ld**(CP). Tools are invoked through the use of **execvp** (see **exec**(S)), which searches directories in a user’s **\$PATH** environment variable. Also, prefixes to **mkshlib** are parsed in much the same manner as prefixes to the **cc**(CP) command and invoked tools are given the prefix, where appropriate. For example, **3mkshlib** invokes **3ld**.

Guidelines for writing shared library code

The main advantage of a shared library over an archive library is sharing and the space it saves. These guidelines stress ways to increase sharing while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility.

We recommend that you read these guidelines once to get a perspective of the concepts involved in building a shared library.

When building a shared library there are a few restrictions involving static linking that should be considered. For example:

- External symbols have fixed addresses. If an external symbol moves, you have to re-link all *a.out* files that use the library. This restriction applies both to text and data symbols.

Use of the **#hide** directive to limit externally visible symbols can help avoid problems in this area. (For more details, see “Use #hide and #export to limit externally visible symbols” (page 462).

- If the library’s text changes for one process at run time, it changes for all processes.
- If the library uses a symbol directly, that symbol’s run time value (address) must be known when the library is built.
- Imported symbols cannot be referenced directly. Their addresses are not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library’s data.

Choosing library members

Include large, frequently used routines

Large, frequently used routines are prime candidates for sharing. Placing them in a shared library saves code space for individual *a.out* files and saves memory, too, when several concurrent processes need the same code. **printf(S)** and related C library routines are good examples.

NOTE Since the **printf** family of routines is used frequently, **printf** and related routines in to the shared C library.

Exclude infrequently used routines

Putting infrequently used routines in a shared library can degrade performance, particularly on paging systems. Traditional *a.out* files contain all code they need at run time. The code in an *a.out* file is related to the process. Therefore, if a process calls a function, it may already be in memory because of its proximity to other text in the process. See also “Organize to improve locality” (page 472).

Exclude routines that use much static data

Routines that use much static data increase the size of processes. Every process that uses a shared library gets its own private copy of the library’s data, regardless of how much of the data is needed. Library data is static: it is not shared and cannot be loaded selectively, with the provision that unreferenced pages may be removed from the working set.

For example, `getgrent(S)` is not used by many standard UNIX system commands. Some versions of the module define over 1400 bytes of unshared, static data. It probably should not be included in a shared library.

Exclude routines that complicate maintenance

All external symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols do not have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of external data may affect symbol addresses and break compatibility.

Include routines the library itself needs

Make the library self-contained. For example, `printf(S)` requires much of the standard I/O library. A shared library containing `printf` should contain the rest of the standard I/O routines, too.

NOTE This guideline should not take priority over the others in this section. If you exclude some routine that the library itself needs based on a previous guideline, consider leaving the symbol out of the library and importing it.

Changing existing code for the shared library

All C code that works in a shared library will also work in a non-shared library. However, the reverse is not true because a shared library must explicitly handle imported symbols. The following guidelines are meant to help you produce shared library code that is still valid for non-shared libraries (although it may be slightly bigger and slower). The guidelines explain how to structure data for ease of maintenance, since most compatibility problems involve restructuring data.

Minimize global data

All external data symbols are visible to applications. This can make maintenance difficult. Try to reduce global data, as described below.

First, try to use automatic (stack) variables. Don't use permanent storage if automatic variables work. Using automatic variables saves static data space and reduces the number of symbols visible to application processes.

Second, see whether variables really must be external. Static symbols are not visible outside the library, so they may change addresses between library versions. Only external variables must remain constant. For further tips, see "Use `#hide` and `#export` to limit externally visible symbols" (page 462).

Third, allocate buffers at run time instead of defining them at compile time. This reduces the size of the library's data region for all processes (saving memory). It also allows the size of the buffer to change from one release to the next without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Define text and global data in separate source files

Separating text from global data to prevent data symbols from moving. If new external variables are needed, they can be added at the end of the old definitions to preserve the old symbols' addresses.

Libraries let the link editor extract individual members. This works fine for relocatable files, but shared libraries have a different set of restrictions. If external variables were scattered throughout the library modules resulting in external and static data being intermixed. Changing static data, like **hello** in the following example, moves subsequent data symbols, even the external symbols:

Before	Broken Successor
...	...
int head = 0;	int head = 0;
...	...
func()	func()
{	{
...	...
p = "hello";	p = "hello, world";
...	...
}	}
...	...
int tail = 0;	int tail = 0;
...	...

Assume the relative virtual address of **head** is 0 for both examples. The string literals will have the same address too, but they have different lengths. The old and new addresses of **tail** thus might be 12 and 20, respectively. If **tail** is supposed to be visible outside the library, the two versions will not be compatible.

NOTE The compilation system sometimes defines and uses static data invisibly to the user (for example, tables for switch statements).

Adding new external variables to a shared library may change the addresses of static symbols, but this does not affect compatibility. An *a.out* file has no way to reference static library symbols directly, so it cannot depend on their values.

NOTE There is a real danger of mixing in static data with exported data when building a shared library. This may not immediately cause problems but may cause incompatibilities in later versions of the shared library. If the external data modules are not first, a seemingly harmless change (such as a new string literal) can break existing *a.out* files. Even changing the code may not be necessary to cause a problem: the user might count on the compiler to place statics at a known default location. If the compiler is ever replaced, that default location may change and then existing *a.out* files will also break. By this time it may be too late to do anything about and the only alternative may be to distribute multiple versions of the same shared library.

To avoid these problems, group all exported data symbols and place them at lower addresses than static (hidden) data. The following are suggestions for locating exported data symbols at lower addresses than static data:

1. Put the exported data symbols in a file (or files) by themselves.
2. Do not put other external data in this file. Remember, if the **#hide** directive has been used, all other external data may become static.
3. Do not initialize exported data symbols with static data. For example, initializing an exported data symbol to an unnamed string literal is a bad idea since the exported data file will now contain static data intermixed with the exported data. If there is a need to initialize the exported data symbol then do so in a way that the initializations are themselves data symbols which can be defined in another data file.
4. Put all other external data in a separate data file. Shared library users get all library data at run time, regardless of the source file organization. Consequently, all external variables' definitions can be put in a single source file without a space penalty.
5. Place data and text object files in the **#objects** list as follows:
 - a. imported symbols definition files (remember symbols mentioned in **#init** and **#branch** directives are always external)
 - b. exported data files
 - c. all other data files
 - d. all other text files
6. Check the exported data object file by dumping the object file after it has been built. The size of all data should exactly match the size of the exported data symbols.
7. Add new exported data to the end of the exported data file.

Initialize global data

Initialize external variables, including the pointers for imported symbols. Although this uses more disk space in the target shared library, the expansion is limited to a single file. **mkshlib** will give a fatal error if it finds an uninitialized external symbol.

Using the specification file for compatibility

The way in which directives are used in the specification file can affect compatibility across versions of a shared library. This section gives some guidelines on how to use the directives **#branch**, **#hide**, and **#export** (see also **mkshlib(CP)**).

Preserve branch table order

Add new functions only at the end of the branch table. Try and maintain compatibility with previous versions after a specification file for the library is created. New functions can be added without breaking old *a.out* files as long as the previous assignments are not changed. This allows distribution of a new version of the library without having to re-link all of the *a.out* files that used the previous version.

Use **#hide** and **#export** to limit externally visible symbols

Variables (or functions) can be referenced from several object files for inclusion in the shared library. However, they are not intended to be available to users of the shared library. They must be external so that the link editor can properly resolve all references to symbols and create the target shared library, but should be hidden from the user's view to prevent their use. Unintended use can result in compatibility problems if the symbols move or are removed between versions of the shared library.

The **#hide** and **#export** directives can resolve this dilemma. The **#hide** directive causes **mkshlib**, after resolving all references within the shared library, to alter the symbol tables of the shared library so that all specified external symbols are made static and inaccessible from user code. You can specify the symbols to be so treated either individually or through the use of regular expressions.

The **#export** directive allows you to specify those symbols in the range of an accompanying **#hide** directive regular expression that should remain external. For example, in the shared C library all data symbols are hidden by default. Symbols required outside of the library are then explicitly exported:

```
#hide linker
#export linker
errno
optarg
opterr
optind
optopt
```

The advantage to this approach is that future changes to the library won't introduce new external symbols (possibly causing name collisions), unless the new symbols are explicitly exported. The symbols to be exported are chosen by looking at a list of all the current external symbols in the shared C library and finding out what each symbol is used for. The symbols that are global but were only used in the shared C library are not exported; these symbols will be hidden from applications code. All other symbols are explicitly exported.

NOTE It is a fatal error to try to explicitly name the same symbol in a **#hide** and an **#export** directive.

The **#export** directive is useful when building a complicated shared library where many symbols are to be made static. In these cases, it is more efficient to use regular expressions to make all external variables static and individually list those symbols you need to be external.

NOTE Symbols mentioned in the **#branch** and **#init** directives are services of the shared library, must be external symbols, and cannot be made static through the use of these directives.

Importing symbols

Shared library code cannot directly use symbols defined outside a library, but an escape hatch exists. You can define pointers in the data area and arrange for those pointers to be initialized to the addresses of imported symbols. Library code then accesses imported symbols indirectly, delaying symbol binding until run time. Libraries can import both text and data symbols. Moreover, imported symbols can come from the user's code, another library, or even the library itself. In Figure 14-4, "Imported symbols in a shared library" (this page), the symbols `_libc_ptr1` and `_libc_ptr2` are imported from user's code and the symbol `_libc_malloc` from the library itself.

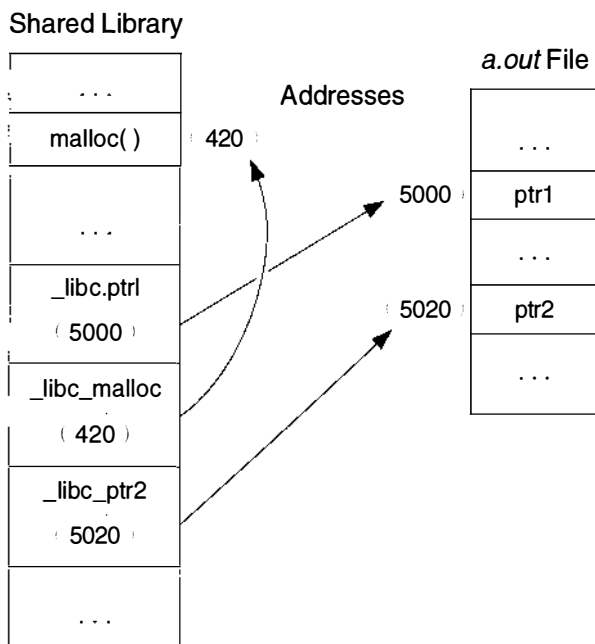


Figure 14-4 Imported symbols in a shared library

The following guidelines describe when and how to use imported symbols.

Imported symbols that the library does not define

Non-shared libraries typically contain relocatable files, which allow undefined references. Although the host shared library is an archive, too, that archive is constructed to mirror the target library, which more closely resembles an `a.out` file. Neither target shared libraries nor `a.out` files can have unresolved references to symbols.

Shared libraries must import any symbols they use but do not define. Some shared libraries will derive from existing non-shared libraries. For the reasons stated above, it may not be appropriate to include all the non-shared archive's modules in the target shared library. Remember though that if you exclude a symbol from the target shared library that is referenced from the target shared library, you will have to import the excluded symbol.

Imported symbols that users must be able to redefine

Optionally, shared libraries can import their own symbols. Two standard libraries, **libc** and **libmalloc**, provide a **malloc** family. Even though most UNIX system commands use the **malloc** from the C library, they can choose either library or define their own.

Three possible strategies exist for building the shared C library. First, exclude the **malloc(S)** family. But other library members might need it, and so it will have to be an imported symbol. This will work, but it means less savings.

Second, include the **malloc** family; do not import it. This provides more savings for typical commands. However, other library routines call **malloc** directly, and those calls can not be overridden. If an application tries to redefine **malloc**, the library calls will not have to use the alternate version. Furthermore, the link editor will find multiple definitions of **malloc** while building the application. To resolve this the library developer will have to change source code to remove the custom **malloc**, or refrain from using the shared library.

Finally, the most flexible, is to include **malloc** in a shared library, treating it as an imported symbol. Even though **malloc** is in the library, nothing else there refers to it directly; all references are through an imported symbol pointer. If the application does not redefine **malloc**, both application and library calls are routed to the library version. All calls are mapped to the alternate, if present.

You might want to permit redefinition of all library symbols in some libraries. You can do this by importing all symbols the library defines, in addition to those it uses but does not define. Although this adds a little space and time overhead to the library, the technique allows a shared library to be one hundred percent compatible with an existing non-shared library at link time and run time. This is the strategy used for the installed version of the Shared C library.

Mechanics of importing symbols

For example, assume a shared library wants to import the symbol **malloc**. The original non-shared code and the shared library code appear below.

Non-Shared Library

```
extern char *malloc();

funce()
{
    ...
    p = malloc(n);
    ...
}
```

Shared Library Code

```
/* See pointers.c on next page */

extern char *(*_libc_malloc)();

funce()
{
    ...
    p = (*_libc_malloc)(n);
    ...
}
```

Making this transformation is straightforward, but two sets of source code would be necessary to support both a non-shared and a shared library. Some simple macro definitions can hide the transformations and allow source code compatibility. A header file defines the macros, and a different version of this header file would exist for each type of library. The **-I** flag to **cc(CP)**, documented in the *Programmer's Reference*, would direct the C preprocessor to look in the appropriate directory to find the desired file.

Non-shared import.h

```
/* empty */
```

Shared import.h

```
/*
 * Macros for importing
 * symbols. One #define
 * per symbol.
 */

...
#define malloc    (*_libc_malloc)
...
extern char *malloc();
...
```

These header files allow one source both to serve the original archive source and to serve a shared library, too, because they supply the indirections for imported symbols. The declaration of **malloc** in *import.h* actually declares the pointer **_libc_malloc**.

```

Common Source

#include "import.h"

extern char *malloc();

funce()
{
    ...
    p = malloc(n);
    ...
}

```

Alternatively, one can hide the `#include` with `#ifdef`:

```

Common Source

#ifdef SHLIB
#    include "import.h"
#endif

extern char *malloc();

funce()
{
    ...
    p = malloc(n);
    ...
}

```

NOTE When building the shared library the code can be conditionally turned on by defining `shlib` via the `-D` flag to `cc(CP)`.

Of course the transformation is not complete. You must define the pointer `_libc_malloc`.

```

File pointers.c
...
char *(*_libc_malloc)() = 0;
...

```

NOTE `_libc_malloc` is initialized to zero, because it is an external data symbol.

Special initialization code sets the pointers. Shared library code should not use the pointer before it contains the correct value. In the example the address of `malloc` must be assigned to `_libc_malloc`. Tools that build the shared library generate the initialization code according to the library specification file.

Pointer initialization fragments

A host shared library archive member can define one or many imported symbol pointers. Regardless of the number, every imported symbol pointer should have initialization code.

This code goes into the *a.out* file and does two things. First, it creates an unresolved reference to make sure the symbol being imported gets resolved. Second, initialization fragments set the imported symbol pointers to their values before the process reaches **main**. If the imported symbol pointer can be used at run time, the imported symbol will be present, and the imported symbol pointer will be set properly.

NOTE Initialization fragments reside in the host, not the target, shared library. The link editor copies initialization code into *a.out* files to set imported pointers to their correct values.

Library specification files describe how to initialize the imported symbol pointers. For example, the following specification line would set **_libc_malloc** to the address of **malloc**:

```
...
#init pmalloc.o
_libc_malloc    malloc
...
```

When **mkshlib** builds the host library, it modifies the file **pmalloc.o**, adding relocatable code to perform the following assignment statement:

```
_libc_malloc = &malloc ;
```

When the link editor extracts **pmalloc.o** from the host library, the relocatable code goes into the *a.out* file. As the link editor builds the final *a.out* file, it resolves the unresolved references and collects all initialization fragments. When the *a.out* file is executed, the run time startup routines execute the initialization fragments to set the library pointers.

Selectively loading imported symbols

You can reduce unnecessary loading by writing C source files that define imported symbol pointers singly or in related groups. If an imported symbol must be individually selectable, put its pointer in its own source file (and archive member). This will give the link editor a finer granularity to use when it resolves the reference to the symbol.

For example, a single source file might define all pointers to imported symbols:

old pointers.c

```
...
int (*_libc_ptr1)() = 0;
char (*_libc_malloc)() = 0;
int (*_libc_ptr2)() = 0;
...
```

Allowing the loader to resolve only those references that are needed requires multiple source files and archive members. Each of the new files defines a single pointer:

File	Contents
<i>ptr1.c</i>	int (*_libc_ptr1)() = 0;
-	
<i>pmalloc.c</i>	char (*_libc_malloc)() = 0;
<i>ptr2.c</i>	int (*_libc_ptr2)() = 0;

Using the three files ensures that the link editor will only look for definitions for imported symbols and load in the corresponding initialization code in cases where the symbols are actually used.

NOTE If one version of a library does not resolve any pointers in an archive member, it is then not possible to use any of those pointers in a future version of the shared library without breaking backward compatibility.

Referencing symbols in a shared library from another shared library

In general, import all symbols defined outside the shared library whenever possible.

However, this is not always possible, as for example when floating-point operations are performed in a shared library to be built. When such operations are encountered in any C code, the standard C compiler generates calls to functions to perform the actual operations. These functions are defined in the C library and are normally resolved in a manner invisible to the user when an *a.out* is created, since the `cc` command automatically causes the relocatable (non-shared) version of the C library to be searched. These floating-point routine references must be resolved at the time the shared library is being built. But, the symbols cannot be imported, because their names and usage are invisible.

The **#objects noload** directive **mkshlib(CP)** is provided to allow symbol references such as these to be resolved at the time the shared library is built, provided that the symbols are defined in another shared library. If there are unresolved references to symbols after the object files listed with the **#objects** directive have been link edited, the host shared libraries specified with the

#objects noload directive are searched for absolute definitions of the symbols. The normal use of the directive is to search the shared version of the C library to resolve references to floating-point routines.

For this use, the syntax in the specification file would be

```
#objects noload
-lc_s
```

This would cause **mkshlib** to search for the host shared library **libc_s.a** in the default library locations and to use it to resolve references to any symbols left unresolved in the shared library being built. The **-L** option can be used to cause **mkshlib** to look for the specified library in other than the default locations.

Using or building a shared library

When building a shared library using **#objects noload**, you must make sure that for each symbol with an unresolved reference there is a version of the symbol with an absolute definition in the searched host shared libraries, before any relocatable version of that symbol. **mkshlib** will give a fatal error if this is not the case, because relocatable definitions do not have absolute addresses and therefore do not allow complete resolution of the target shared library.

When using a shared library built with references to symbols resolved from another shared library, both libraries must be specified on the **cc** command line. The dependent library must be specified on the command line before the libraries on which it depends. (See the section “Building a shared library” (page 454) for more details.) If you provide a shared library which references symbols in another shared library, you should make sure that your documentation clearly states that users must specify both libraries when building *a.out* files.

Finally, it is possible to use **#objects noload** to resolve references to any symbols not defined in a shared library, as long as they are defined in some other shared library. Therefore, you are strongly encourage you to import as many symbols as possible and to use **#objects noload** only when absolutely necessary. Probably you will only need to use this feature to resolve references to floating-point routines generated by the C compiler.

However, importing symbols has several important benefits over resolving references through **#objects noload**. First, importing symbols is more flexible in that it allows you to define your own version of library routines. You can define your own versions with archive versions of a library. Preserving this ability with the shared versions helps maintain compatibility.

Importing symbols also helps prevent unexpected name space collisions. The link editor will complain about multiple definitions of a symbol, references to which are resolved through the **#objects noload** mechanism, if a user of the shared library also has an external definition of the symbol.

Finally, **#objects noload** has the drawback that both the library you build and all the libraries on which it depends must be available on all the systems. Anyone who wishes to create *a.out* files using your shared library will need to use the host shared libraries. Also, the targets of all the libraries must be available on all systems on which the *a.out* files are to be run.

Providing compatibility with non-shared libraries

Having compatible libraries makes it easy to substitute one for the other. In almost all cases, this can be done without makefile or source file changes. For example, the shared C library is built using the existing non-shared library as the base. The host library archive file is compatible with the relocatable non-shared C library. However, the shared library target file does not include all routines from the archive, because including them all would have hurt performance.

These goals are reached as follows. The host library is built in two steps. First, the available shared library tools are used to create the host library to exactly match the target. The resulting archive file may not be compatible with the archive C library at this point. Second, add to the host library the set of relocatable objects residing in the archive C library that are missing from the host library. Although this set is not in the shared library target, its inclusion in the host library makes the relocatable and shared C libraries compatible.

Tuning the shared library code

Some suggestions for how to organize shared library code to improve performance are presented here. They apply to paging systems.

The non-shared C library contains several diverse groups of functions. Many processes use different combinations of these groups, making the paging behavior of any shared C library difficult to predict. A shared library should offer greater benefits for more homogeneous collections of code. For example, a database library probably could be organized to reduce system paging substantially, if its static and dynamic calling dependencies were more predictable.

Profile the code

First, profile the code that might go into the shared library (see `prof(CP)`).

Choose library contents

Based on profiling information, make some decisions about what to include in the shared library. *a.out* file size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained. See “Choosing library members” (page 458). for more information.

Organize to improve locality

Try to improve locality of reference by grouping dynamically related functions. If every call of `funcA` generates calls to `funcB` and `funcC`, try to put them in the same page. `cflow(CP)` (documented in the *Programmer's Reference*) generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

Align for paging

Arrange the shared library target's object files so that frequently used functions do not unnecessarily cross page boundaries. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. Use name lists and disassemblies of the shared library target file, to determine where the page boundaries fall.

After grouping related functions, break them into page-sized chunks. Although some object files and functions are larger than a single page, most of them are smaller. Use the infrequently called functions as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault decreases.

After determining the branch table, arrange the library's object files without breaking compatibility. Put frequently used, unrelated functions together because they probably will be called randomly enough to keep the pages in memory. System calls go into another page as a group, and so on. The following example shows how to change the order of the C library's object files:

Before	After
#objects	#objects
...	...
printf.o	strcmp.o
fopen.o	malloc.o
malloc.o	printf.o
strcmp.o	fopen.o
...	...

Avoid hardware thrashing

Improved performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment.

Checking for compatibility

The following guidelines explain how to check for upwardly compatible shared libraries. However, upward compatibility may not always be an issue. Consider the case in which a shared library is one piece of a larger system and is not delivered as a separate product. In this restricted case, identify all *a.out* files that use a particular library. As long as all the *a.out* files are rebuilt every time the library changes, the *a.out* files will run successfully, even though versions of the library are not compatible.

Checking versions of shared libraries using **chkshlib**(CP)

a.out files will not execute properly if newer versions of a library are not compatible with the previous ones.

If you use shared libraries, you might need to find out if different versions of a shared library are compatible, or if executable files could have been built with a particular host shared library or can run with a particular target shared library. For example, you might have a new version of a target shared library, and you need to know if all the executable files that ran with the older version will run with the new one. You might need to find out if a particular target shared library can reference symbols in another shared library. **chkshlib**(CP) has been provided to allow you to do these and other comparisons.

chkshlib takes names of target shared libraries, host shared libraries, and executable files as input, and checks to see if those files satisfy the compatibility criteria. **chkshlib** checks to see if every library symbol in the first file that needs to be matched exists in the second file and has the same address.

The following table shows what types of files and how many of them **chkshlib** accepts as input. The rows listed down represent the first input given, and the columns listed across represent secondary input. For example, if the first input file you give **chkshlib** is a target shared library, you must give another input file that is a target or host shared library.

	Nothing	Executable	Target	Host
Executable	OK	No	OK*	OK*
Target	No	No	OK	OK
Host	OK	No	OK	OK

* The executable file must be one that was built using a host shared library. A useful way to confirm this is to use **dump -L** to find out which target file(s) gets loaded when the program is run. See **dump(CP)**.

** You can also have *executable target1...targetn* and *executable host1...hostn*.

An example of a **chkshlib** command line is shown below:

```
chkshlib /shlib/libc_s /lib/libc_s.a
```

In this example, */shlib/libc_s* is a target shared library and */lib/libc_s.a* is a host shared library. **chkshlib** will check to see if executable files built with */shlib/libc_s* would be able to run with */lib/libc_s.a*.

Depending on the input it receives, **chkshlib** checks to find out if the following is true:

- an executable file will run with the given target shared library
- an executable file could have been built using the given host shared library
- an executable file produced with a given host shared library will run with a given target shared library
- an executable file that ran with an old version of a target shared library will run with a new version
- a new host shared library can replace the old host shared library; that is, executable files built with the new host shared library will run with the old target shared library
- a target shared library can reference symbols in another target shared library

To determine if files are compatible, **chkshlib** has to determine which library symbols in the first file need to be matched in the second file.

- For target shared libraries, the symbols of concern are all external, defined symbols with non-zero values, except for branch labels (branch labels always start with **.bt**), and the special symbols **etext**, **edata**, and **end**.
- For host shared libraries, the symbols of concern are all external, absolute symbols with a non-zero value.
- For executable files, the symbols of concern are all external, absolute symbols with a non-zero value, except for the special symbols **etext**, **edata**, and **end**.

For two files to be compatible, the target pathnames must be identical in both files (unless the **chkshlib -i** option has been specified).

The following table displays the output you will receive when you use **chkshlib** to check different combinations of files for compatibility. In this table **file1** represents the name of the first file given, and **file2,3,...** represents the names of any more files given as input.

Input	Output
file1 is executable file2,3,... (if any) are targets	file1 can [may not] execute using file2 file1 can [may not] execute using file3
file1 is executable file2,3 are hosts	file1 may [may not] have been produced using file2 file1 may [may not] have been produced using file3
file1 is host file2 (if any) is target	file1 can [may not] produce executables which will run with file2
file1 is target file2 is host	file2 can [may not] produce executables which will run with file1
both files are targets <i>or</i> both files are hosts	file1 can [may not] replace file2 file2 can [may not] replace file1
both files are targets and -n option is specified*	file1 can [may not] include file2

- * The **-n** option tells **chkshlib** that the two files are target shared libraries, the first of which can reference (include) symbols in the other. See "Referencing symbols in a shared library from another shared library" for details.

For more information on **chkshlib**, see **chkshlib(CP)**.

NOTE Symbols that have been hidden via the **#hide** directive cannot be referenced directly. **chkshlib** will ignore them in its check for compatibility.

Dealing with incompatible libraries

There are two methods of dealing with incompatible libraries. First, you can rebuild all the *a.out* files that use your library. If feasible, this is probably the best method. Unfortunately, you might not be able to find those *a.out* files, let alone force their owners to rebuild them with your new library.

The second method is to give a different target pathname to the new version of the library. The host and target pathnames are independent; you do not have to change the host library pathname. New *a.out* files will use your new target library, but old *a.out* files will continue to access the old library.

As the library developer, it is your responsibility to check for compatibility and, probably, to provide a new target library pathname for a new version of a library that is incompatible with older versions. If you fail to resolve compatibility problems, *a.out* files that use your library will not work properly.

NOTE You should try to avoid multiple library versions. If too many copies of the same shared library exist, they might actually use more disk space and more memory than the equivalent relocatable version would have.

An example

This section contains the process by which a small specialized shared library is created and built. We refer to the guidelines given earlier in this chapter.

The original source

The name of the library to be built is *libmaux* (for math auxiliary library). The interface consists of three functions, an external variable, and a header file.

The three functions are:

- logd** floating-point logarithm to a given base; defined in the file *log.c*
- polyd** evaluate a polynomial; defined in the file *poly.c*
- maux_stat** return usage counts for the other two routines in a structure; defined in *stats.c*,

The external variable is:

- mauxerr** set to non-zero if there is an error in the processing of any of the functions in the library and set to zero if there is no error (unlike *errno* in the C library),

The header file is:

- maux.h* declares the return types of the function and the structure returned by **maux_stat**.

The source files before any modifications for inclusion in a shared library follow.

Example 14-1 File *log.c*

```

/* log.c */
#include "maux.h"
#include <math.h>

/*
 * Return the log of "x" relative to the base "a".
 *
 *      logd(base, x) := log(x) / log(base);
 * where "log" is "log to the base E".
 */

double logd(base, x)
double base, x;
{
    extern int stats_logd;
    extern int total_calls;

    double logbase;
    double logx;

    total_calls++;
    stats_logd++;

    logbase = log((double)base);
    logx = log((double)x);
    if(logbase == -HUGE || logx == -HUGE) {
       iauxerr = 1;
        return(0);
    }
    else
       iauxerr = 0;
    return(logx/logbase);
}

```


Example 14-2 File *poly.c*

```

/* poly.c */

#include "maux.h"
#include <math.h>

/* Evaluate the polynomial
 *      f(x) := a[0] * (x ^ n) + a[1] * (x ^ (n-1)) + ... + a[n];
 * Note that there are N+1 coefficients!
 * This uses Horner's Method, which is:
 *      f(x) := (((((a[0]*x) + a[1])*x) + a[2]) + ...) + a[n];
 * It's equivalent, but uses many fewer operations
 *      and is more precise. */

double polyd(a, n, x)
double a[];
int n;
double x;
{
    extern int stats_polyd;
    extern int total_calls;
    double result;
    int i;

    total_calls++;
    stats_polyd++;
    if (n < 0) {
        mauxerr = 1;
        return(0);
    }
    result = a[0];
    for (i = 1; i <= n; i++)
    {
        result *= (double)x;
        result += (double)a[i];
    }
    mauxerr = 0;
    return(result);
}

```

Example 14-3 File *stats.c*

```

/* stats.c */
#include "maux.h"

int total_calls = 0;
int stats_logd = 0;
int stats_polyd = 0;

int mauxerr;

/* Return structure with usage stats for functions in library
 * or 0 if space cannot be allocated for the structure */
struct mstats *
maux_stat()
{
    extern char * malloc();
    struct mstats * st;

    if((st = (struct mstats *)
        malloc(sizeof(struct mstats))) == 0)
        return(0);
    st->st_polyd = stats_polyd;
    st->st_logd = stats_logd;
    st->st_total = total_calls;
    return(st);
}

```

Example 14-4 Header File *maux.h*

```

/* maux.h */

struct mstats {
    int st_polyd;
    int st_logd;
    int st_total;
};

extern double polyd();
extern double logd();
extern struct mstats * maux_stat();

extern int mauxerr;

```

Choosing region addresses and the target pathname

First, we choose the region addresses for the library's **.text** and **.data** sections from the segments reserved for private use on the 80386 Computer. Note that the region addresses must be on a segment boundary (4 MB):

```
.text    0xB0600000
.data    0xB0a00000
```

Also we choose the pathname for our target library:

```
/my/directory/libmaux_s
```

NOTE The choice of region addresses can be important. See the comments in “Step 1: choosing region addresses” (page 454). A table of existing and suggested addresses is also given there.

Selecting library contents

This example is for illustration purposes, and so we will include everything in the shared library. In a real case, it is unlikely that you would make a shared library with these three small routines, unless you had many programmers using them frequently.

Rewriting existing code

According to the guidelines given earlier in the chapter, you need to first minimize the global data. We realize that **total_calls**, **stats_logd**, and **stats_polyd** do not need to be visible outside the library, but are needed in multiple files within the library. Hence, we will use the **#hide** directive in our specification file to make these variables static after the shared library is built.

We need to define text and global data in separate source files. The only piece of global data we have left is **mauxerr**, which we will remove from **stats.c** and put in a new file **maux_defs.c**. We will also have to initialize it to zero, since shared libraries cannot have any uninitialized variables.

Next, we notice that there are some references to symbols that we do not define in our shared library (i.e. **log** and **malloc**). We can import these symbols. To do so, we create a new header file, **import.h**, which will be included in each of **log.c**, **poly.c**, and **stats.c**. The header file defines C preprocessor macros for these symbols to make transparent the use of indirection in the actual C source files. Use the **_libmaux_** prefixes on the pointers to the symbols because those pointers are made external, and the use of the library name as a prefix helps prevent name conflicts.

```

/* New header file import.h */
#define malloc      (*_libmaux_malloc)
#define log         (*_libmaux_log)

extern char * malloc();
extern double log();

```

Now, you need to define the imported symbol pointers somewhere. You have already created a file for global data **maux_defs.c**, so add the definitions to that.

```

/* Data file maux_defs.c */

int mauxerr = 0;
double (*_libmaux_log)() = 0;
char * (*_libmaux_malloc)() = 0;

```

Finally, you observe that there are floating-point operations in the code, and you remember that the routines for these cannot be imported. (If you tried to write the specification file and build the shared library without taking this into account, **mkshlib** would give you errors about unresolved references.) This means you will have to use the **#objects noload** directive in our specification file to search the C host shared library to resolve the references.

Writing the specification file

This is the specification file for **libmaux**:

Example 14-5 libmaux.sl

```

1  ##
2  ## libmaux.sl - libmaux specification lfile
3  #address .text 0xB0680000
4  #address .data 0xB06a0000
5  #target /my/directory/libmaux_s
6  #branch
7      polyd          1
8      logd           2
9     iaux_stat       3
10 #objects
11    iaux_defs.o
12     poly.o
13     log.o
14     stats.o
15 #objects noload
16     -lc_s
17 #hide linker *
18 #export linker
19    iauxerr
20 #initiaux_defs.o
21     _libmaux_malloc malloc
22     _libmaux_log    log

```

Briefly, this is what the specification file does. Lines 1 and 2 are comment lines. Lines 3 and 4 give the virtual addresses for the shared library text and data regions, respectively. Line 5 gives the pathname of the shared library on the target machine. The target shared library must be installed there for *a.out* files that use it to work correctly. Line 6 contains the **#branch** directive. Line 7 through 9 specify the branch table. They assign the functions **polyd()**, **logd()**, and **iaux_stat()** to branch table entries 1, 2, and 3. Only external text symbols, such as C functions, should be placed in the branch table.

Line 10 contains the **#objects** directive. Lines 11 through 14 give the list of object files that will be used to construct the host and target shared libraries. When building the host shared library archive, each file listed here will reside in its own archive member. When building the target library, the order of object files will be preserved. The data files must be first. Otherwise, an addition of static data to **poly.o**, for example, would move external data symbols and break compatibility.

Line 15 contains the **#objects noload** directive, and line 16 gives information about where to resolve the references to the floating-point routines.

Lines 17 through 19 contain the **#hide linker** and **#export linker** directives, which tell what external symbols are to be left external after the shared library is built. Together, these **#hide** and **#export** directives say that only **mauxerr** will remain external. The symbols in the branch table and those specified in the **#init** directive will remain external by definition.

Line 20 contains the **#init** directive. Lines 21 and 22 give imported symbol information for the object file **maux_defs.o**. You can imagine assignments of the symbol values on the right to the symbols on the left. Thus **_libmaux** will hold a pointer to **malloc**, and so on.

Building the shared library

Now, you have to compile the **.o** files as you would for any other library:

```
cc -c maux_defs.c poly.c log.c stats.c
```

Next, you need to invoke **mkshlib** to build our host and target libraries:

```
mkshlib -s libmaux.sl -t libmaux_s -h libmaux_s.a
```

Presuming all of the source files have been compiled appropriately, the **mkshlib** command line shown above will create both the host library, **libmaux_s.a**, and the target library, **libmaux_s**. Before any **a.out** files built with **libmaux_s.a** can be executed, the target shared library **libmaux_s** must be moved to **/my/directory/libmaux_s** as specified in the specification file.

Using the shared library

To use the shared library with a file, **x.c**, which contains a reference to one or more of the routines in **libmaux**, issue the following command line:

```
cc x.c libmaux_s.a -lm -lc_s
```

This command line causes the following:

- Imported symbol pointer reference to **log** is resolved from **libm**.
- Imported symbol pointer reference to **malloc** is resolved with the shared version from **libc_s**.

The most important thing to note from the command line, however, is that you have to specify the C host shared library (in this case with the **-lc_s**) on the command line, since **libmaux** was built with direct references to the floating-point routines in that library.

Appendix A

ANSI implementation-defined behavior

The American National Standards Institute (ANSI) has adopted a standard that specifies the form and establishes the interpretation of programs written in the programming language C. This standard is detailed in *ANSI Document Number X3J11/90-013*. The International Standards Organization (ISO) has adopted the functionally equivalent standard which is detailed in the (ISO) *Programming Languages — C*, document number ISO/IEC 9899:1990 (E). Both standards provide a list of characteristics and extensions to be implementation-defined. The characteristics and extensions that must be documented is provided in an appendix to the ANSI document, subsection “F.3 Implementation-Defined Behaviour”.

According to both the ANSI standard, and the ISO standard, each conforming implementation of the C language must be accompanied by a document that defines all implementation-defined characteristics and extensions.

This appendix is organized into the following sections, corresponding to the sections in the appendix to the ANSI document:

- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating Point
- Arrays and Pointers
- Registers

- Structure, Unions, Enumerations, and Bit-fields
- Qualifiers
- Declarators
- Statements
- Preprocessing Directives
- Library Functions
- Locale-specific Behaviour

Translation

This section describes the implementation-defined characteristics of source code translation. It corresponds to section “F.3.1 Translation” of the ANSI document.

Identifying diagnostics

Diagnostic messages are generated on standard error. Diagnostics from the compiler are of the form:

"filename", line number: level:message

where *level* is “error” or “warning”.

Diagnostics from other passes, (such as the `cc(CP)` command itself, or `as(CP)` or `ld(CP)`) may have a different form.

Environment

This section describes the implementation-defined characteristics of the execution environment. It corresponds to section “F.3.2 Environment” of the ANSI document.

Arguments to main()

The function `main` can have zero or two arguments, and is declared as follows:

```
int main( void)
int main( int argc, char *argv[] )
```

`argc` will always be greater than or equal to 1.

`argv[0]` contains the pathname by which the program invoked was executed.

`argv[1]` through `argv[argc - 1]` contain the exact strings as supplied to `exec`. No transformations on the arguments are performed by `exec` although the shell may transform the arguments.

Interactive devices

The input and output dynamics of an interactive device are specified by the standard. An interactive device is defined by the UNIX system, typically the console and terminals.

Identifiers

This section describes the implementation-defined characteristics of identifiers. It corresponds to section “F.3.3 Identifiers” in the ANSI document.

Significant characters without external linkage

The compiler recognizes at least 1024 significant initial characters in an identifier without external linkage. For debugging purposes, only the first 128 characters are recognized.

Significant characters with external linkage

The compiler recognizes at least 1024 significant initial characters in an identifier with external linkage. For debugging purposes, only the first 128 characters are recognized.

Significance of character case

Case is significant in an identifier with external linkage.

Characters

This section describes the implementation-defined characteristics of characters. It corresponds to section “F.3.4 Characters” in the ANSI document.

Source and execution character sets

The source character set is the set of legal characters that can appear in source files. The execution character set is the set of legal characters interpreted in the execution environment. For the compiler, the source and execution character sets are the same, the standard 8-bit character set.

Bits per character

The number of bits in a character in the execution character set is represented by the manifest constant `CHAR_BIT`, which is defined in *limits.h*. It is defined as 8 bits.

Mapping character sets

The source code character set maps directly to the execution character set. The source character set, a proper subset of the ASCII character set, commences with ASCII 32 and ends with ASCII 126. This includes all of the printable graphic characters of the ASCII character set.

Constants with unrepresented characters and escape sequences

Certain non-graphic characters may be represented in the source character set by escape sequences which begin with a backslash followed by a lowercase letter. These escape sequences map onto specific characters in the ASCII set as shown in the following table:

Table A-1 Nongraphic character escape sequences

Escape sequence	ASCII value	Character Name
<code>\'</code>	39	single quote
<code>\"</code>	34	double quote
<code>\?</code>	63	question mark
<code>\\</code>	92	backslash
<code>\a</code>	7	bell
<code>\b</code>	8	backspace
<code>\f</code>	12	formfeed
<code>\n</code>	10	new-line
<code>\r</code>	13	carriage return
<code>\t</code>	9	horizontal tab
<code>\v</code>	11	vertical tab

For escape sequences other than those listed in Table A-1, the backslash is stripped and the characters in the sequence are treated like ordinary characters. All other characters map directly from the source character set to the external character set.

Constants with multiple or wide characters

When a character constant contains more than one character, the individual values of the characters are concatenated together to form an **int**. Since **int** is equal to 4 bytes and each byte can hold a character, a character constant may be up to four characters long. Excess characters are dropped from the right side. For example if:

```
int i = 'abcd';
```

then the value of **i** in hex is 61626364, the individual values for "a", "b", "c", and "d" concatenated together. Again, if

```
int ii = 'abcdef';
```

then the value of **ii** in hex is 61626364, the individual values for "a", "b", "c", and "d" concatenated together. Note that the excess characters on the right, "e" and "f", are dropped.

Locale used for multi-byte conversion

The compiler uses the "C" locale to convert multibyte characters into corresponding wide characters.

Range of char values

A "plain" **char** has the same range of values as a **signed char**. The **-J** command line option changes the range of values to be the same as an **unsigned char**.

Integers

This section describes the implementation-defined characteristics of integers. It corresponds to section “F.3.5 Integers” in the ANSI document.

Integer range and representation

Integers are represented as follows:

```
char == signed char
      unsigned char

short == signed short
       unsigned short

int == signed int == long == signed long
unsigned int == unsigned long
```

NOTE The section “C Implementation Limits description”, later in this appendix, presents the range of values each integer type can take on.

Demotion of integers

When an integer is converted to a shorter signed integer, if the value cannot be represented, the excess bytes are truncated. For example if:

```
char c = (char) 0xabcdef;
```

then the value of `c` is `0xef`.

When an unsigned integer is converted to a signed integer of equal length, if the value cannot be represented, the conversion is the same, but the value is treated like a signed number.

Signed bitwise operations

In bitwise operations on signed integers, each of the operands of a bitwise operation is subject to the “usual arithmetic conversions”. Consequently, narrow integers are subject to “integral promotions”. For example, a **short** is promoted into an **int**. If the value of the **short** is negative, the sign is extended in the **int**. After the promotion, bitwise operations on signed **ints** are applied as if they were applied on unsigned **ints**.

Sign of division remainder

When integers are divided and the division is inexact, the sign of the remainder is the same as the sign of the dividend.

Right shift of negative-valued signed integer

When a signed integral type with a negative value is right shifted, the sign is extended.

Floating point

This section describes the implementation-defined characteristics of floating point numbers. It corresponds to section “F.3.6 Floating Point” in the ANSI document.

Floating-point range and representation

The representations and sets of values of the various types of floating-point numbers in the compiler are:

<code>float</code>	4 byte IEEE format
<code>double</code>	8 byte IEEE format
<code>long double</code>	12 byte IEEE format

NOTE The section “C Implementation Limits description”, later in this appendix, presents the range of values each floating point type can take on.

Converting an integer to a floating-point

An integral value is truncated to the nearest floating-point representation if it cannot be converted exactly.

Converting a floating-point to a narrower floating-point

When a floating-point number is converted to a narrower floating-point number, the direction of truncation or rounding is towards the nearest floating-point number.

An overflow causes a run-time exception to occur. For example:

```
double d;
float f;

main()
{
    d = 1.2345e300;
    f = d;           /* dumps core */
}
```

If an underflow occurs, the value is rounded towards 0.

Arrays and pointers

This section describes the implementation-defined characteristics of arrays and pointers. It corresponds to section “F.3.7 Arrays and Pointers” in the ANSI document.

Largest array size

The type of integer required to hold the maximum size of an array, that is, the type of the `sizeof` operator, `size_t`, is an **unsigned int**.

Casting pointers

The effect of casting a pointer to an integer or vice versa depends on the type of **int**. When casting any pointer into a **long int** or an **int** and back, no information is lost. However, when casting any pointer to type **char** or **short** and back, the original pointer will not be restored.

Pointer subtraction

The difference between two pointers to members of the same array is held in `ptrdiff_t`, a **signed int**.

Registers

This section describes the implementation-defined characteristics of registers. It corresponds to section “F.3.8 Registers” in the ANSI document.

Using registers

In the absence of optimization, the compiler will place the first three variables declared with **register** storage class into registers as long as the variable is accessed and is of an appropriate type. Appropriate types are **signed** and **unsigned** integral types **long**, **int**, and **short**, and pointers. In the presence of optimization, the **register** storage class may be ignored.

Structures, unions, enumerations, and bit-fields

This section describes the implementation-defined characteristics of structures, unions, enumerations, and bit-fields. It corresponds to section “F.3.9 Structures, Unions, Enumerations, and Bit-Fields” in the ANSI document.

Improper access to a union

Attempts to access a union member will always succeed. The value of the member object will be the value of the first “n” bytes of the union (where “n” is the size of the object being accessed) interpreted according to the type of the accessed member.

Padding and alignment of members of structures

The Intel® 80x86 does not impose a restriction on the alignment of objects; any object can start at any address. However, for certain objects, having a particular starting address can speed up processor access.

The C compiler aligns the whole structure on a 4-byte boundary by default (see “Pragmas” (page 496)). All [4 | 8 | 10]-byte objects are aligned on a 4-byte boundary, 2-byte objects are aligned on a 2-byte boundary, while 1-byte objects are not aligned.

Sign of bit-fields

A “plain” **int** bit-field is treated as a **signed int** bit-field.

Order of allocation of bit-fields

Bit-fields are allocated in an **int** starting from the lowest order bit to the highest. For example, the structure

```
struct { int x:10; int y:20; int z:2;} y;
```

would be stored like

```
zzyyyyyyyyyyyyyyyyyyyyyxxxxxxxxx
4444444433333333222222211111111
```

where z, y, and x represent the bits of the corresponding bit-field, and 1, 2, 3, and 4, represent the bits of the corresponding bytes.

The order of allocation of bit-fields within an **int** is determined by the architecture of Intel processors which store the low order byte of an **int** at a lower address than the highest order byte in that **int**. For example:

1st byte	- low memory or address 'x'
2nd byte	- address 'x' + 1
3rd byte	- address 'x' + 2
4th byte	- address 'x' + 3

Alignment of bit-fields

A bit-field cannot straddle its storage-unit boundary, an **int**, which is 32 bits wide.

The type of values of an enumerated type

The values of an enumeration type are represented by an **int**.

Qualifiers

This section describes the implementation-defined characteristics of qualifiers. It corresponds to section "F.3.10 Qualifiers" in the ANSI document.

Access to volatile objects

Any reference to or use of an object that has been qualified with the volatile type constitutes an access to that object unless the compiler is able to determine that the object cannot be volatile. All parts of a referenced object are assumed to be referenced if any part of the object is referenced.

Declarators

This section describes the implementation-defined characteristics of declarators. It corresponds to section "F.3.11 Declarators" in the ANSI document.

Maximum number of declarators

An unlimited number of declarators (pointer, array, or function) may modify an arithmetic, structure, or union type. Note that for COFF debugging purposes only, the maximum number of declarators is "6".

Statements

This section describes the implementation-dependent characteristics of statements. It corresponds to section “F.3.12 Statements” in the ANSI document.

Maximum number of case values

There is no fixed limit to the number of case values in a switch statement.

Preprocessing directives

This section describes the implementation-defined characteristics of preprocessing directives. It corresponds to section “F.3.13 Preprocessing Directives” in the ANSI document.

Character constants and conditional inclusion

The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. This may be a negative value.

Locating includable source files

To locate bracketed, includable source files, the preprocessor searches through any paths specified on the command line by the **-I** option and then through the default path or paths, normally an API-specific hierarchy such as */usr/include/ansi*, and */usr/include*.

Including files with quoted names

To locate quoted, includable source files, the preprocessor looks in the directory containing the file being compiled, then searches through any paths specified on the command line by the **-I** option, and then searches through the default path or paths, normally an API-specific hierarchy such as */usr/include/ansi*, and */usr/include*. If the file is not found, the preprocessor reparses the the “#include string newline” as if it was a “< >” delimited file and repeats the search as described in the previous section.

Filenames can be any legal UNIX filename including a full specification of paths.

Character sequences

Both source and execution sets include the full ASCII character sets. The delimited sequence is mapped directly into the external filename.

Pragmas

The behavior of each recognized **#pragma** directive in C is listed below:

asm

```
#pragma asm [full_optimization|partial_optimization] asm_func
```

specifies full or partial optimization for *asm_func*, which must have an entry in the symbol table and be an asm.

comment

```
#pragma comment (compiler)
```

which is silently ignored,

or

```
#pragma comment (comment_type, "string")
```

where *comment_type* must be one of **exestr**, **user** or **lib**.

When *comment_type* is **exestr**, the string, *string*, is put in a comment section of the object file. This is equivalent to the previous non-ANSI construction **#ident "string"**. When *comment_type* is either **user** or **lib**, the **#pragma** statement is silently ignored.

ident

```
#pragma ident "version"
```

which is identical in function to the previous non-ANSI construction **#ident "version"**.

int_to_unsigned

```
#pragma int_to_unsigned identifier
```

which identifies *identifier* as a function whose type was **int** in previous releases of the compilation system, but whose type is **unsigned int** in this release. The declaration for *identifier* must precede the **#pragma**.

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

#pragma int_to_unsigned makes it possible for the compiler to identify expressions in which the function's changed type may affect the evaluation of the expression. In the **-Xk** or **-Xt** mode, the compiler treats the function as if it were declared to return **int** rather than **unsigned int**.

pack

```
#pragma pack (n)
```

which controls the layout of structure offsets. *n* is a number, 1, 2, or 4, that specifies the strictest alignment desired for any structure member. If *n* is omitted, the alignment reverts to the default, which may have been set by the **-Zp** option to **cc**.

A value of 4 is the default.

weak

```
#pragma weak identifier
```

which identifies *identifier* as a weak global symbol,

or

```
#pragma weak identifier = identifier2
```

which identifies *identifier* as a weak global symbol whose value is the same as *identifier2*. *identifier* should otherwise be undefined. For more information on weak global symbols, see "Multiply defined symbols" (page 60).

Definitions for date and time

The macros **__DATE__** and **__TIME__** return the date and time of translation respectively. The date and time are always available to the macros.

Library functions

This section describes the implementation-defined characteristics of the compiler library functions. It corresponds to section “F.3.14 Library Functions” in the ANSI document.

Expanding the NULL macro

The macro **NULL** expands to the null pointer constant:

0

Diagnostic printed by the assert function

The **assert** function prints the diagnostic

"Assertion failed: <expression>, <filename>, <line number>"

On termination, the **assert** function prints out a string of the form:

Assertion failed: <comparison operation>, file <source filename>, line <source line number>.

Character testing

The sets of characters that the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions test for are listed below in the section “Locale-Specific Behavior”.

Math functions and domain errors

Domain errors occur when an input argument is outside the domain over which the mathematical function is defined. When inputs are **NaNs** or infinities, all the mathematical functions may return various forms of **NaNs**. When inputs are not **NaNs** or infinities the values returned by the mathematics functions on the occurrence of domain errors are as presented in the following table:

Table A-2 Math functions and domain errors

Function	Returns
<code>pow()</code>	-HUGE_VAL or -HUGE or 0.0 depending on input values.
<code>sqrt()</code>	0.0
<code>asin()</code>	0.0
<code>acos()</code>	0.0
<code>fmod()</code>	0.0
<code>atan2()</code>	0.0

The value of **EDOM** is "33".

Underflow of floating-point values

On underflow range errors, the mathematics functions listed below set the integer expression **errno** to the value of the macro **ERANGE**:

`exp()`
`pow()`
`ldexp()`

The mathematics functions listed below will get range errors and set the integer expression **errno** to the value of the macro **ERANGE** only if the input argument is $(\pm)\infty$.

`sin()`
`cos()`
`exp()`
`log()`
`tan()`
`pow()`
`sinh()`
`cosh()`
`tanh()`
`ldexp()`

Domain errors and the fmod function

The **fmod** function computes the floating point remainder of x/y . A domain error occurs and zero is returned when the **fmod** function has a second argument of zero (**errno**=**EDOM**).

The signal function

The set of signals for the **signal** function are:

Table A-3 signal function messages

Signal	Number	Function
SIGHUP	1	hangup
SIGINT	2	interrupt (rubout)
SIGQUIT	3	quit (ASCII FS)
SIGILL	4	illegal instruction (not reset when caught)
SIGTRAP	5	trace trap (not reset when caught)
SIGIOT	6	IOT instruction
SIGABRT	6	used by abort, replace SIGIOT in the future
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child
SIGCHLD	18	death of a child
SIGPWR	19	power-fail restart
SIGWINCH	20	window change
SIGPOLL	22	pollable event occurred
SIGSTOP	23	sendable stop signal not from tty
SIGTSTP	24	stop signal from tty
SIGCONT	25	continue a stopped process
SIGTTIN	26	to readers pgrp upon background tty read
SIGTTOU	27	like TTIN for output if tp->t_local&TOSTOP
SIGVTALRM	28	virtual timer alarm
SIGPROF	29	profile alarm
SIGXCPU	30	CPU time limit exceeded
SIGXFSZ	31	File size limit exceeded

Default signals

The default handling for each signal recognized by the **signal** function is to not ignore these signals.

A program runs in a process which inherits its handling of signals from its parent process (such as the shell). If signals are set to **SIG_IGN** in the parent process, that remains the same. If the signals were set to be caught in the parent they are reset to **SIG_DFL** which may be to ignore the signal or terminate the program. The setting of **SIG_DFL** remains the same.

Signal blocking

If the equivalent of "signal(sig, **SIG_DFL**);" is not executed prior to the call of a signal handler, signal blocking is performed. Signals are set to their default values just before program execution.

The SIGILL signal

The default is not reset if the **SIGILL** signal is received by a handler specified to the signal function.

Terminating new-line characters

A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a new-line character. The last line of a text stream does not require a terminating new-line character.

Space characters before a new-line character

Space characters written out to a text stream immediately before a new-line character appear when read in.

Null characters appended to a binary stream

A binary stream is an ordered sequence of characters that can transparently record internal data. An unlimited number of null characters may be appended to data written to a binary stream.

File position in append mode

The file position indicator of an append mode stream is initially positioned at the end of the file.

File buffering

Files are fully buffered or unbuffered depending on the setting of **setbuf()** and **setvbuf()**. The default buffer size is set by **BUFSIZ** in *stdio.h*.

The existence of zero-length files

A zero-length file, one on which no characters have been written by an output stream, actually exists.

Composing valid filenames

The rules for composing valid filenames are:

- The maximum filename length is 14 characters under SCO release 3.2.2.
- The maximum filename length is 255 characters under SCO release 3.2.4.
- The maximum filename length is at least 255 characters and can vary upwards with the type of file system under this release.
- All ASCII characters are acceptable with the exception of `"/"`.
- File names `“.”` and `“..”` are not acceptable.

File access limits

The same file can be simultaneously open multiple times.

Removing open files

The effect of the **remove** function on an open file is to make the file inaccessible to other programs or to users. The file remains accessible to the program, and other running programs that have already opened the file, through already open file descriptors, but not through new attempts to open a file with the same name. Once all open file descriptors are closed, the file will be irrevocably gone unless versioning is enabled.

Renaming with a name that exists

If a file with the new name exists prior to a call to the **rename** function, **rename** will succeed, overwriting the existing file in the process.

Output of pointer values

The output for `%p` conversion in the **fprintf** function is a hex number. The value may be preceded by a `“0x”` prefix if `%#p` is used.

Input of pointer values

The input for **%p** conversion in the **fscanf** function is a number of hex digits interpreted as a pointer. If **%#p** is used, "0x" is used as a prefix. This option is identical to the **%x** conversion specifier.

Reading ranges

A "-" character that is neither the first nor the last character in the scanlist for **%[** conversion in the **fscanf** function is interpreted as a range indicator as long as the character to the left is less than the character to the right of the "-". The following example selects all upper-case ASCII characters from "A" to "Z" inclusive:

```
%[A-Z]
```

File position errors

The **errno** is set to **EBADF** or **ESPIPE** by the **fgetpos** or **ftell** function on failure. **EINVAL** may be generated by the **lseek()** system call (**ftell()** and **fgetpos()** call **lseek()**).

Messages generated by the perror function

The messages generated by the **perror** function look like:

```
<some user message>: <error message>
```

The error message is selected, from the list shown below under **strerror**, depending on the value of **errno** just before calling **perror()**. If **errno** has a value of "0" then the first string in the list is used and so on.

Allocating zero memory

The **calloc**, **malloc** and **realloc** functions return a pointer to a zero-sized block if the size requested is zero. For **calloc** and **malloc** this is different behavior from the behavior of the functions in **libmalloc.a** which return a **NULL** on **malloc(0)**.

The abort function and open and temporary files

When the **abort** function is executed, the buffers of open and temporary files are flushed, all files are closed, and execution returns to the shell.

The exit function

The **exit()** function returns the lower byte of its argument to the shell if the value of the argument is other than zero, **EXIT_SUCCESS**, or **EXIT_FAILURE**. The shell or the user may interpret this value in any way. Unsuccessful termination is indicated by a return of non-zero. Successful termination is indicated by a return value of zero.

Environment names

The set of environment names is that set exported by the parent process.

The system function

Any UNIX command line that is acceptable to the Bourne Shell can be passed as the string to the **system** function. The Bourne Shell is assumed to be the command processor and that shell is run as a child process to the program's (that is, parent) process.

The strerror function

The **strerror** function generates the following messages:

Table A-4 **strerror** function messages

Number	Message	Error
1	Operation not permitted	EPERM
2	No such file or directory	ENOENT
3	No such process	ESRCH
4	Interrupted system call	EINTR
5	I/O error	EIO
6	No such device or address	ENXIO
7	Arg list too long	E2BIG
8	Exec format error	ENOEXEC
9	Bad file number	EBADF
10	No child processes	ECHILD
11	Resource temporarily unavailable	EAGAIN
12	Not enough space	ENOMEM
13	Permission denied	EACCES
14	Bad address	EFAULT
15	Block device required	ENOTBLK
16	Device busy	EBUSY

(Continued on next page)

Table A-4 **strerror** function messages*(Continued)*

Number	Message	Error
17	File exists	EEXIST
18	Cross-device link	EXDEV
19	No such device	ENODEV
20	Not a directory	ENOTDIR
21	Is a directory	EISDIR
22	Invalid argument	EINVAL
23	File table overflow	ENFILE
24	Too many open files	EMFILE
25	Inappropriate I/O control operation	ENOTTY
26	Text file busy	ETXTBSY
27	File too large	EFBIG
28	No space left on device	ENOSPC
29	Illegal seek	ESPIPE
30	Read-only file system	EROFS
31	Too many links	EMLINK
32	Broken pipe	EPIPE
33	Argument out of domain	EDOM
34	Result too large or too small	ERANGE
35	No message of desired type	ENOMSG
36	Identifier removed	EIDRM
37	Channel number out of range	ECHRNG
38	Level 2 not synchronized	EL2NSYNC
39	Level 3 halted	EL3HLT
40	Level 3 reset	EL3RST
41	Link number out of range	ELNRNG
42	Protocol driver not attached	EUNATCH
43	No CSI structure available	ENOCSI
44	Level 2 halted	EL2HLT
45	Deadlock situation detected/avoided	EDEADLK
46	No record locks available	ENOLCK
50	Bad exchange descriptor	EBADE
51	Bad request descriptor	EBADR
52	Message tables full	EXFULL
53	Anode table overflow	ENOANO
54	Bad request code	EBADRQC
55	Invalid slot	EBADSLT
56	File locking deadlock	EDEADLOCK
57	Bad font file format	EBFONT

(Continued on next page)

Table A-4 strerror function messages
(Continued)

Number	Message	Error
60	Not a stream device	ENOSTR
61	No data available	ENODATA
62	Timer expired	ETIME
63	Out of stream resources	ENOSR
64	Machine is not on the network	ENONET
65	Package not installed	ENOPKG
66	Object is remote	EREMOTE
67	Link has been severed	ENOLINK
68	Advertise error	EADV
69	Srmount error	ESRMNT
70	Communication error on send	ECOMM
71	Protocol error	EPROTO
74	Multihop attempted	EMULTIHOP
75	Inode is remote	ELBIN
76	Cross mount point	EDOTDOT
77	Not a data message	EBADMSG
78	Filename too long	ENAMETOOLONG
79	Value too large for defined data type	E_OVERFLOW
80	Name not unique on network	ENOTUNIQ
81	File descriptor in bad state	EBADFD
82	Remote address changed	EREMCHG
83	Can not access a needed shared library	ELIBACC
84	Accessing a corrupted shared library	ELIBBAD
85	.lib section in a.out corrupted	ELIBSCN
86	Attempting to link in more shared libraries than system limit	ELIBMAX
87	Can not exec a shared library directly	ELIBEXEC
88	Illegal byte sequence	EILSEQ
89	Function not implemented	ENOSYS
90	Operation would block	_TCPERR
91	Operation now in progress	EINPROGRESS
92	Operation already in progress	EALREADY
93	Socket operation on non-socket	ENOTSOCK
94	Destination address required	EDESTADDRREQ
95	Message too long	EMSGSIZE
96	Protocol wrong type for socket	EPROTOTYPE
97	Protocol not supported	EPROTONOSUPPORT
98	Socket type not supported	ESOCKTNOSUPPORT

(Continued on next page)

Table A-4 strerror function messages
(Continued)

Number	Message	Error
99	Operation not supported on socket	EOPNOTSUPP
100	Protocol family not supported	EPFNOSUPPORT
101	Address family not supported by protocol family	EAFNOSUPPORT
102	Address already in use	EADDRINUSE
103	Cannot assign requested address	EADDRNOTAVAIL
104	Network is down	ENETDOWN
105	Network is unreachable	ENETUNREACH
106	Network dropped connection on reset	ENETRESET
107	Software caused connection abort	ECONNABORTED
108	Connection reset by peer	ECONNRESET
110	Socket is already connected	EISCONN
111	Socket is not connected	ENOTCONN
112	Cannot send after socket shutdown	ESHUTDOWN
113	Too many references: cannot splice	ETOOMANYREFS
114	Connection timed out	ETIMEDOUT
115	Connection refused	ECONNREFUSED
116	Host is down	EHOSTDOWN
117	No route to host	EHOSTUNREACH
118	Protocol not available	ENOPROTOOPT
135	Structure needs cleaning	EUCLEAN
137	Not a name file	ENOTNAM
138	Not available	ENAVAIL
139	Is a name file	EISNAM
140	Remote I/O error	EREMOTEIO
141	Reserved for future use	EINIT
142	Unknown error: 142	EREMDEV
145	Directory not empty	ENOTEMPTY
150	Too many symbolic links in path	ELOOP
151	Stale NFS file handle	ESTALE
152	Restartable system call	ERESTART
153	No sleeping in stream head of pipe/FIFO	ESTRPIPE
500	Unknown error	EIORESID

The time zone

The local time zone (with or without daylight savings) is read from an environment variable called **TZ** at program runtime. That variable assists in interpreting the system clock. If no such environment variable is set the default is to GMT (non-daylight savings time).

The clock function

The era for the **clock** function is: January 1 1970 00:00 GMT.

Locale-specific behavior

This section describes the locale-specific behavior of the hosted environment. It corresponds to section “F.4 Locale-Specific Behaviour” in the ANSI document.

Content of execution character set

In the default “C” locale, the execution character set contains the entire 8-bit character set.

Direction of printing

The direction of printing is left to right.

Decimal point character

The decimal-point character is “.”.

Character testing and case mapping

The implementation-defined aspects of character testing functions are presented in the following table:

Table A-5 Character testing functions

Function	Character(s)	ASCII value
isalnum()	0 - 9	48 through 57
	A - Z	65 through 90
	a - z	97 through 122
isalpha()	A - Z	65 through 90
	a - z	97 through 122
iscntrl()		0 through 31 and 127
isdigit()	0 - 9	48 through 57
isgraph()	0 - 9	48 through 57
	A - Z	65 through 90
	a - z	97 through 122
	!"#\$%&'()*+,-./	33 through 47
	;<=>?@	58 through 64
	[\]^_`	91 through 96
	{ }~	123 through 126
islower()	a - z	97 through 122
isprint()	0 - 9	48 through 57
	A - Z	65 through 90
	a - z	97 through 122
	!"#\$%&'()*+,-./	33 through 47
	;<=>?@	58 through 64
	[\]^_`	91 through 96
	{ }~	123 through 126
ispunct()	!"#\$%&'()*+,-./	33 through 47
	;<=>?@	58 through 64
	[\]^_`	91 through 96
	{ }~	123 through 126
isspace()	"", \f, \n, \r, \t, \v	32, 12, 10, 13, 9, 11
isupper()	A - Z	65 through 90
isxdigit()	0 - 9	48 through 57
	A - F	65 through 70
	a - f	97 through 103

The implementation-defined aspects of case mapping functions are presented in the following table:

Table A-6 Case mapping functions

Function	Input Character(s)	Output Character(s)
<code>toupper()</code>	a - z	A - Z
<code>tolower()</code>	A - Z	a - z

Collation sequence

In the default “C” locale, characters are collated as they would be if they were interpreted as numeric values. For normal letters, this corresponds to the normal alphabetic collation sequence.

Time and date formats

The formats for time and date are AM and PM for morning and evening 12-hour periods of the day respectively.

The short formats for names of the days of the week are: Sun, Mon, Tue, Wed, Thu, Fri, Sat. The long formats for names of the days of the week are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

The short formats for the names of the months are: Jan, Feb, Mar, Apr, Jun, Jul, Aug, Sep, Oct, Nov, Dec. The long formats for the names of the months are: January, February, March, April, June, July, August, September, October, November, December.

C Implementation limits description

This section describes the limits on values defined for this implementation.

Environmental limits

The following environmental limits are defined for this implementation:

Table A-7 limits.h

Variable	Limit
CHAR_BIT	8
CHAR_MAX	SCHAR_MAX
CHAR_MIN	SCHAR_MIN
INT_MAX	2147483647
INT_MIN	-INT_MAX-1
LONG_MAX	2147483647L
LONG_MIN	-LONG_MAX-1
MB_LEN_MAX	5
SCHAR_MIN	(-128)
SCHAR_MAX	127
SHRT_MAX	32767
SHRT_MIN	(-32768)
UCHAR_MAX	255
UINT_MAX	4294967295U
ULONG_MAX	4294967295UL
USHRT_MAX	65535

Table A-8 float.h

Variable	Limit
FLT_ROUNDS	(+__flt_rounds)
FLT_RADIX	2
FLT_MANT_DIG	24
FLT_EPSILON	((float)1.19209290E-07F)
FLT_DIG	6
FLT_MIN_EXP	(-125)
FLT_MIN	((float)1.17549435E-38F)
FLT_MIN_10_EXP	(-37)
FLT_MAX_EXP	128
FLT_MAX	((float)3.40282347E+38F)
FLT_MAX_10_EXP	38
DBL_MANT_DIG	53
DBL_EPSILON	2.2204460492503131E-16
DBL_DIG	15
DBL_MIN_EXP	(-1021)
DBL_MIN	2.2250738585072014E-308
DBL_MIN_10_EXP	(-307)
DBL_MAX_EXP	1024
DBL_MAX	1.7976931348623157E+308
LDBL_MANT_DIG	64
LDBL_EPSILON	1.0842021724855044340075E-19L
LDBL_DIG	18
LDBL_MIN_EXP	(-16381)
LDBL_MIN	(+__ldmin[0])
LDBL_MIN_10_EXP	(-4931)
LDBL_MAX_EXP	16384
LDBL_MAX	(+__ldmax)
LDBL_MAX_10_EXP	4932

Translation limits

The following translation limits are defined for this implementation:

Table A-9 Defined translation limits

Item	Actual	Minimum Acceptable
nesting levels of compound statements, iteration control structures, and selection control structures.	≥ 256	15
nesting levels of conditional inclusion	≥ 32	8
pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration	≥ 32	12
declarators nested by parentheses within a full declarator	≥ 32	12
expressions nested by parentheses within a full expression	≥ 128	32
significant initial characters in an internal identifier or a macro name	≥ 1024	31
significant initial characters in an external identifier	≥ 1024	6
external identifiers in one translation unit	≥ 1024	511
identifiers with block scope declared in one block	≥ 2048	127
macro identifiers simultaneously defined in one translation unit	≥ 4096	1024
parameters in one function definition	≥ 256	31
arguments in one function call	≥ 256	31
parameters in one macro definition	≥ 128	31
arguments in one macro invocation	≥ 128	31
characters in a logical source line	≥ 4096	509
characters in a character string literal or wide string literal (after concatenation)	≥ 4096	509
bytes in an object (in a hosted environment)	≥ 65535	32767
nesting levels for <code>#include'd</code> files	≥ 16	8
case labels for a switch statement (excluding those for any nested switch statements)	unlimited	257
members in a single structure or union	≥ 512	127
enumeration constants in a single enumeration	≥ 512	127
levels of nested structure or union definitions in a single struct-declaration-list	≥ 64	15
functions that can be registered by <code>atexit()</code> .	37	32

Appendix B

C compiler diagnostics

This chapter contains the text and explanation for all the warning and error messages produced by the compiler. The messages are listed in alphanumeric order (special characters are ignored). Numbers precede capital letters and capital letters precede lowercase letters. *n*, when it represents a number, comes at the beginning of the list.

The message entries are formatted as follows:

Entry	Comment
cannot initialize typedef: <i>name</i>	Text of message.
Type: Error — Options: all	Type of message and command-line options (see next page)
A typedef may not have an initializer.	Explanation of message.
typedef int INT = 1;	Example of code that might generate the message.
<i>"file"</i> , line 1: cannot initialize typedef: INT Message output.	

When an error occurs, the error message is preceded by a file name and line number. All message output are one line long (no newline characters). The line number is usually the line on which a problem has been diagnosed. Occasionally the compiler must read the next token before it can diagnose a problem, in which case the line number in the message may be a higher line number than that of the offending line.

Note that **lint**(CP) issues all the messages listed in this chapter, and additional messages about potential bugs and portability problems.

NOTE For further information, see Chapter 5, “lint analyzer” (page 153).

Message types and applicable options

Each message description includes a *Type* and an *Options* field as follows:

Type Indicates whether the message is a warning, an error, a fatal error, or a combination of error types (see below).

Options Indicates which **cc** command options must be set for the message to appear. “all” implies that the message is independent of **cc** options.

The following paragraphs explain the differences between warnings messages, error messages, and fatal errors.

Warning messages, in which **warning:** appears after the file name and line number, provide useful information without interrupting compilation. They may diagnose a programming error, or a violation of C syntax or semantics, for which the compiler will nevertheless generate valid object code.

Error messages, in which **error:** appears after the file name and line number, will cause the **cc** command to fail. Errors occur when the compiler has diagnosed a serious problem that makes it unable to understand the program or to continue to generate correct object code. It will attempt to examine the rest of the program for other errors, however. The **cc** command will not link the program if the compiler diagnoses errors.

Fatal errors cause the compiler to stop immediately and return an error indication to the **cc** command. A fatal error message is prefixed with **fatal:**. Such messages typically apply to start-up conditions, such as being unable to find a source file.

Operator names in messages

Some messages include the name of a compiler operator, as in:

operands must have arithmetic type: op "+"

Usually the operator in the message is a familiar C operator. At other times the compiler uses its internal name for the operator, like **U-**. The “Operator Names” section of this document, found after the message list, lists these internal names and describes what they mean.

Messages

n extra byte(s) in string literal initializer ignored

Type: Warning — *Options:* all

A string literal that initializes a character array contains *n* more characters than the array can hold.

```
char ca[3] = "abcd";
```

"*file*", line 1: warning: 1 extra byte(s) in string literal initializer ignored

0 is invalid in # <number> directive

Type: Error — *Options:* all

The line number in a line number information directive (which the compiler uses for internal communication) must be a positive, non-zero value.

```
# 0 "foo.c"
```

"*file*", line 1: 0 is invalid in # <number> directive

0 is invalid in #line directive

Type: Error — *Options:* all

This diagnostic is similar to the preceding one, except the invalid line number appeared in a **#line** directive.

```
#line 0
```

"*file*", line 1: 0 is invalid in #line directive

ANSI C behavior differs; not modifying typedef with *modifier*

Type: Warning — *Options:* **-Xa**, **-Xc**

A **typedefed** type may not be modified with the **short**, **long**, **signed**, or **unsigned** type modifiers, although earlier releases permitted it. *modifier* is ignored. A related message is "modifying typedef with "*modifier*"; only qualifiers allowed".

```
typedef int INT;
```

```
unsigned INT ui;
```

"*file*", line 2: warning: ANSI C behavior differs; not modifying typedef with "unsigned"

ANSI C predefined macro cannot be redefined

Type: Warning — *Options:* all

The source code attempted to define or redefine a macro that is predefined by ANSI C. The predefined macro is unchanged.

```
#define __FILE__ "xyz.c"
```

"file", line 1: warning: ANSI C predefined macro cannot be redefined

ANSI C predefined macro cannot be undefined

Type: Warning — *Options:* all

The source code contains an attempt to undefine a macro that is predefined by ANSI C.

```
#undef __FILE__
```

"file", line 1: warning: ANSI C predefined macro cannot be undefined

ANSI C requires formal parameter before ...

Type: Warning — *Options:* **-Xc, -w3**

The compiler implementation allows you to define a function with a variable number of arguments and no fixed arguments. ANSI C requires at least one fixed argument.

```
f(...) {}
```

"file", line 1: warning: ANSI C requires formal parameter before "..."

ANSI C treats constant as unsigned: op *operator*

Type: Warning — *Options:* **-w3, -w2**

The type promotion rules for ANSI C are slightly different from those of previous releases. Obtain the ANSI C interpretation by using the **-Xa** or **-Xc** option for the **cc** command, obtain the old behavior by using **-Xt** or **-Xk**.

Previous type promotion rules were “unsigned-preserving.” If one of the operands of an expression was of unsigned type, the operands were promoted to a common unsigned type before the operation was performed.

ANSI C uses “value-preserving” type promotion rules. An unsigned type is promoted to a signed type if all its values may be represented in the signed type.

ANSI C also has a different rule from previous releases for the type of an integral constant that implicitly sets the sign bit.

The different type promotion rules may lead to different program behavior for the operators that are affected by the unsigned-ness of their operands:

- The division operators: `/`, `/=`, `%`, `%=`.
- The right shift operators: `>>`, `>>=`.
- The relational operators: `<`, `<=`, `>`, `>=`.

The warning message tells that the program contains an expression in which the behavior of *operator* will change in the future. To guarantee the desired behavior, insert an explicit cast in the expression.

```
f(void){
    int i;
    /* constant was integer, unsigned in ANSI C */
    i /= 0xf0000000;
}
```

"file", line 4: warning: ANSI C treats constant as unsigned: op `"/="`

To get the same behavior as in previous releases add an explicit cast:

```
f(void){
    int i;
    /* constant was integer, unsigned in ANSI C */
    i /= (int) 0xf0000000;
}
```

-D option argument not an identifier

Type: Error — Options: all

An identifier must follow the **-D** cc command line option.

```
cc -D3b2 -c x.c
```

command line: **-D** option argument not an identifier

-D option argument not followed by =

Type: Warning — Options: all

If any tokens follow an identifier in a **-D** command line option to the cc command, the first such token must be `"="`.

```
cc -DTWO+2 -c x.c
```

command line: warning: **-D** option argument not followed by `"="`

EOF in argument list of macro: *name*

Type: Error — Options: all

The compiler reached end-of-file while reading the arguments for an invocation of function-like macro *name*.

```
#define mac(a)
mac( arg1
```

"file", line 5: EOF in argument list of macro: `mac`

EOF in asm function definition

Type: Error — *Options:* all

The compiler reached end-of-file while reading an enhanced **asm** function definition.

EOF in character constant

Type: Error — *Options:* all

The compiler encountered end-of-file inside a character constant.

EOF in comment

Type: Warning — *Options:* all

The compiler encountered end-of-file while reading a comment.

EOF in string literal

Type: Error — *Options:* all

The compiler encountered end-of-file inside a string literal.

NUL in asm function definition

Type: Warning — *Options:* all

The compiler encountered a NUL (zero) character while reading an enhanced **asm** function definition. The NUL is ignored.

-U option argument not an identifier

Type: Error — *Options:* all

An identifier must follow the **-U** cc command line option.

```
cc -U3b2 -c x.c
```

```
command line: -U option argument not an identifier
```

a cast does not yield an lvalue

Type: Warning, Error — *Options:* all

A cast may not be applied to the operand that constitutes the object to be changed in an assignment operation. The diagnostic is a warning if the size of the operand type and the size of the type being cast to are the same; otherwise it is an error.

```
f(void){
    int i;
    (long) i = 5;
    (short) i = 4;
}
```

"file", line 3: warning: a cast does not yield an lvalue

"file", line 4: a cast does not yield an lvalue

`\a` is ANSI C alert character

Type: Warning — *Options:* **-Xt**

In earlier releases, “`\a`” was equivalent to “`a`”. However, ANSI C defines “`\a`” to be an alert character. In the implementation, the corresponding character code is **07**, the **BEL** character.

```
int c = '\a';
```

"file", line 1: warning: `\a` is ANSI C "alert" character

access through void pointer ignored

Type: Warning — *Options:* all

A pointer to **void** may not be used to access an object. An expression was written that does an indirection through a (possibly qualified) pointer to **void**. The indirection is ignored, although the rest of the expression (if any) is honored.

```
f(){
    volatile void *vp1, *vp2;
    *(vp1 = vp2);          /* assignment does get done */
}
```

"file", line 3: warning: access through "void" pointer ignored

argument cannot have unknown size: arg #*n*

Type: Error — *Options:* all

An argument in a function call must have a completed type. A **struct**, **union**, or **enum** object was passed whose type is incomplete.

```
f(){
    struct s *st;
    g(*st);
}
```

"file", line 3: argument cannot have unknown size: arg #1

argument does not match remembered type: arg #*n*

Type: Warning — *Options:* **-w3**

At a function call, the compiler determined that the type of the *n*-th argument passed to a function disagrees with other information it has about the function. That other information comes from two sources:

1. An old-style (non-prototype) function definition, or
2. A function prototype declaration that has gone out of scope, but whose type information is still remembered.

The argument in question is promoted according to the default argument promotion rules.

This diagnostic may be incorrect if the old-style function definition case applies and the function takes a variable number of arguments.

```
void f(i)
int i;
{ }

void g()
{
    f("erroneous");
}
```

"file", line 7: warning: argument does not match remembered type: arg #1

argument is incompatible with prototype: arg #*n*

Type: Error — Options: all

A function was called with an argument whose type cannot be converted to the type in the function prototype declaration for the function.

```
struct s {int x;} q;
f(void){
    int g(int,int);
    g(3,q);
}
```

"file", line 4: argument is incompatible with prototype: arg #2

argument mismatch

Type: Warning — Options: all

The number of arguments passed to a macro was different from the number in the macro definition.

```
#define twoarg(a,b) a+b
int i = twoarg(4);
```

"file", line 2: warning: argument mismatch

argument mismatch: *n1* arg[*s*] passed, *n2* expected

Type: Warning — Options: -w3

At a function call, the compiler determined that the number of arguments passed to a function disagrees with other information it has about the function. That other information comes from two sources:

1. An old-style (non-prototype) function definition, or
2. A function prototype declaration that has gone out of scope, but whose type information is still remembered.

This diagnostic may be incorrect if the old-style function definition case applies and the function takes a variable number of arguments.

```

extern int out_of_scope();
int f()
{
    /* function takes no args */
    extern int out_of_scope(int);
}

int g()
{
    f(1);                /* f takes no args */
    out_of_scope();       /* out_of_scope expects one arg */
}

```

"file", line 9: warning: argument mismatch: 1 arg passed, 0 expected

"file", line 10: warning: argument mismatch: 0 args passed, 1 expected

array too big

Type: Error — Options: all

An array declaration has a combination of dimensions such that the declared object is too big for the target machine.

```
int bigarray[1000][1000][1000];
```

"file", line 1: array too big

asm() argument must be normal string literal

Type: Error — Options: all

The argument to an old-style **asm()** must be a normal string literal, not a wide one.

```
asm(L"wide string literal not allowed");
```

"file", line 1: asm() argument must be normal string literal

asm definition cannot have old-style parameters

Type: Error — Options: all

The definition of an enhanced **asm** function may use the ANSI C function prototype notation to declare types for parameters. It may not declare parameters by using the old-style C function definition notation of an identifier list, followed by a declaration list that declares parameter types.

__asm is an extension of ANSI C

Type: Warning — Options: -Xc

Code declaring an enhanced **asm** function was compiled with **-Xc**. This warning tells that the enhanced **__asm** is a violation of ANSI C syntax, which the compiler is obliged to diagnose, and is not a compatible extension.

asm valid only for function definition

Type: Warning — *Options:* all

The **asm** storage class may only be used for function definitions. It is ignored here.

```
asm int f(void);
```

"*file*", line 1: warning: "asm" valid only for function definition

#assert identifier (... expected

Type: Error — *Options:* all

In a **#assert** directive, the token following the predicate was not the "(" that was expected.

```
#assert system unix
```

"*file*", line 1: "#assert identifier (... " expected

#assert identifier expected

Type: Error — *Options:* all

In a **#assert** directive, the token following the directive was not the name of the predicate.

```
#assert 5
```

"*file*", line 1: "#assert identifier" expected

#assert missing)

Type: Error — *Options:* all

In a **#assert** directive, the parenthesized form of the assertion lacked a closing ")".

```
#assert system(unix
```

"*file*", line 1: "#assert" missing ")"

assignment type mismatch

Type: Warning, Error — *Options:* all

The operand types for an assignment operation are incompatible. The message is a warning when the types are pointer types that do not match. Otherwise the message is an error.

```

struct s { int x; } st;
f(void){
    int i;
    char *cp;
    const char *ccp;
    i = st;
    cp = ccp;
}

```

"file", line 6: assignment type mismatch

"file", line 7: warning: assignment type mismatch

auto/register/asm inappropriate here

Type: Error — *Options:* all

A declaration outside any function has storage class **auto** or **register** or a declaration within a function has storage class **asm**.

```

auto int i;
f(void){
    asm int j;
}

```

"file", line 1: auto/register/asm inappropriate here

"file", line 3: auto/register/asm inappropriate here

automatic redeclares external: **name**

Type: Warning — *Options:* all

An automatic variable was declared **name** in the same block and with the same name as another symbol that is **extern**. ANSI C prohibits such declarations, but previous compilation systems of allowed them. For compatibility with previous releases, references to **name** in this block will be to the automatic.

```

f(void){
    extern int i;
    int i;
}

```

"file", line 3: warning: automatic redeclares external: i

bad file specification

Type: Error — *Options:* all

The file specifier in a **#include** directive was neither a string literal nor a well-formed header name.

```
#include stdio.h
```

"file", line 1: bad file specification

bad octal digit: "*digit*"

Type: Warning — Options: **-Xt**

An integer constant that began with "0" included the non-octal digit *digit*. An "8" is taken to have value 8, and a "9" is taken to have value 9, even though they are invalid.

```
int i = 08;
```

"*file*", line 1: warning: bad octal digit: "'8'"

bad #pragma pack value: *n*

Type: Warning — Options: all

The value *n* that was specified in a **#pragma pack** directive was not one of the acceptable values: 1, 2, or 4. The erroneous value is ignored and the directive has no effect.

bad token in #error directive: *token*

Type: Error — Options: all

The tokens in a **#error** directive must be valid C tokens. The source program contained the invalid token *token*.

```
#error "this is an invalid token"
```

"*file*", line 1: bad token in #error directive: "

"*file*", line 1: #error: "this is an invalid token"

bad use of # or ## in macro #define

Type: Warning — Options: all

In a macro definition, a "#" or "##" operator was followed by a "#" or "##" operator.

```
#define bug(s) # # s
```

```
#define bug2(s) # ## s
```

"*file*", line 1: warning: bad use of "#" or "##" in macro #define

"*file*", line 2: warning: bad use of "#" or "##" in macro #define

base type is really *type tag*: *name*

Type: Warning — Options: **-Xt**

A type was declared with a **struct**, **union**, or **enum** type specifier and with tag *tag*, and then used with a different type specifier to declare *name*. *type* is the type specifier used for the original declaration. For compatibility with previous releases, the compiler treats the two types as being the same. In ANSI C (with the **-Xa** or **-Xc** options), the types are different.

```
struct s { int x,y,z; };
f(void){
    union s foo;
}
```

"file", line 3: warning: base type is really "struct s": foo

"file", line 3: warning: declaration introduces new type in ANSI C: union s

bit-field size <= 0: **name**

Type: Error — Options: all

The declaration for bit-field **name** specifies a zero or negative number of bits.

```
struct s { int x:-3; };
```

"file", line 1: bit-field size <= 0: x

bit-field too big: **name**

Type: Error — Options: all

The declaration for bit-field **name** specifies more bits than will fit in an object of the declared type.

```
struct s { char c:20; };
```

"file", line 1: bit-field too big: c

break outside loop or switch

Type: Error — Options: all

A function contains a **break** statement in an inappropriate place, namely outside any loop or **switch** statement.

```
f(void){
    break;
}
```

"file", line 2: "break" outside loop or switch

cannot access member of non-struct/union object

Type: Error — Options: all

The structure or union member must be completely contained within the left operand of the "." operator.

```
f(void){
    struct s { int x; };
    char c;
    c.x = 1;
}
```

"file", line 4: warning: left operand of "." must be struct/ union object

"file", line 4: cannot access member of non-struct/union object

cannot begin macro replacement with ##

Type: Warning — *Options:* all

The “##” operator is a binary infix operator and may not be the first token in the macro replacement list of a macro definition.

```
#define mac(s) ## s
```

"file", line 1: warning: cannot begin macro replacement with "##"

cannot concatenate wide and regular string literals

Type: Warning, Error — *Options:* all

Regular string literals and string literals for wide characters may be concatenated only if they are both regular or both wide. The compiler issues a warning if a wide string literal is followed by a regular one (and both are treated as wide); it issues an error if a regular string literal is followed by a wide one.

```
#include <stddef.h>
wchar_t wa[] = L"abc" "def";
char a[] = "abc" L"def";
```

"file", line 2: warning: cannot concatenate wide and regular string literals

"file", line 3: cannot concatenate wide and regular string literals

cannot declare array of functions or void

Type: Error — *Options:* all

An array of functions was or an array of **void** was declared.

```
int f[5]();
```

"file", line 1: cannot declare array of functions or void

cannot define defined

Type: Warning — *Options:* all

The predefined preprocessing operator **defined** may not be defined as a macro name.

```
#define defined xyz
```

"file", line 1: warning: cannot define "defined"

cannot dereference non-pointer type

Type: Error — *Options:* all

The operand of the “*” (pointer dereference) operator must have pointer type. This diagnostic is also issued for an array reference to a non-array.

```
f(){
    int i;
    *i = 4;
    i[4] = 5;
}
```

"*file*", line 3: cannot dereference non-pointer type

"*file*", line 4: cannot dereference non-pointer type

cannot do pointer arithmetic on operand of unknown size

Type: Error — Options: all

An expression involves pointer arithmetic for pointers to objects whose size is unknown.

```
f(void){
    struct s *ps;
    g(ps+1);
}
```

"*file*", line 3: cannot do pointer arithmetic on operand of unknown size

cannot end macro replacement with # or ##

Type: Warning — Options: all

A "#" or "##" operator may not be the last token in the macro replacement list of a macro definition.

```
#define mac1(s) abc ## s ##
#define mac2(s) s #
```

"*file*", line 1: warning: cannot end macro replacement with "#" or "##"

"*file*", line 2: warning: cannot end macro replacement with "#" or "##"

cannot find include file: *filename*

Type: Error — Options: all

The file *filename* specified in a **#include** directive could not be located in any of the directories along the search path.

```
#include "where_is_it.h"
```

"*file*", line 1: cannot find include file: "where_is_it.h"

cannot have ... in asm function

Type: Warning — Options: all

An enhanced **asm** definition may not be a function prototype definition with ellipsis notation.

cannot have void object: **name**

Type: Error — Options: all

An object may not be declared as having type **void**.

```
void v;
```

"file", line 1: cannot have void object: v

cannot initialize extern declaration: **name**

Type: Error — Options: all

Within a function, the declaration of an object with **extern** storage class may not have an initializer.

```
f(void){
    extern int i = 1;
}
```

"file", line 2: cannot initialize "extern" declaration: i

cannot initialize function: **name**

Type: Error — Options: all

A name declared as a function may not have an initializer.

```
int f(void) = 3;
```

"file", line 1: cannot initialize function: f

cannot initialize parameter: **name**

Type: Error — Options: all

Old-style function parameter **name** may not have an initializer.

```
int f(i)
int i = 4;
{
}
```

"file", line 2: cannot initialize parameter: i

cannot initialize typedef: **name**

Type: Error — Options: all

A **typedef** may not have an initializer.

```
typedef int INT = 1;
```

"file", line 1: cannot initialize typedef: INT

cannot open *file*: *explanation*

Type: Fatal — *Options*: all

The compiler was unable to open an input or output file. Usually this means the file name argument passed to the `cc` command was incorrect. *explanation* describes why *file* could not be opened.

```
cc glorch.c -c x.c
```

```
command line: fatal: cannot open  glorch.c: No such file or directory
```

cannot open include file (too many open files): *filename*

Type: Error — *Options*: all

The compiler could not open a new include file, *filename*, because too many other include files are already open. This scenario could arise if you have *file1* that includes *file2* that includes *file3*, and so on. The compiler supports at least eight levels of *nesting*, up to a maximum defined by the operating system. The most likely reason for the diagnostic is that at some point an include file includes a file that had already been included. For example, this could happen if *file1* includes *file2*, which includes *file1* again.

In this example, imagine that the file *i1.h* contains **#include i1.h**.

```
#include "i1.h"
```

```
"/i1.h", line 1: cannot open include file (too many open files): "i1.h"
```

cannot recover from previous errors

Type: Error — *Options*: all

Earlier errors in the compilation have confused the compiler, and it cannot continue to process the program. Please correct those errors and try again.

cannot return incomplete type

Type: Error — *Options*: all

When a function is called that returns a structure or union, the complete declaration for the structure or union must have been seen already. Otherwise this message results.

```
f(){
    struct s g();
    g();
}
```

```
"file", line 3: cannot return incomplete type
```

cannot take address of bit-field: *name*

Type: Error — *Options:* all

It is not permitted to take the address of a bit-field member of a structure or union.

```
f(void){
    struct s { int x:3, y:4; } st;
    int *ip = &st.y ;
}
```

"*file*", line 3: cannot take address of bit-field: y

cannot take address of register: *name*

Type: Warning, Error — *Options:* all

When *name* is declared with register storage class, it is not permissible to take the address of *name*, even if the compiler actually allocates the object to a **register**. The attempt to take an object's address may have been implicit, such as when an array is dereferenced. The diagnostic is an error if a register was allocated for the object and a warning otherwise.

```
f(void){
    register int i;
    register int ia[5];
    int *ip = &i ;
    ia[2] = 1;
}
```

"*file*", line 4: cannot take address of register: i

"*file*", line 5: warning: cannot take address of register: ia

cannot take sizeof bit-field: *name*

Type: Warning — *Options:* all

The **sizeof** operator may not be applied to bit-fields.

```
struct s { int x:3; } st;
int i = sizeof(st.x);
```

"*file*", line 2: warning: cannot take sizeof bit-field: x

cannot take sizeof function: *name*

Type: Error — *Options:* all

The **sizeof** operator may not be applied to functions.

```
int f(void);
int i = sizeof(f);
```

"*file*", line 2: cannot take sizeof function: f

cannot take sizeof void

Type: Error — *Options:* all

The **sizeof** operator may not be applied to type **void**.

```
void v(void);
int i = sizeof(v());
```

"file", line 2: cannot take sizeof void

cannot undefine defined

Type: Error — *Options:* all

The predefined preprocessing operator **defined** may not be undefined.

```
#undef defined
```

"file", line 1: warning: cannot undefine "defined"

case label affected by conversion: *value*

Type: Warning — *Options:* -w3

The *value* for the case label cannot be represented by the type of the controlling expression of a **switch** statement. If the type of the case expression and the type of the controlling expression have the same size, the actual bit representation of the case expression is unchanged, but its interpretation is different. For example, the controlling expression may have type **int** and the case expression may have type **unsigned int**. In the diagnostic, *value* is represented as a hexadecimal value if the case expression is unsigned, decimal if it is signed.

```
f(){
    int i;

    switch( i ){
        case 0xfffffffffu:
            ;
    }
}
```

"file", line 5: warning: case label affected by conversion: 0xffffffff

In this example **0xfffffffffu** is not representable as an **int**. When the case expression is converted to the type of the controlling expression (**int**), its effective value is **-1**, which means that the case will be reached if **i** has the value **-1**, rather than **0xffffffff**.

case outside switch

Type: Error — *Options:* all

A **case** statement occurred outside the scope of any **switch** statement.

```
f(void){
    case 4: ;
}
```

"file", line 2: "case" outside switch

character constant too long

Type: Warning — *Options:* all

The character constant contains too many characters to fit in an integer. Only the first four characters of a regular character constant, and only the first character of a wide character constant, are used. (Character constants that are longer than one character are non-portable.)

```
int i = 'abcde';
```

"file", line 1: warning: character constant too long

character escape does not fit in character

Type: Warning — *Options:* all

A hexadecimal or octal escape sequence in a character constant or string literal produces a value that is too big to fit in an **unsigned char**. The value is truncated to fit.

```
char *p = "\x1fff\400";
```

"file", line 1: warning: \x is ANSI C hex escape

"file", line 1: warning: character escape does not fit in character

"file", line 1: warning: character escape does not fit in character

character escape does not fit in wide character

Type: Warning — *Options:* all

This message diagnoses a condition similar to the previous one, except the character constant or string literal is prefixed by "L" to designate a wide character constant or string literal. The character escape is too large to fit in an object of type **wchar_t** and is truncated to fit.

comment does not concatenate tokens

Type: Warning — *Options:* -Xa, -Xc

In previous releases, it was possible to "paste" two tokens together by juxtaposing them in a macro with a comment between them. This behavior was never defined or guaranteed. ANSI C provides a well-defined operator, "##", that serves the same purpose and should be used. This diagnostic warns that the old behavior is not being provided.

```
#define PASTE(a,b) a/*GLUE*/b
int PASTE(prefix,suffix) = 1;    /* does not create */
                                /* prefixsuffix */
```

"file", line 1: warning: comment does not concatenate tokens
 "file", line 2: syntax error, probably missing "(", ";", ";" or "="
 "file", line 2: syntax error before or at: suffix
 "file", line 2: warning: declaration missing specifiers: assuming "int"

comment is replaced by ##

Type: Warning — *Options:* -Xt

This message is closely related to "comment does not concatenate tokens". The diagnostic tells that the compiler is treating an apparent concatenation as if it were the "##" operator. The source code should be updated to use the new operator.

```
#define PASTE(a,b) a/*GLUE*/b
int PASTE(prefix,suffix) = 1;    /* creates prefixsuffix */
```

"file", line 1: warning: comment is replaced by "##"

const object should have initializer: *name*

Type: Warning — *Options:* -w3

A **const** object cannot be modified. If an initial value is not supplied, the object will have a value of zero, or for automatics its value will be indeterminate.

```
const int i;
```

"file", line 1: warning: const object should have initializer: i

continue outside loop

Type: Error — *Options:* all

The program contains a **continue** statement outside the scope of any loop.

```
f(void){
    continue;
}
```

"file", line 2: "continue" outside loop

controlling expressions must have scalar type

Type: Error — *Options:* all

The expression for an **if**, **for**, **while**, or **do-while** must be an integral, floating-point, or pointer type.

```
f(void){
    struct s {int x;} st;
    while (st) {}
}
```

"*file*", line 3: controlling expressions must have scalar type

conversion of double to float is out of range

Type: Warning, Error — *Options:* all

A **double** expression has too large a value to fit in a **float**. The diagnostic is a warning if the expression is in executable code and an error otherwise.

```
float f = 1e30 * 1e30;
```

"*file*", line 1: conversion of double to float is out of range

conversion of double to integral is out of range

Type: Warning, Error — *Options:* all

A **double** constant has too large a value to fit in an integral type. The diagnostic is a warning if the expression is in executable code and an error otherwise.

```
int i = 1e100;
```

"*file*", line 1: conversion of double to integral is out of range

conversion of floating-point constant to **type** out of range

Type: Error — *Options:* all

A floating-point constant has too large a value to fit in type **type** (**float**, **double**, **long double**).

```
float f = 1e300f;
```

"*file*", line 1: conversion of floating-point constant to float out of range

declaration hides parameter: **name**

Type: Warning — *Options:* all

An identifier **name** was declared with the same name as one of the parameters of the function. References to **name** in this block will be to the new declaration.

```
int f(int i,int INT){
    int i;
    typedef int INT;
}
```

"file", line 2: warning: declaration hides parameter: i

"file", line 3: warning: declaration hides parameter: INT

declaration introduces new type in ANSI C: **type tag**

Type: Warning — *Options:* -Xt

struct, **union**, or **enum tag** has been redeclared in an inner scope. In previous releases, this tag was taken to refer to the previous declaration of tag. In ANSI C, the declaration introduces a new **type**. When the -Xt option is selected, the earlier behavior is reproduced.

```
struct s1 { int x; };
f(void){
    struct s1;
    struct s2 { struct s1 *ps1; }; /* s1 refers to line 1 */
    struct s1 { struct s2 *ps2; };
}
```

"file", line 3: warning: declaration introduces new type in ANSI C: struct s1

declaration missing specifiers; assuming int

Type: Warning — *Options:* all

Objects and functions that are declared at file scope must have a storage class or type specifier. If both are omitted the following warning is displayed:

```
i;
f(void);
```

"file", line 1: warning: declaration missing specifiers; assuming "int"

"file", line 2: warning: declaration missing specifiers; assuming "int"

default outside switch

Type: Error — *Options:* all

A **default** label appears outside the scope of a **switch** statement.

```
f(void){
default: ;
}
```

"file", line 2: "default" outside switch

`#define` requires macro name

Type: Error — *Options:* all

A **#define** directive must be followed by the name of the macro to be defined.

```
#define +3
```

"file", line 1: `#define` requires macro name

digit sequence expected after `#line`

Type: Error — *Options:* all

The compiler expected to find the digit sequence that comprises a line number after **#line**, but the token it found there is either an inappropriate token or a digit sequence whose value is zero.

```
#line 09a
```

"file", line 1: digit sequence expected after `"#line"`

directive is an upward-compatible ANSI C extension

Type: Warning — *Options:* **-Xc**

This diagnostic is issued when the C compiler sees a directive that it supports, but that is not part of the ANSI C standard, and **-Xc** has been selected.

```
#assert system( unix )
```

"file", line 1: warning: directive is an upward-compatible ANSI C extension

directive not honored in macro argument list

Type: Warning, Error — *Options:* all

A directive has appeared between the ()'s that delimit the arguments of a function-like macro invocation. The following directives are disallowed in such a context: **#ident**, **#include**, **#line**, **#undef**. The diagnostic is a warning if it appears within a false group of an if-group, and an error otherwise.

```
#define flm(a) a+4
int i = flm(
#ifdef flm                /* allowed */
    #undef flm            /* disallowed:  error */
    4
#else                    /* allowed */
    #undef flm            /* disallowed:  warn */
    6
#endif                  /* allowed */
);
```

"file", line 4: directive not honored in macro argument list

"file", line 7: warning: directive not honored in macro argument list

division by 0

Type: Warning, Error — *Options:* all

An expression contains a division by zero that was detected at compile-time. If the division is part of a **#if** or **#elif** directive, the result is taken to be zero. The diagnostic is a warning if the division is in executable code, an error otherwise.

```
f(void) {
    int i = 1/0;
}
```

"file", line 2: warning: division by 0

dubious **type** declaration; use tag only: **tag**

Type: Warning — *Options:* all

A new **struct**, **union**, or **enum type** with tag **tag** was declared within a function prototype declaration or the parameter declaration list of an old-style function definition, and the declaration includes a declarator list for **type**. Calls to the function would always produce a type mismatch, because the tag declaration goes out of scope at the end of the function prototype declaration or definition, according to ANSI C's scope rules. It is not possible to declare an object of that type outside the function. This can be fixed by declaring the **struct**, **union**, or **enum** ahead of the function prototype or function definition and then referring to it just by its tag.

```
int f(struct s {int x;} st)
{}
```

"file", line 1: warning: dubious struct declaration; use tag only: s

Rewrite this as:

```
struct s {int x;};
int f(struct s st)
{}
```

dubious escape: **\c**

Type: Warning — *Options:* all

Only certain characters may follow **"\"** in string literals and character constants; **c** was not one of them. The **"\"** is ignored.

```
int i = '\q';
```

"file", line 1: warning: dubious escape: \q

dubious escape: \<*hex value*>

Type: Warning — *Options:* all

This message diagnoses the same condition as the preceding one, but the character that follows “\” in the program is a non-printing character. The *hex value* between the brackets in the diagnostic is the character’s code, printed as a hexadecimal number.

dubious reference to *type* typedef: *typedef*

Type: Warning — *Options:* all

This message is similar to “dubious tag in function prototype: *type tag*”. A function prototype declaration refers to a *type struct*, *union*, or *enum* typedef with name *typedef*. Because the *struct*, *union*, or *enum* has been declared within a function, it could not be in scope when the function whose prototype is being declared was defined. The prototype declaration and function definition thus could never match.

```
f(){
    struct s { int x; };
    typedef struct s ST;
    extern int g(ST, struct s);
}
```

"file", line 4: warning: dubious reference to struct typedef: ST

"file", line 4: warning: dubious tag in function prototype: struct s

dubious static function at block level

Type: Warning — *Options:* -Xc

A function was declared with storage class **static** at block scope. The ANSI C standard says that the behavior is undefined if a function is declared at block scope with an explicit storage class other than **extern**. Although functions may be declared this way, other implementations may not recognize them, or may attach a different meaning to such a declaration.

```
void
f(void){
    static void g(void);
}
```

"file", line 3: warning: dubious static function at block level

dubious tag declaration: *type tag*

Type: Warning — *Options:* all

A new **struct**, **union**, or **enum** *type* with tag *tag* was declared within a function prototype declaration or the parameter declaration list of an old-style function definition. Calls to the function would always produce a type mismatch, because the tag declaration goes out of scope at the end of the function declaration or definition, according to ANSI C’s scope rules. It is not possible to declare an object of that type outside the function.

```
int f(struct s *);
```

"file", line 1: warning: dubious tag declaration: struct s

dubious tag in function prototype: *type tag*

Type: Warning — Options: all

This message is similar to the previous one. A function prototype declaration refers to a **struct**, **union**, or **enum type** with tag *tag*. The *tag* has been declared within a function. Therefore it could not be in scope when the function whose prototype is being declared was defined. The prototype declaration and function definition thus could never match.

```
f(){
    struct s {int x;};
    int g(struct s *);
}
```

"file", line 3: warning: dubious tag in function prototype: struct s

duplicate case in switch: *value*

Type: Error — Options: all

There are two **case** statements in the current **switch** statement that have the same constant value *value*.

```
f(void){
    int i = 5;
    switch(i) {
        case 4:
        case 4:
            break;
    }
}
```

"file", line 5: duplicate case in switch: 4

duplicate default in switch

Type: Error — Options: all

There are two **default** labels in the current **switch** statement.

```
f(void){
    int i = 5;
    switch(i) {
        default:
        default:
            break;
    }
}
```

"file", line 5: duplicate "default" in switch

duplicate formal parameter: *name*

Type: Warning — *Options:* all

In a function-like macro definition, *name* was used more than once as a formal parameter.

```
#define add3(a,a,c) a + b + c
```

"*file*", line 1: warning: duplicate formal parameter: a

duplicate member name: *member*

Type: Error — *Options:* all

A **struct** or **union** declaration uses the name *member* for more than one member.

```
union u {
    int i;
    float i;
};
```

"*file*", line 3: duplicate member name: i

duplicate name in % line specification: *name*

Type: Error — *Options:* all

Formal parameter *name* was mentioned more than once in the "%" line of an enhanced **asm** function.

#elif follows **#else**

Type: Warning — *Options:* all

A preprocessing if-section must be in the order **#if**, optional **#elif**'s, followed by optional **#else** and **#endif**. The code contains a **#elif** after the **#else** directive.

```
#if defined(ONE)
    int i = 1;
#elif defined(TWO)
    int i = 2;
#else
    int i = 3;
#elif defined(FOUR)
    int i = 4;
#endif
```

"*file*", line 7: warning: **#elif** follows **#else**

#elif has no preceding #if

Type: Error — Options: all

An **#elif** directive must be part of a preprocessing if-section, which begins with a **#if** directive. The code in question lacked the **#if**.

```
#elif defined(TWO)
```

```
    int i = 2;
```

```
#endif
```

"file", line 1: #elif has no preceding #if

"file", line 3: #if-less #endif

#elif must be followed by a constant expression

Type: Error — Options: all

There was no expression following the **#elif** directive.

```
#if defined(ONE)
```

```
    int i = 1;
```

```
#elif
```

```
    int i = 4;
```

```
#endif
```

"file", line 3: warning: #elif must be followed by a constant expression

#else has no preceding #if

Type: Error — Options: all

An **#else** directive was encountered that was not part of a preprocessing if-section.

```
#else
```

```
    int i =7;
```

```
#endif
```

"file", line 1: #else has no preceding #if

"file", line 3: #if-less #endif

embedded NUL not permitted in asm()

Type: Error — Options: all

The string literal that appears in an old-style **asm()** contains an embedded NUL character (character code 0).

```
asm("this is an old-style asm with embedded NUL: \0");
```

"file", line 1: embedded NUL not permitted in asm()

empty `#assert` directive

Type: Error — *Options:* all

A **#assert** directive contained no predicate name to assert.

```
#assert
```

"file", line 1: empty `#assert` directive

empty character constant

Type: Error — *Options:* all

The program has a character constant without any characters in it.

```
int i = '';
```

"file", line 1: empty character constant

empty constant expression after macro expansion

Type: Error — *Options:* all

A **#if** or **#elif** directive contained an expression that, after macro expansion, consisted of no tokens.

```
#define EMPTY
```

```
#if EMPTY
```

```
    char *mesg = "EMPTY is non-empty";
```

```
#endif
```

"file", line 2: empty constant expression after macro expansion

empty `#define` directive line

Type: Error — *Options:* all

A **#define** directive lacked both the name of the macro to define and any other tokens.

```
#define
```

"file", line 1: empty `#define` directive line

empty file name

Type: Error — *Options:* all

The file name in a **#include** directive is null.

```
#include <>
```

"file", line 1: empty file name

empty header name

Type: Error — *Options:* all

This diagnostic is similar to the preceding one, but the null file name arises after macro substitution.

```
#define NULLNAME <>
#include NULLNAME
```

"*file*", line 2: empty header name

empty predicate argument

Type: Error — *Options:* all

The compiler expects to find tokens between the ()'s that delimit a predicate's assertions in a **#unassert** directive. None were present.

```
#unassert machine()
```

"*file*", line 1: empty predicate argument

empty translation unit

Type: Warning — *Options:* all

The source file has no tokens in it after preprocessing is complete. The ANSI C standard requires the compiler to diagnose a file that has no tokens in it.

```
#ifdef COMPILE
    int token;
#endif
```

"*file*", line 5: warning: empty translation unit

empty #unassert directive

Type: Error — *Options:* all

A **#unassert** contained no predicate name to discard.

```
#unassert
```

"*file*", line 1: empty #unassert directive

empty #undef directive, identifier expected

Type: Error — *Options:* all

A **#undef** directive lacked the name of a macro to "undefine."

```
#undef
```

"*file*", line 1: empty #undef directive, identifier expected

`{}`-enclosed initializer required

Type: Warning — *Options:* all

When initializing an aggregate, except in the case of initializing a character array with a string literal or an automatic structure with an expression, the initializes must be enclosed in `{}`'s.

```
int ia[5] = 1;
f(void){
    struct s { int x,y; } st = 1;
}
```

"file", line 1: warning: `{}`-enclosed initializer required

"file", line 3: warning: `{}`-enclosed initializer required

"file", line 3: struct/union-valued initializer required

end-of-loop code not reached

Type: Warning — *Options:* all

A loop was written in such a way that the code at the end of the loop that the compiler generates to branch back to the beginning of the loop is not reachable and will never be executed.

```
f(void){
    int i = 1;
    while (i) {
        return 4;
    }
}
```

"file", line 5: warning: end-of-loop code not reached

enum constants have different types: op **operator**

Type: Warning — *Options:* **-w3**

A relational operator **operator** was used to compare enumeration constants from two different enumeration types. This may indicate a programming error. Note also that the sense of the comparison is known at compile time, because the constants' values are known.

```
enum e1 { ec11, ec12 } ev1;
enum e2 { ec21, ec22 } ev2;
void v(void){
    if (ec11 > ec22)
        ;
}
```

"file", line 4: warning: enum constants have different types: op ">"

```
enum type mismatch: arg #n
```

Type: Warning — Options: -w3

The program is passing an enumeration constant or object to a function for which a prototype declaration is in scope. The passed argument is of a different enumerated type from the one in the function prototype, which may indicate a programming error.

```
enum e1 { ec11 } ev1;
enum e2 { ec21 } ev2;
void ef(enum e1);
```

```
void v(void){
    ef(ec21);
}
```

```
"file", line 6: warning: enum type mismatch: arg #1
```

```
enum type mismatch: op operator
```

Type: Warning — Options: -w3

This message is like the previous one. One of the operands of *operator* is an enumeration object or constant, and the other is an enumeration object or constant from a different enumerated type.

```
enum e1 { ec11, ec12 } ev1;
enum e2 { ec21, ec22 } ev2;
void v(void){
    if (ev1 > ec22)
        ;
}
```

```
"file", line 4: warning: enum type mismatch: op ">"
```

```
enumeration constant hides parameter: name
```

Type: Warning — Options: all

A declaration of an enumerated type within a function includes an enumeration constant with the same name as parameter *name*. The enumeration constant hides the parameter.

```
int
f(int i){
    enum e { l, k, j, i };
}
```

```
"file", line 3: warning: enumeration constant hides parameter: i
```

enumerator used in its own initializer: *name*

Type: Warning — *Options:* all

When setting the value of enumerator *name*, *name* was used in the expression. ANSI C's scope rules take *name* in the expression to be whatever symbol was in scope at the time.

```
int i;
f(void){
    enum e { i = i+1, j, k };    /* uses global i in i+1 */
}
```

"*file*", line 3: warning: enumerator used in its own initializer: i

"*file*", line 3: integral constant expression expected

enumerator value overflows INT_MAX (2147483647)

Type: Warning — *Options:* all

The value for an enumeration constant overflowed the maximum integer value.

```
enum e { e1=2147483647, e2 };    /* overflow for e2 */
```

"*file*", line 1: warning: enumerator value overflows INT_MAX
(2147483647)

#error: *tokens*

Type: Error — *Options:* all

A **#error** directive was encountered in the source file. The other *tokens* in the directive are printed as part of the message.

```
#define ONE 2
#if ONE != 1
#error ONE != 1
#endif
```

"*file*", line 3: #error: ONE != 1

%error encountered in asm function

Type: Error — *Options:* all

A **%error** specification line was encountered while an enhanced **asm** was being expanded.

error in asm; expect ; or \n, saw '*c*'

Type: Error — *Options:* all

In a "**%**" line of an enhanced **asm** function, the compiler expected to read a semi-colon or new-line and found character *c* instead.

error writing output file

Type: Error — *Options:* all

An output error occurred while the compiler attempted to write its output file or a temporary file. The most likely problem is that a file system is out of space.

```

) expected
  Type: Error — Options: all
  In an #unassert directive, the assertion of a predicate to be dropped must
  be enclosed in (.

  #unassert system(unix

  "file", line 1: ")" expected

( expected after # identifier
  Type: Error — Options: all
  When the "#" operator is used in a #if or #elif directive to select a predi-
  cate instead of a like-named macro, the predicate must be followed by a
  parenthesized list of tokens.

  #assert system(unix)
  #define system "unix"
  #if #system
      char *systype = system;
  #endif

  "file", line 3: "(" expected after "# identifier"

( expected after first identifier
  Type: Error — Options: all
  In an #unassert directive, the assertion of a predicate to be dropped must
  be enclosed in (.

  #unassert system unix

  "file", line 1: "(" expected after first identifier

extern and prior uses redeclared as static: name
  Type: Warning — Options: -Xc, -w3
  name was declared at file scope as an extern, then later the same object or
  function was declared as static. ANSI C rules require that the first declara-
  tion of an object or function give its actual storage class. The compilation
  system accepts the declaration and treats the object or function as if the
  first declaration had been static.

  extern int i;
  static int i;

  "file", line 2: warning: extern and prior uses redeclared as static: i

```


first operand must have scalar type: op ?:

Type: Error — *Options:* all

The conditional expression in a `? :` expression must have scalar (integral, floating-point, or pointer) type.

```
struct s { int x; } st;
f(void){
    int i = st ? 3 : 4;
}
```

"file", line 3: first operand must have scalar type: op "?:"

floating-point constant calculation out of range: op **operator**

Type: Warning, Error — *Options:* all

The compiler detected an overflow at compile time when it attempted the **operator** operation between two floating-point operands. The diagnostic is a warning if the expression is in executable code and an error otherwise.

```
double d1 = 1e300 * 1e300;
```

"file", line 1: floating-point constant calculation out of range: op "*"

floating-point constant folding causes exception

Type: Error — *Options:* all

This message is like the previous one, except that the operation caused a floating-point exception that causes the compiler to exit.

formal parameter lacks name: param #*n*

Type: Error — *Options:* all

In a function prototype definition, a name was not provided for the *n*-th parameter.

```
int f(int){
}
```

"file", line 1: formal parameter lacks name: param #1

function actually returns double: **name**

Type: Warning — *Options:* -w3

A function that was declared to return type **float** actually returns **double**. This information may be useful to know if you try to write an assembly language version of the called routine, or if you write the routine in C++.

```
float f();
```

"file", line 1: warning: function actually returns double: f

function cannot return function or array

Type: Error — Options: all

A function was declared whose return type would be a function or array, rather than a pointer to one of those.

```
int f(void)[]; /* function returning array of ints */
```

"file", line 1: function cannot return function or array

function designator is not of function type

Type: Error — Options: all

An expression was used in a function call as if it were the name of a function or a pointer to a function when it was not.

```
f(void){
    char *p;
    p();
}
```

"file", line 3: function designator is not of function type

function expects to return value: **name**

Type: Warning — Options: -w3

The current function was declared with a non **void** type, but contained a **return** statement with no return value expression.

```
f(void){
    return;
}
```

"file", line 2: warning: function expects to return value: f

function prototype parameters must have types

Type: Warning — Options: all

A function prototype declaration cannot contain an identifier list; it must declare types. The identifier list is ignored.

```
int f(i);
```

"file", line 1: warning: function prototype parameters must have types

identifier expected after #

Type: Error — Options: all

The compiler expected to find an identifier, a predicate name, after a **"#"** in a conditional compilation directive, and none was there.

```
#if #system(unix) || #
    char *os = "sys";
#endif
```

"file", line 1: identifier expected after **"#"**

identifier expected after #undef

Type: Error — *Options:* all

A **#undef** must be followed by the name of the macro to be undefined. The token following the directive was not an identifier.

```
#undef 4
```

"file", line 1: identifier expected after #undef

identifier or - expected after -A

Type: Error — *Options:* all

The **cc** command line argument **-A** must be followed by the name of a predicate to assert, or by a **"-"**, to eliminate all predefined macros and predicates. The token following **-A** was neither of these.

```
cc -A3b2 -c x.c
```

command line: identifier or "-" expected after -A

identifier or digit sequence expected after #

Type: Error — *Options:* all

An invalid token or non-decimal number follows the **"#"** that introduces a preprocessor directive line.

```
# 0x12
```

"file", line 1: identifier or digit sequence expected after "#"

identifier redeclared: *name*

Type: Warning, Error — *Options:* all

The identifier *name* was declared in a way that is inconsistent with a previous appearance of *name*, or *name* was declared twice in the same scope.

Previous releases were forgiving of inconsistent redeclarations if the types were "nearly" the same. ANSI C considers the types to be different. The **-Xt** option will allow you to retain the previous behavior, although the compiler will issue a warning. When the types are manifestly different, the diagnostic is always an error. The **-Xa** and **-Xc** options always produce an error when the types are different.

```
int x;
long x;
int y;
double y;
```

"file", line 2: warning: identifier redeclared: x

"file", line 4: identifier redeclared: y

Declarations of functions with and without argument information can often lead to confusing diagnostics. The following example illustrates.

```
int f(char);
int f();
```

"file", line 2: warning: identifier redeclared: f

According to ANSI C's type compatibility rules, a function declaration that lacks type information (i.e., one that is not a function prototype declaration) is compatible with a function prototype only when each parameter type is unchanged by the default argument promotion rules. In the example, **char** would be affected by the promotion rules (it would be promoted to **int**). Therefore the two declarations have incompatible types.

identifier redeclared; ANSI C requires static: *name*

Type: Warning — Options: all

name was declared twice at file scope. The first one used storage class **static**, but the second one specified no storage class. ANSI C's rules for storage classes require that all redeclarations of *name* after the first must specify **static**.

```
static int i;
int i;
```

"file", line 2: warning: identifier redeclared; ANSI C requires "static": i

identifier redefined: *name*

Type: Error — Options: all

name was defined more than once. An object with an initializer was declared more than once, or a function was defined more than once.

```
int i = 1;
int i = 1;
```

"file", line 2: identifier redefined: i

#if must be followed by a constant expression

Type: Warning — Options: all

No expression appeared after a **#if** directive.

```
#if
    int i = 4;
#endif
```

"file", line 1: warning: #if must be followed by a constant expression

#if on line *n* has no **#endif**

Type: Error — *Options:* all

The compiler reached end of file without finding the **#endif** that would end the preprocessing if-section that began with the **if** directive that was on line *n*. The **if** directive is one of **#if**, **#ifdef**, or **#ifndef**.

```
#ifdef NOENDIF
    int i = 1;
```

"*file*", line 5: **#ifdef** on line 1 has no matching **#endif**

"*file*", line 5: warning: empty translation unit

#if-less #endif

Type: Error — *Options:* all

An **#endif** directive was encountered that was not part of a preprocessing if-section.

```
    int i = 1;
#endif
```

"*file*", line 2: **#if-less #endif**

#ifdef must be followed by an identifier

Type: Warning — *Options:* all

A **#ifdef** preprocessing directive must be followed by the name of the macro to check for being defined. The source code omitted the identifier. The **#ifdef** is treated as if it were false.

```
#ifdef
    int i = 1;
#endif
```

"*file*", line 1: warning: **#ifdef** must be followed by an identifier

#ifndef must be followed by an identifier

Type: Warning — *Options:* all

The **#ifndef** directive must be followed by the identifier that is to be tested for having been defined.

```
#ifndef
    int i = 5;
#endif
```

"*file*", line 1: warning: **#ifndef** must be followed by an identifier

ignoring malformed #pragma comment (exestr, "string")

Type: Warning — *Options:* all

The compiler encountered a **#pragma comment** that did not have the form shown. The erroneous directive is ignored.

#pragma comment (exestr, not a string)

"file", line 1: warning: malformed #pragma comment (exestr, "string")

ignoring malformed #pragma int_to_unsigned symbol

Type: Warning — *Options:* all

The compiler encountered a **#pragma int_to_unsigned** directive that did not have the form shown. The erroneous directive is ignored.

#pragma int_to_unsigned strlen();

"file", line 1: warning: ignoring malformed #pragma
int_to_unsigned symbol

ignoring malformed #pragma pack(n)

Type: Warning — *Options:* all

The compiler encountered a **#pragma pack** directive that did not have the form shown. The erroneous directive is ignored.

ignoring malformed #pragma weak symbol [=value]

Type: Warning — *Options:* all

The compiler encountered a **#pragma weak** directive that did not have the form shown. The erroneous directive is ignored.

#pragma weak write,_write

"file", line 1: warning: ignoring malformed #pragma weak
symbol [=value]

implicitly declaring function to return int: *name*()

Type: Warning — *Options:* -w3

The program calls function *name*, which has not been previously declared. The compiler warns that it is assuming that function *name* returns **int**.

```
void v(void){
    g();
}
```

"file", line 2: warning: implicitly declaring function to return int: g()

improper cast of void expression

Type: Error — *Options:* all

A **void** expression cannot be cast to something other than **void**.

```
f(void){
    void v(void);
    int i = (int) v();
}
```

"file", line 3: improper cast of void expression

improper member use: *name*

Type: Warning, Error — *Options:* all

The program contains an expression with a “->” or “.” operator, and *name* is not a member of the structure or union that the left side of the operator refers to, but it is a member of some other structure or union. This diagnostic is an error if the member is not “unique.” A unique member is part of one or more structures or unions but has the same type and offset in all of them.

```
struct s1 { int x,y; };
struct s2 { int q,r; };
f(void){
    struct s1 *ps1;
    ps1->r = 3;
}
```

"file", line 5: warning: improper member use: r

improper pointer subtraction

Type: Warning, Error — *Options:* all

The operands of a subtraction are both pointers, but they point at different types. Only pointers of the same type that point to the same array may be subtracted.

The diagnostic is a warning if the pointers point to objects of the same size, and an error otherwise.

```
f(void){
    int *ip;
    char *cp;
    int i = ip - cp;
}
```

"file", line 4: improper pointer subtraction

improper pointer/integer combination: arg #*n*

Type: Warning — *Options:* all

At a function call for which there is a function prototype declaration in scope, the code is passing an integer where a pointer is expected, or vice versa.

```
int f(char *);
g(void){
    f(5);
}
```

"*file*", line 3: warning: improper pointer/integer combination: arg #1

improper pointer/integer combination: op *operator*

Type: Warning — *Options:* all

One of the operands of *operator* is a pointer and the other is an integer, but this combination is invalid.

```
f(void){
    int i = "abc";
    int j = i ? 4 : "def";
}
```

"*file*", line 2: warning: improper pointer/integer combination: op "="

"*file*", line 3: warning: improper pointer/integer combination: op ":"

"*file*", line 3: warning: improper pointer/integer combination: op "="

inappropriate qualifiers with void

Type: Warning — *Options:* all

When **void** stands by itself, it may not be qualified with **const** or **volatile**.

```
int f(const void);
```

"*file*", line 1: warning: inappropriate qualifiers with "void"

#include <... missing '>'>

Type: Warning — *Options:* all

In a **#include** directive for which the header name began with "<", the closing ">" character was omitted.

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

"*file*", line 1: warning: #include <... missing '>'>

#include directive missing file name

Type: Error — *Options:* all

A **#include** directive did not specify a file to include.

```
#include
```

"*file*", line 1: #include directive missing file name


```
#include of /usr/include/... may be non-portable
```

Type: Warning — *Options:* all

The source file included a file with the explicit prefix `"/usr/include"`. Such an inclusion is implementation-dependent and non-portable. On some systems the list of default places to look for a header might not include the `/usr/include` directory. In such a case the wrong file might be included.

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

"file", line 1: warning: #include of /usr/include/... may be non-portable

```
incomplete #define macro parameter list
```

Type: Error — *Options:* all

In the definition of a function-like parameter, the compiler did not find a `"")"` character on the same (logical) line as the `#define` directive.

```
#define mac(a
```

"file", line 1: incomplete #define macro parameter list

```
incomplete struct/union/enum tag: name
```

Type: Error — *Options:* all

An object *name*, with **struct**, **union**, or **enum** type and tag *tag*, was declared but the type is incomplete.

```
struct s st;
```

"file", line 1: incomplete struct/union/enum s: st

```
inconsistent redeclaration of extern: name
```

Type: Warning — *Options:* all

A function or object was redeclared *name* with storage class **extern** for which there was a previous declaration that has since gone out of scope. The second declaration has a type that conflicts with the first.

```
f(void){
    int *p = (int *) malloc(5*sizeof(int));
}
g(void){
    void *malloc();
}
```

"file", line 5: warning: inconsistent redeclaration of extern: malloc

inconsistent redeclaration of static: *name*

Type: Warning — *Options:* all

An object or function that was originally declared with storage class **static** was redeclared. The second declaration has a type that conflicts with the first.

The two most frequent conditions under which this diagnostic may be issued are:

1. A function was originally declared at other than file scope and with storage class **static**. The subsequent declaration of the function has a type that conflicts with the first.
2. A function or object was originally declared at file scope and with storage class **static**. A subsequent declaration of the same object or function at other than file scope used storage class **extern** (or possibly no storage class, if a function), and there was an intervening, unrelated, declaration of the same name.

```
f(void){
    static int myfunc(void);
}
g(void){
    static char *myfunc(void);
}
```

"file", line 5: warning: inconsistent redeclaration of static: myfunc

```
static int x;
f(void){
    int x;                      /* unrelated */
    {
        extern float x; /* related to first declaration */
    }
}
```

"file", line 5: warning: inconsistent redeclaration of static: x

inconsistent storage class for function: *name*

Type: Warning — *Options:* all

ANSI C requires that the first declaration of a function or object at file scope establish its storage class. Function *name* was redeclared in an inconsistent way according to these rules.

```
g(void){
    int f(void);
    static int f(void);
}
```

"file", line 3: warning: inconsistent storage class for function: f

initialization type mismatch

Type: Warning — *Options:* all

The type of an initializer value is incompatible with the type of the object being initialized. This specific message usually applies to pointers.

```
int a;
unsigned int *pa = &a ;
```

"file", line 2: warning: initialization type mismatch

initializer does not fit: *value*

Type: Warning — *Options:* -w3, -w2

The value *value* does not fit in the space provided for it. If it were fetched from that space, it would not reproduce the same value as was put in. In the message, *value* is represented as a hexadecimal value if the initializer is unsigned, decimal if it is signed.

```
struct s {signed int m1:3; unsigned int m2:3;} st = {4, 5};
unsigned char uc = 300u;
```

"file", line 1: warning: initializer does not fit: 4

"file", line 2: warning: initializer does not fit: 0x12c

integer overflow detected: op *operator*

Type: Warning — *Options:* all

The compiler attempted to compute the result of an *operator* expression at compile-time, and determined that the result would overflow. The low-order 32 bits of the result are retained, and the compiler issues this diagnostic.

```
int i = 1000000 * 1000000;
```

"file", line 1: warning: integer overflow detected: op "**"

integral constant expression expected

Type: Error — *Options:* all

The compiler expected (required) an integral constant or an expression that can be evaluated at compile time to yield an integral value. The expression written contained either a non-integral value, a reference to an object, or an operator that cannot be evaluated at compile time.

```
int ia[5.0];
```

"file", line 1: integral constant expression expected

integral constant too large

Type: Warning — *Options:* all

An integral constant is too large to fit in an **unsigned long**.

```
int i = 1234567890123;
```

"file", line 1: warning: integral constant too large

internal compiler error: **message**

Type: Error — *Options:* all

This message does not diagnose a user programming error (usually), but rather a problem with the compiler itself. One of the compiler's internal consistency checks has failed.

interpreted as a #line directive

Type: Warning — *Options:* -Xc

A source line was encountered that had a number where the directive name usually goes. Such a line is reserved for the compiler's internal use, but it must be diagnosed in the -Xc (strictly conforming) mode.

```
# 9
```

"file", line 1: warning: interpreted as a #line directive

"file", line 1: warning: directive is an upward-compatible
ANSI C extension

invalid cast expression

Type: Error — *Options:* all

A cast cannot be applied to the expression because the types are unsuitable for casting. Both the type of the expression being cast and the type of the cast must be scalar types. A pointer may only be cast to or from an integral type.

```
f(void){
    struct s {int x;} st;
    int i = (int) st;
}
```

"file", line 3: invalid cast expression

invalid class in asm % line: **class**

Type: Error — *Options:* all

The storage class **class** that the compiler encountered in an enhanced **asm** "% " line is not one of the acceptable classes.

invalid compiler control line in .i file

Type: Error — *Options:* all

A .i file, the result of a cc -P command, is assumed to be a reserved communication channel between the preprocessing phase and the compilation phase of the compiler. The .i file lets you examine that intermediate form to detect errors that may otherwise be hard to detect. However, the

compiler expects to find only a few directives that are used for internal communication. The source file that was compiled (a *.i* file) contained a preprocessing directive other than one of the special directives.

invalid directive

Type: Error — *Options:* all

The identifier that follows a “#” in a preprocessing directive line was one that the compiler did not recognize.

unknown

"file", line 1: invalid directive

invalid initializer

Type: Error — *Options:* all

The program contains an initializer for an **extern** or **static** that attempts to store a pointer in a smaller than pointer-sized object. Such initializations are not supported.

```
int j;
```

```
char c = (char) &j ;
```

"file", line 2: invalid initializer

invalid multibyte character

Type: Error — *Options:* all

A multibyte character in a string literal or character constant could not be converted to a single wide character in the host environment.

invalid source character: ‘‘c’’

Type: Error — *Options:* all

The compiler encountered a character (*c*) in the source program that is not a valid ANSI C token.

```
int i = 1$;
```

"file", line 1: invalid source character: ‘‘\$’’

invalid source character: <hex value>

Type: Error — *Options:* all

This message diagnoses the same condition as the previous one, but the invalid character is not printable. The *hex value* between the brackets in the diagnostic is the hexadecimal value of the character code.

invalid switch expression type

Type: Error — *Options:* all

The controlling expression of a switch statement could not be converted to **int**. This message always follows “switch expression must have integral type”.

```
f(){
    struct s {int x;} sx;
    switch(sx){
        case 4: ;
    }
}
```

"file", line 3: switch expression must have integral type

"file", line 3: invalid switch expression type

invalid token: **non-token**

Type: Error — Options: all

The compiler encountered a sequence of characters that does not comprise a valid token. An invalid token may result from the preprocessing "##" operator. The offending **non-token** is shown in the diagnostic. If the **non-token** is longer than 20 characters, the first 20 are printed, followed by ". . .", and the offending invalid token is ignored.

```
#define PASTE(l,r) l ## r
double d1 = 1e;
double d2 = PASTE(1,e);
int i = 1veryverylongnontoken;
```

"file", line 2: invalid token: 1e

"file", line 2: syntax error before or at: ;

"file", line 2: warning: syntax error: empty declaration

"file", line 3: invalid token: 1e

"file", line 3: syntax error before or at: ;

"file", line 3: warning: syntax error: empty declaration

"file", line 4: invalid token: 1veryverylongnontoke...

"file", line 4: syntax error before or at: ;

"file", line 4: warning: syntax error: empty declaration

invalid token in #define macro parameters: **token**

Type: Error — Options: all

The compiler encountered an inappropriate token while processing the argument list of a function-like macro definition. **token** is the erroneous token.

```
#define mac(a,4) a b c
```

"file", line 1: invalid token in #define macro parameters: 4

invalid token in directive

Type: Error — *Options:* all

The compiler found an invalid token at the end of what would otherwise be a correctly formed directive.

#line 7 "file.c

"*file*", line 1: warning: string literal expected after #line <number>

"*file*", line 1: invalid token in directive: "

"*file*", line 1: warning: tokens ignored at end of directive line

invalid type combination

Type: Error — *Options:* all

An inappropriate combination of type specifiers was used in a declaration.

short float f;

"*file*", line 1: invalid type combination

invalid type for bit-field: *name*

Type: Error — *Options:* all

The type chosen for bit-field *name* is not permitted for bit-fields. Bit-fields may only be declared with integral types.

struct s { float f:3; };

"*file*", line 1: invalid type for bit-field: f

invalid use of defined operator

Type: Error — *Options:* all

A **defined** operator in a **#if** or **#elif** directive must be followed by an identifier or ()'s that enclose an identifier. The source code did not use it that way.

```
#if defined
    int i = 1;
#endif
```

"*file*", line 1: invalid use of "defined" operator

invalid white space character in directive

Type: Warning — *Options:* all

The only white space characters that are permitted in preprocessing directives are space and horizontal tab. The source code included some other white space character, such as form feed or vertical tab. The compiler treats this character like a space.

label redefined: *name*

Type: Error — *Options:* all

The same label *name* has appeared more than once in the current function. (A label's scope is an entire function.)

```

f(void){
    int i;
    i = 1;
    if (i) {
L:
        while (i)
            g();
        goto L;
    }
L: ;
}

```

"file", line 10: label redefined: L

left operand must be modifiable lvalue: op **operator**

Type: Error — **Options:** all

The operand on the left side of **operator** must be a modifiable lvalue, but it wasn't.

```

f(void){
    int i = 1;
    +i -= 1;
}

```

"file", line 3: left operand must be modifiable lvalue: op **"-="**

left operand of -> must be pointer to struct/union

Type: Warning, Error — **Options:** all

The operand on the left side of a **"->"** operator must be a pointer to a structure or union, but it wasn't. The diagnostic is a warning if the operand is a pointer, an error otherwise.

```

struct s { int x; };
f(void){
    long *lp;
    lp->x = 1;
}

```

"file", line 4: warning: left operand of **"->"** must be pointer to struct/union

left operand of `.` must be lvalue in this context

Type: Warning — *Options:* all

The operand on the left side of a `“.”` operator is an expression that does not yield an lvalue. Usually this results from trying to change the return value of a function that returns a structure.

```
struct s { int ia[10]; };
struct s sf(void);
f(void){
    sf().ia[0] = 3;
}
```

“file”, line 4: warning: left operand of `“.”` must be lvalue in this context

left operand of `.` must be struct/union object

Type: Warning, Error — *Options:* all

The `“.”` operator is only supposed to be applied to structure or union objects. The diagnostic is an error if the operand to the left of `“.”` is an array, pointer, function call, enumeration constant or variable, or a register value that got allocated to a register; it is a warning otherwise.

```
f(void){
    struct s { short s; };
    int i;
    i.s = 4;
}
```

“file”, line 4: warning: left operand of `“.”` must be struct/union object

`()`-less function definition

Type: Error — *Options:* all

The declarator portion of a function definition must include parentheses. A function cannot be defined by writing a typedef name for a function type, followed by an identifier and the braces `{ }` that define a function.

```
typedef int F();
F f{ }
```

“file”, line 2: `()`-less function definition

loop not entered at top

Type: Warning — *Options:* all

The controlling expression at the beginning of a **for** or **while** loop cannot be reached by sequential flow of control from the statement before it.

```
f(void){
    int i;
    goto lab;
    for (i = 1; i > 0; --i) {
lab:;
        i=5;
    }
}
```

"file", line 4: warning: loop not entered at top

macro nesting too deep

Type: Fatal — *Options:* all

The source code defines a macro that causes the compiler to run out of memory during preprocessing.

macro recursion

Type: Warning — *Options:* **-Xt**

The source code calls a macro that calls itself, either directly or indirectly. ANSI C's semantics prevent further attempts to rescan the macro. Older C compilers would try to rescan the macro.

Because the rescanning rules are different for ANSI C and its predecessor, the compiler provides the old behavior in **-Xt** and in **-Xk** modes, which includes producing this diagnostic when macro recursion is detected.

```
#define a(x) b(x)
#define b(x) a(x)
a(3)
```

"file", line 3: warning: macro recursion

macro redefined: *name*

Type: Warning — *Options:* all

The source code redefined a macro. Previous releases allowed such redefinitions silently if both definitions were identical except for the order and spelling of formal parameters. ANSI C requires that, when a macro is redefined correctly, the definitions must be identical including the order and spelling of formal parameters. This diagnostic is produced under all options if the new macro definition disagrees with the old one. For strict conformance, it is also produced under the **-Xc** option when the macro definitions disagree only in the spelling of the formal parameters.

```
#define TIMES(a,b) a * b
#define TIMES(a,b) a - b
```

"file", line 2: warning: macro redefined: TIMES

macro replacement within a character constant

Type: Warning — *Options:* -Xt

Previous releases allowed the value of a formal parameter to be substituted in a character constant that is part of a macro definition. ANSI C does not permit such a use.

```
#define CTRL(x) ('x'&037)      /* form control character */
```

```
int ctrl_c = CTRL(c);
```

"file", line 1: warning: macro replacement within a character constant

The proper way to express this construct in ANSI C is the following:

```
#define CTRL(x) (x&037) /* form control character */
```

```
int ctrl_c = CTRL('c');
```

macro replacement within a string literal

Type: Warning — *Options:* -Xt

This message diagnoses a similar condition to the preceding one, except the substitution is being made into a string literal.

```
#define HELLO(name)    "hello, name"
```

```
char *hello_dave = HELLO(Dave);
```

"file", line 1: warning: macro replacement within a string literal

ANSI C provides a way to accomplish the same thing. The `"#" "string-ize"` operator turns the tokens of a macro argument into a string literal, and adjacent string literals are concatenated. The correct form is:

```
#define HELLO(name)    "hello, " #name
```

```
char *hello_dave = HELLO(Dave);
```

member cannot be function: *name*

Type: Error — *Options:* all

A function may not be a member of a structure or union, although a pointer to a function may. Member *name* was declared as a function.

```
struct s { int f(void); };
```

"file", line 1: member cannot be function: f

mismatched ? and :

Type: Error — *Options:* all

An expression in a **#if** or **#elif** directive contained a malformed **?:** expression.

```
#if defined(foo) ? 5
```

```
    int i;
```

```
#endif
```

"file", line 1: mismatched "?" and ":"

mismatched parentheses

Type: Error — *Options:* all

Parentheses were mismatched in a preprocessing conditional compilation directive.

```
#if ((1)
```

```
    int i = 1;
```

```
#endif
```

"file", line 1: mismatched parentheses

missing)

Type: Error — *Options:* all

In a test of a predicate that follows a **"#"** operator in a **#if** or **#elif** directive, the **"")"** that follows the assertion was missing.

```
#if # system(unix
```

```
    char *system = "unix";
```

```
#endif
```

"file", line 1: missing ")"

missing formal name in % line

Type: Error — *Options:* all

In an enhanced **asm** function, a **"%"** line specified a storage class, but not the formal parameter that has that storage class.

missing operand

Type: Error — *Options:* all

The constant expression of a preprocessing conditional compilation directive is malformed. An expected operand for some operator was missing.

```
#define EMPTY
```

```
#if EMPTY / 4
```

```
    int i = 1;
```

```
#endif
```

"file", line 2: missing operand

missing operator

Type: Error — *Options:* all

The constant expression of a preprocessing conditional compilation directive is malformed. An operator was expected but was not encountered.

```
#if 1 4
        int i = 1;
#endif
```

"*file*", line 1: missing operator

missing tokens between parentheses

Type: Error — *Options:* all

In a **#assert** directive, there are no assertions within the parentheses of the predicate.

```
#assert system()
```

"*file*", line 1: missing tokens between parentheses

modifying typedef with *modifier*; only qualifiers allowed

Type: Warning — *Options:* -Xt

ANSI C prohibits applying a type *modifier* to a **typedef** name. ANSI C only permits modifying a **typedef** with a type qualifier (**const**, **volatile**). However, for compatibility, the compiler accepts the declaration and treats it as did previous releases. Future releases will reject this declaration.

```
typedef int INT;
unsigned INT i;
```

"*file*", line 2: warning: modifying typedef with "unsigned";
only qualifiers allowed

modulus by zero

Type: Warning, Error — *Options:* all

The second operand of a "%" operator is zero. If the modulus operation is part of a **#if** or **#elif** directive, the result is taken to be zero. The diagnostic is a warning if the modulus is in executable code, an error otherwise.

```
#if 42 % 0
        int i = 1;
#endif
```

"*file*", line 1: warning: modulus by zero

more than one character honored in character constant: **constant**

Type: Warning — *Options:* all

A character constant has an integral value that derives from the character codes of the characters. If a character constant comprises more than one character, the encoding of the additional characters depends on the implementation. This warning alerts you that the encoding that the preprocessing phase uses for the character constant **constant** is different in this release of the C compiler from the one in previous releases, which only honored the first character. (The encoding for character constants you use in executable code is unchanged.)

```
#if 'ab' != ('b' * 256 + 'a')
#error unknown encoding
#endif
```

```
"file", line 1: warning: more than one character honored in
character constant: 'ab'
```

must be followed by formal identifier in #define

Type: Error — *Options:* all

The “string-ize” operator “#” must be followed by the name of a formal parameter in a function-like macro.

```
#define mac(a) # + a
```

```
"file", line 1: "#" must be followed by formal identifier in #define
```

must have type function-returning-unsigned: **name**

Type: Warning — *Options:* all

The **name** that is a part of a **#pragma int_to_unsigned** directive must be an identifier whose type is function-returning-unsigned.

```
extern int f(int);
#pragma int_to_unsigned f
```

```
"file", line 2: warning: must have type "function-returning-unsigned": f
```

name in asm % line is not a formal: **name**

Type: Error — *Options:* all

The identifier **name** that followed a storage class specifier in the “%” line of an enhanced **asm** function was not one of the formal parameters of the function.

nested asm calls not now supported

Type: Error — *Options:* all

The compiler does not now support calls to enhanced **asm** functions as part of the argument expression for another enhanced **asm** function.

newline in character constant

Type: Error — *Options:* all

A character constant was written that had no closing ' on the same line as the beginning '.

```
int i = 'a
;
```

"file", line 1: newline in character constant

newline in string literal

Type: Warning, Error — *Options:* all

A string literal was written that had no closing " on the same line as the beginning ". The diagnostic is a warning if the string literal is part of a preprocessing directive (and the compiler provides the missing ") and an error otherwise.

```
char *p = "abc
;
```

"file", line 1: newline in string literal

newline not last character in file

Type: Warning — *Options:* all

Every non-empty source file and header must consist of complete lines. This diagnostic warns that the last line of a file did not end with a newline.

no actual for asm formal: *name*

Type: Error — *Options:* all

An enhanced **asm** function was called with fewer arguments than there were parameters in the definition. Thus there was no actual argument for parameter *name*.

no closing > in #include <...

Type: Error — *Options:* all

A **#include** directive that used the "<>" form of header omitted the closing ">".

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

"file", line 1: warning: #include <... missing '>'

no file name after expansion

Type: Error — *Options:* all

The form of **#include** directive was used that permits macro expansion of its argument, but the resulting expansion left no tokens to be taken as a file name.

```
#define EMPTY
#include EMPTY
```

"file", line 2: no file name after expansion

no hex digits follow \x

Type: Warning — *Options:* -Xa, -Xc

The \x escape in character constants and string literals introduces a hexadecimal character escape. The \x must be followed by at least one hexadecimal digit.

```
char *cp = "\xz";
```

"file", line 1: warning: no hex digits follow \x

no macro replacement within a character constant

Type: Warning — *Options:* -Xa, -Xc

This message is the inverse of "macro replacement within a character constant". This informs you that the macro replacement that was done for -Xt or -Xk mode is not being done in -Xa or -Xc mode.

no macro replacement within a string literal

Type: Warning — *Options:* -Xa, -Xc

This message is the inverse of "macro replacement within a string literal". It informs you that the macro replacement that was done for -Xt mode is not being done in -Xa or -Xt mode.

no tokens after expansion

Type: Error — *Options:* all

After macro expansion was applied to the expression in a **#line** directive, there were no tokens left to be interpreted as a line number.

```
#define EMPTY
#line EMPTY
```

"file", line 2: no tokens after expansion

no tokens follow #pragma

Type: Warning — *Options:* -w3

The compiler encountered a **#pragma** directive that contained no other tokens.

#pragma

"file", line 1: warning: no tokens follow "#pragma"

no tokens following #assert name (

Type: Error — *Options:* all

A use of the **#assert** directive is malformed. The assertions and the ")" that should follow are missing.

#assert system(

"file", line 1: no tokens following "#assert name ("

no tokens in #line directive

Type: Error — *Options:* all

The rest of a **#line** directive was empty; the line number and optional file name were missing.

#line

"file", line 1: no tokens in #line directive

non-constant initializer: op *operator*

Type: Error — *Options:* all

The initializer for an **extern**, **static**, or array object must be a compile-time constant. The initializers for an automatic structure or union object, if enclosed in { }, must also be compile-time constants. *operator* is the operator whose operands could not be combined at compile time.

int j;

int k = j+1;

"file", line 2: non-constant initializer: op "+"

non-formal identifier follows # in #define

Type: Warning — *Options:* all

The identifier that follows a "#" operator in a macro definition must be a formal parameter of a function-like macro.

#define mac(a) "abc" # b

"file", line 1: non-formal identifier follows "#" in #define

non-integral case expression

Type: Error — *Options:* all

The operand of a **case** statement must be an integral constant.

```
f(void){
    int i = 1;
    switch (i) {
        case 5.0: ;
    }
}
```

"*file*", line 4: non-integral case expression

non-unique member requires struct/union: *name*

Type: Error — *Options:* all

The operand on the left side of a "." operator was not a structure, union, or a pointer to one, and member *name* was not unique among all structure and union members that you have declared. Use "." only with structures or unions. The member should belong to the structure or union corresponding to the left operand.

```
struct s1 { int x,y; };
struct s2 { int y,z; };
f(void){
    long *lp;
    lp.y = 1;
}
```

"*file*", line 5: non-unique member requires struct/union object: y

"*file*", line 5: left operand of "." must be struct/union object

non-unique member requires struct/union pointer: *name*

Type: Error — *Options:* all

This message diagnoses the same condition as the preceding one, but for the "->" operator.

null character in input

Type: Error — *Options:* all

The compiler encountered a null character (a character with a character code of zero).

null dimension: **name**

Type: Warning, Error — *Options:* all

A dimension of an array is null in a context where that is prohibited. The diagnostic is a warning if the offending dimension is outermost and an error otherwise.

```
int ia[4][];
struct s { int x, y[]; };
int i = sizeof(int []);
```

"file", line 1: null dimension: ia

"file", line 2: warning: null dimension: y

"file", line 3: warning: null dimension: sizeof()

number expected

Type: Error — *Options:* all

The compiler did not find a number where it expected to find one in a **#if** or **#elif** directive.

```
#if 1 +
        int i = 1;
#endif
```

"file", line 1: number expected

old-style declaration hides prototype declaration: **name**

Type: Warning — *Options:* -w3

The function **name** was declared in an inner scope. The outer declaration was a function prototype declaration, but the inner one lacks parameter information. By ANSI C's scoping rules, the parameter information is hidden and the automatic conversions of types that the prototype would have provided are suppressed.

```
extern double sin(double);
f(void){
    extern double sin();
    double d;
    d = sin(1); /* Note: no conversion to double! */
}
```

"file", line 3: warning: old-style declaration hides prototype declaration: sin

"file", line 5: warning: argument does not match remembered type: arg #1

only one storage class allowed

Type: Error — Options: all

More than one storage class was specified in a declaration.

```
f(void){
    register auto i;
}
```

"*file*", line 2: only one storage class allowed

only qualifiers allowed after *

Type: Error — Options: all

Only the **const** or **volatile** type qualifiers may be specified after a "*" in a declaration.

```
int * const p;
int * unsigned q;
```

"*file*", line 2: only qualifiers allowed after *

only register valid as formal parameter storage class

Type: Error — Options: all

A storage class specifier may be specified in a function prototype declaration, but only **register** is permitted.

```
int f(
    register int x,
    auto int y
);
```

"*file*", line 3: only "register" valid as formal parameter storage class

operand cannot have void type: op *operator*

Type: Error — Options: all

One of the operands of *operator* has **void** type.

```
f(void){
    void v(void);
    int i = v();
}
```

"*file*", line 3: operand cannot have void type: op "="

"*file*", line 3: assignment type mismatch

operand must be modifiable lvalue: op **operator**

Type: Error — Options: all

The operand of **operator** must be a modifiable lvalue, but it wasn't.

```
f(void){
    int i = --3;
}
```

"**file**", line 2: operand must be modifiable lvalue: op "--"

operand treated as unsigned: **constant**

Type: Warning — Options: -Xt

An operand used in a **#if** or **#elif** directive has a value greater than **LONG_MAX** (2147483647) but has no unsigned modifier suffix (u or U). Previous releases treated such **constants** as signed quantities which, because of their values, actually became negative. ANSI C treats such constants as unsigned long integers, which may affect their behavior in expressions. This diagnostic is a transition aid that informs you that the value is being treated differently from before.

```
#if 2147483648 > 0
    char *mesg = "ANSI C-style";
#endif
```

"**file**", line 1: warning: operand treated as unsigned:
2147483648

operands have incompatible pointer types: op **operator**

Type: Warning — Options: all

operator was applied to pointers to different types.

```
f(void){
    char *cp;
    int *ip;
    if (ip < cp)
        ;
}
```

"**file**", line 4: warning: operands have incompatible pointer types: op "<"

operands have incompatible types: op **operator**

Type: Error — Options: all

The types of the operands for **operand** are unsuitable for that type of operator.

```
f(void){
    char *cp;
    int *ip;
    void *vp = ip + cp;
}
```

"file", line 4: operands have incompatible types: op "+"

operands must have *category* type: op *operator*

Type: Error — Options: all

The operands for *operator* do not fall into the appropriate category for that operator. *category* may be **arithmetic**, **integral**, or **scalar**.

```
f(void){
    int ia[5];
    int *ip = ia/4;
}
```

"file", line 3: operands must have arithmetic type: op "/"

out of scope extern and prior uses redeclared as static: *name*

Type: Warning — Options: -Xc, -w3

name was declared as **extern** in a block that has gone out of scope. Then *name* was declared again, this time as **static**. The compiler treats the object or function as if it were **static**, and all references, including ones earlier in the source file, apply to the **static** version.

```
f(void){
    extern int i;
}
static int i;
```

"file", line 4: warning: out of scope extern and prior uses
redeclared as static: i

overflow in hex escape

Type: Warning — Options: all

In a hexadecimal escape (**\x**) in a character constant or string literal, the accumulated value for the escape grew too large. Only the low-order 32 bits of value are retained.

```
int i = '\xabcdefdc';
```

"file", line 1: warning: \x is ANSI C hex escape

"file", line 1: warning: overflow in hex escape

"file", line 1: warning: character escape does not fit in character

parameter mismatch: ***ndecl*** declared, ***ndef*** defined

Type: Warning — *Options:* all" A function prototype declaration and an old-style definition

of the function disagree in the number of parameters. The declaration had ***ndecl*** parameters, while the definition had ***ndef***.

```
int f(int);
int f(i,j)
int i,j;
{ }
```

"***file***", line 4: warning: parameter mismatch: 1 declared, 2 defined

parameter not in identifier list: ***name***

Type: Error — *Options:* all

Variable ***name*** appears in an old-style function definition's parameter declarations, but it does not appear in the parameter identifier list.

```
f(a,b)
int i;
{ }
```

"***file***", line 2: parameter not in identifier list: i

parameter redeclared: ***name***

Type: Error — *Options:* all

name was used more than once as the name for a parameter in a function definition.

```
int f(int i, int i) { }
int g(i,j)
int i;
int i;
{ }
```

"***file***", line 1: parameter redeclared: i

"***file***", line 4: parameter redeclared: i

preprocessing a .i file

Type: Warning — *Options:* all

The source file is a .i file, a file that has already been preprocessed, and the -E cc option was selected. The compiler will simply copy the input file to the standard output without further processing.

prototype mismatch: ***n1*** arg[s] passed, ***n2*** expected

Type: Error — *Options:* all

A function for which there is a function prototype declaration in scope, and the number of arguments in the call, ***n2***, did not match the number of parameters in the declaration, ***n1***.

```
int f(int);
g(void){
    f(1,2);
}
```

"file", line 3: prototype mismatch: 2 args passed, 1 expected

return value type mismatch

Type: Error — *Options:* all

A value from a function cannot be returned that cannot be converted to the return-type of the function.

```
f(void){
    struct s { int x; } st;
    return( st );
}
```

"file", line 3: return value type mismatch

semantics of **operator** change in ANSI C; use explicit cast

Type: Warning — *Options:* -w3, -w2

The type promotion rules for ANSI C are slightly different from those of previous releases. In the current release the default behavior is to duplicate the previous rules. In future releases the default will be to use ANSI C rules. To obtain the ANSI C interpretation, use the -Xa option for the cc command.

Previous type promotion rules were "unsigned-preserving." If one of the operands of an expression was of unsigned type, the operands were promoted to a common unsigned type before the operation was performed.

ANSI C uses "value-preserving" type promotion rules. An unsigned type is promoted to a signed type if all its values may be represented in the signed type.

The different type promotion rules may lead to different program behavior for the operators that are affected by the unsigned-ness of their operands:

- The division operators: /, /=, %, %=.
- The right shift operators: >>, >>=.
- The relational operators: <, <=, >, >=.

The warning message tells that the program contains an expression in which the behavior of *operator* will change in the future. To guarantee the desired behavior, insert an explicit cast in the expression.

```
f(void){
    unsigned char uc;
    int i;
    /* was unsigned divide, signed in ANSI C */
    i /= uc;
}
```

"file", line 5: warning: semantics of "/"= change in ANSI C; use explicit cast

To get the same behavior as in previous releases add an explicit cast:

```
f(void){
    unsigned char uc;
    int i;
    /* was unsigned divide, signed in ANSI C */
    i /= (unsigned int) uc;
}
```

shift count negative or too big: *op n*

Type: Warning — Options: all

The compiler determined that the shift count (the right operand) for shift operator *op* is either negative or bigger than the size of the operand being shifted.

```
f(){
    short s;
    s <<= 25;
}
```

"file", line 3: warning: shift count negative or too big: <<= 25

statement not reached

Type: Warning — Options: all

This statement in the program cannot be reached because of **goto**, **break**, **continue**, or **return** statements preceding it.

```
f(void){
    int i;
    return i;
    i = 4;
}
```

"file", line 4: warning: statement not reached

static function called but not defined: **name**()

Type: Warning — Options: all

The program calls function **name**, which has been declared **static**, but no definition of **name** appears in the translation unit. (The line number that is displayed in the message is one more than the number of lines in the file, because this condition can be diagnosed only after the entire translation unit has been seen.)

```
static int statfunc(int);
void
f(){
    int i = statfunc(4);
}
```

"file", line 7: warning: static function called but not defined: statfunc()

static redeclares external: **name**

Type: Warning — Options: all

name was reused as the name of a **static** object or function after having used it in the same block as the name of an **extern** object or function. The version of **name** that remains visible is the **static** version.

```
f(void){
    extern int i;
    static int i;
}
```

"file", line 3: warning: static redeclares external: i

storage class after type is obsolescent

Type: Warning — Options: -w3

According to the ANSI C standard, writing declarations in which the storage class specifier is not first is "obsolescent."

```
int static i;
```

"file", line 1: warning: storage class after type is obsolescent

storage class for function must be static or extern

Type: Warning — Options: all

An inappropriate storage class specifier for a function declaration or definition was used. Only **extern** and **static** may be used, or the storage class may be omitted. The specifier is ignored.

```
f(void){
    auto g(void);
}
```

"file", line 2: warning: storage class for function must be static or extern

string literal expected after # <number>

Type: Warning — *Options:* all

The “#” line information directive takes an optional second token, a file name. If present, it must be in the form of a string literal.

```
# 1 x.c
```

```
"file", line 1: warning: string literal expected after # <number>
```

```
"file", line 1: warning: tokens ignored at end of directive line
```

string literal expected after #file

Type: Error — *Options:* all

The **#file** directive (which is reserved for the compilation system) is used for internal communication between preprocessing and compilation phases. A string literal operand is expected as the operand.

string literal expected after #ident

Type: Error — *Options:* all

A **#ident** directive must be followed by a normal (not wide character) string literal.

```
#ident no-string
```

```
"file", line 1: string literal expected after #ident
```

string literal expected after #line <number>

Type: Warning — *Options:* all

This diagnostic is similar to “string literal expected after # <number>”, except that it applies to the standard **#line** directive.

string literal must be sole array initializer

Type: Warning — *Options:* all

It is not permissible to initialize a character array with both a string literal and other values in the same initialization.

```
char ca[] = { "abc", 'd' };
```

```
"file", line 1: warning: string literal must be sole array initializer
```

struct/union has no named members

Type: Warning — *Options:* all

A structure or union was declared in which none of the members is named.

```
struct s { int :4; char :0; };
```

```
"file", line 1: warning: struct/union has no named members
```

struct/union-valued initializer required

Type: Error — *Options:* all

ANSI C allows you to initialize an automatic structure or union, but the initializer must have the same type as the object being initialized.

```
f(void){
    int i;
    struct s { int x; } st = i;
}
```

"file", line 3: warning: {}-enclosed initializer required

"file", line 3: struct/union-valued initializer required

switch expression must have integral type

Type: Warning, Error — *Options:* all

A **switch** statement was written in which the controlling expression did not have integral type. The message is a warning if the invalid type is a floating-point type and an error otherwise. A floating-point **switch** expression is converted to **int**.

```
f(void){
    float x;
    switch (x) {
        case 4: ;
    }
}
```

"file", line 3: warning: switch expression must have integral type

syntax error before or at: **token**

Type: Error — *Options:* all

This is an all-purpose diagnostic that means you have juxtaposed two (or more) language tokens inappropriately. The compiler shows you the **token** at which the error was detected.

```
f(void){
    int i = 3+;
}
```

"file", line 2: syntax error before or at: ;

syntax error in macro parameters

Type: Error — *Options:* all

The macro parameter list part of a function-like macro definition is malformed. The list must be a comma-separated list of identifiers and was not.

```
#define mac(a,b,) a b
```

"file", line 1: syntax error in macro parameters

syntax error, probably missing `,` `;` or `=`

Type: Error — *Options:* all

A declaration that looked like a function definition was written, except that the type of the symbol declared was not “function returning.” A `“;”` or `“=”` is needed.

```
int i
int j;
```

file, line 2: syntax error, probably missing `“,”`, `“;”` or `“=”`

file, line 2: parameter not in identifier list: `j`

file, line 4: syntax error before or at: `<EOF>`

syntax violation: empty declaration

Type: Warning — *Options:* all

A null statement was written at file scope. This looks like an empty declaration statement and was previously permitted, but ANSI C does not.

```
int i;;
```

file, line 1: warning: syntax violation: empty declaration

syntax violation: `&... invalid`

Type: Warning — *Options:* **`-Xc`**

A `&...` was written in a program that was compiled with the **`-Xc`** option. `&...` is invalid ANSI C syntax. Do not use this notation explicitly.

syntax requires `;` after last struct/union member

Type: Warning — *Options:* all

The `“;”` that C syntax requires after the last structure or union member in a structure or union declaration was omitted.

```
struct s { int x };
```

file, line 1: warning: syntax requires `“;”` after last struct/union member

(*type*) tag redeclared: *name*

Type: Error — *Options:* all

The tag *name* that was originally a *type* (**struct**, **union**, or **enum**) tag was redeclared.

```
struct q { int m1, m2 };
enum q { e1, e2 };
```

file, line 2: (struct) tag redeclared: `q`

token not allowed in directive: *token*

Type: Error — *Options:* all

A *token* was used in a **#if** or **#elif** directive that is neither a valid operator for constant expressions, nor a valid integer constant.

```
#if 1 > "1"
    int i = 1;
#endif
```

"file", line 1: token not allowed in directive: "1"

token-less macro argument

Type: Warning — *Options:* **-Xc**

The actual argument to a preprocessor macro consisted of no tokens. The ANSI C standard regards this condition as undefined. The compiler treats the empty list of tokens as an empty argument, and, under the **-Xc** mode, it also issues this warning.

```
#define m(x) x+3
int i = m();
```

"file", line 2: warning: token-less macro argument

tokens after **-A-** are ignored

Type: Warning — *Options:* all

In the **-A-** option to the **cc** command, there were additional tokens adjacent to the option. They are ignored.

```
cc -A-extra -c x.c
```

command line: warning: tokens after **-A-** are ignored

tokens expected after **#** identifier (

Type: Error — *Options:* all

When the **"#"** operator is used in a **#if** or **#elif** directive to select a predicate instead of a like-named macro, the predicate must be followed by a parenthesized list of tokens.

```
#if #system(
    char *system = "unix";
#endif
```

"file", line 1: tokens expected after **"#** identifier (

tokens expected after (

Type: Error — *Options:* all

In a **#unassert** directive, the assertion(s) and closing ")" after the predicate were missing.

```
#unassert system(
```

"file", line 1: tokens expected after "("

tokens expected between parentheses

Type: Error — *Options:* all

The name of an assertion of a predicate to test was omitted in an **#if** or **#elif** directive.

```
#if #system()  
    char *sysname = "??";  
#endif
```

"file", line 1: tokens expected between parentheses

tokens ignored after -U{identifier}

Type: Warning — *Options:* all

In the command line **-U** option, there were tokens following the name of the macro to be undefined.

```
cc -Uunix,u3b2 -c x.c
```

command line: warning: tokens ignored after "-U{identifier}"

tokens ignored at end of directive line

Type: Warning — *Options:* all

A directive line contains extra tokens that are not expected as part of the directive.

```
#undef a b          /* can only undefine one */
```

"file", line 1: warning: tokens ignored at end of directive line

too many array initializers

Type: Error — *Options:* all

More initializers than the array can hold were provided for the array.

```
int ia[3] = { 1, 2, 3, 4 };
```

"file", line 1: too many array initializers

too many #else's

Type: Warning — *Options:* all

The code contained more than one **#else** directive in a preprocessing if-section. All **#else** directives after the first are taken to be false.

```
#ifdef ONE
    int i = 1;
#else
    int i = 2;
#else
    int i = 3;
#endif
```

"file", line 5: warning: too many #else's

too many errors

Type: Fatal — *Options:* all

The compiler encountered too many errors to make further processing sensible. Rather than produce further diagnostics, the compiler exits.

too many initializers for scalar

Type: Error — *Options:* all

A {}-bracketed initialization for a scalar contains more than one value.

```
int i = { 1, 2 };
```

"file", line 1: too many initializers for scalar

too many struct/union initializers

Type: Error — *Options:* all

Too many initializers for a structure or union were provided.

```
struct s { int x,y; } st = { 1,2,3 };
```

"file", line 1: too many struct/union initializers

trailing , prohibited in enum declaration

Type: Warning — *Options:* **-Xc, -w3**

An extra comma was supplied at the end of an enumeration type declaration. The extra comma is prohibited by the syntax.

```
enum e { e1, e2, };
```

"file", line 1: warning: trailing "," prohibited in enum declaration

trigraph sequence replaced

Type: Warning — *Options:* **-Xt**

ANSI C introduces the notion of trigraphs, three-character sequences that stand for a single character. All such sequences begin with “??”. Because sequences that are interpreted as trigraphs may appear in existing code, the compiler produces a transitional diagnostic when such sequences are encountered.

```
char *surprise = "this is a trigraph??!";
```

"file", line 1: warning: trigraph sequence replaced

type does not match prototype: *name*

Type: Warning — *Options:* all

A function prototype declaration for a function was provided, but it used an old-style definition. The type for parameter *name* in that definition is incompatible with the type you used in the prototype declaration.

```
int f(char *);
int f(p)
int *p;
{}
```

"file", line 4: warning: type does not match prototype: p

The following example shows an especially confusing instance of this diagnostic.

```
int f(char);
int f(c)
char c;
{}
```

"file", line 3: warning: identifier redeclared: f

"file", line 4: warning: type does not match prototype: c

f has an old-style definition. For compatibility reasons, **f**'s arguments must therefore be promoted according to the default argument promotions, which is how they were promoted before the existence of function prototypes. Therefore the value that must actually be passed to **f** is an **int**, although the function will only use the **char** part of the value. The diagnostic, then, identifies the conflict between the **int** that the function expects and the **char** that the function prototype would (conceptually) cause to be passed.

There are two ways to fix the conflict:

1. Change the function prototype to read

```
int f(int);
```

2. Define **f** with a function prototype definition:

```
int f(char);
int f(char c)
{ }
```

typedef already qualified with *qualifier*

Type: Warning — *Options:* all

A type specifier includes a **typedef** and an explicit type qualifier, *qualifier*. The **typedef** already included *qualifier* when it was declared.

```
typedef volatile int VOL;
volatile VOL v;
```

"file", line 2: warning: typedef already qualified with "volatile"

typedef declares no type name

Type: Warning — *Options:* all

In a declaration with storage class **typedef**, no type name was actually declared. This is probably a programming error.

```
typedef struct s { int x; };
```

"file", line 1: warning: typedef declares no type name

typedef redeclared: *name*

Type: Warning — *Options:* all

typedef *name* was declared more than once. The later declaration has an identical type to the first.

```
typedef int i;
typedef int i;
```

"file", line 2: warning: typedef redeclared: i

typedef redeclares external: *name*

Type: Warning — *Options:* all

typedef *name* was declared, but there is an **extern** of the same name in the same block. The **typedef** hides the external.

```
f(void){
    extern int INT;
    typedef int INT;
}
```

"file", line 3: warning: typedef redeclares external: INT

typedef valid only for function declaration

Type: Warning — *Options:* all

A function definition may not have the **typedef** storage class. It is ignored here.

```
typedef int f(void){}
```

"*file*", line 1: warning: "typedef" valid only for function declaration

unacceptable operand for unary &

Type: Error — *Options:* all

An attempt was made to take the address of something whose address cannot be taken.

```
f(void){
    int *ip = &g() ;
}
```

"*file*", line 2: unacceptable operand for unary &

#unassert requires an identifier token

Type: Error — *Options:* all

The **#unassert** directive must name a predicate to "un-assert."

```
#unassert 5
```

"*file*", line 1: #unassert requires an identifier token

undefined label: *label*

Type: Error — *Options:* all

A **goto** was written in the current function, but the target *label* was never defined anywhere within the function.

```
f(void){
    goto L;
}
```

"*file*", line 3: undefined label: L

undefined struct/union member: *name*

Type: Error — *Options:* all

The program made reference to a structure or union member, *name*, that has not been declared as part of any structure.

```
struct s { int x; };
f(void){
    struct s q;
    q.y = 1;
}
```

"*file*", line 4: undefined struct/union member: y

undefined symbol: *name*

Type: Error — *Options:* all

A symbol *name* was referred to for which there is no declaration in scope.

```
f(void){
    g(i);
}
```

"*file*", line 2: undefined symbol: i

undefining __STDC__

Type: Warning — *Options:* -Xt

ANSI C prohibits undefining the predefined symbol __STDC__. However, C issue 5 permits you to do so in transition mode (only). Use this feature to test C code that has been written to work in both an ANSI C and non-ANSI C environment.

For example, suppose you have C code that checks __STDC__, declaring function prototype declarations if it is defined, and old-style function declarations (or definitions) if not. Because the compiler predefines __STDC__, you would ordinarily be unable to check the old-style code, and you would have to run the code through another (non-ANSI C) compiler. By undefining __STDC__ (usually on the command line), you can use the compiler to do the checking.

This diagnostic tells you, as required, that you are violating ANSI C constraints.

```
#undef __STDC__ /* usually -U__STDC__ on cc line */
```

```
#ifdef __STDC__
int
myfunc(const char *arg1, int arg2)
#else
/* non-ANSI C case */
int
myfunc(arg1,arg2)
char *arg1,
/* oops */
int arg2;
#endif
{
}
```

"*file*", line 1: warning: undefining __STDC__

"*file*", line 10: syntax error before or at: int

"*file*", line 12: syntax error before or at: {

```
unexpected (
  Type: Error — Options: all
  A misplaced "(" was encountered in a #if or #elif directive.
```

```
#if 1 (
    int i = 1;
#endif

"file", line 1: unexpected "("
```

```
unexpected )
  Type: Error — Options: all
  A misplaced ")" was encountered in a #if or #elif directive.
```

```
#if ) 1
    int i = 1;
#endif

"file", line 1: unexpected ")"
```

```
unexpected character in asm % line: 'c'
  Type: Error — Options: all
  In the "%" specification line of an enhanced asm function, the compiler
  expected to see an alphabetic character that begins a storage class
  specifier. Instead it encountered the character c.
```

```
unknown comment type: type
  Type: Warning — Options: all
  The compiler encountered a #pragma comment directive with an
  unknown type. The erroneous directive is ignored.
```

```
#pragma comment (unknown, "text")

"file", line 1: warning: unknown comment type: unknown
```

```
unknown operand size: op operator
  Type: Error — Options: all
  operator "++", "--", or "=" was applied to an operand whose size is
  unknown. The operand is usually a pointer to a structure or union whose
  members have not been declared.
```

```
f(void){
    struct s *sp;
    sp++;
}

"file", line 3: unknown operand size: op "++"
```

unnamed **type** member

Type: Warning — *Options:* all

In the **type** declaration, you failed to give a member a name.

```
union s { int; char c; };
```

"*file*", line 1: warning: unnamed union member

unreachable case label: **value**

Type: Warning — *Options:* all

The expression specified in a **case** statement has a value outside the range of the type of the controlling expression of the enclosing **switch** statement. Therefore the case label can never be reached. In the message, **value** is represented as a hexadecimal value if the case expression is unsigned, decimal if it is signed.

```
f(){
    unsigned char uc;

    switch( uc ){
        case 256:
            ;
    }
}
```

"*file*", line 5: warning: unreachable case label: 256

unrecognized #pragma ignored: **pragma**

Type: Warning — *Options:* -w3

Because **#pragma** directives are implementation-specific, when the -w3 compilation flag is set, the compiler warns about any such directives that it is ignoring. The compiler does not recognize **#pragma pragma**.

```
#pragma list
```

"*file*", line 1: warning: unrecognized #pragma ignored: list

unused label: **label**

Type: Warning — *Options:* -w3

When using **acomp** (a component name for the ANSI C compiler) if the -w3 compilation flag is set, the compiler warns if a label is declared but never used in a function. When removing labels from the symbol table, it checks the SY_REF flag and warns if it is unset.

"*file*", line 4: warning: unused label: label

use double instead of long float

Type: Warning — *Options:* all

An object or function was declared to be **long float**, which was a synonym for **double**. ANSI C does not permit **long float**, although the compiler accepts it as a transition aid.

```
long float f = 1.0;
```

"file", line 1: warning: use "double" instead of "long float"

useless declaration

Type: Warning — *Options:* all

ANSI C requires that every declaration actually declare something, such as

- a declarator,
- a structure or union tag,
- structure or union members, or
- enumeration constants.

A declaration was written that provided no information to the compiler.

```
int;           /* no identifier */
enum e { e1, e2 }; /* introduces enum e */
enum e;       /* no new information */
```

"file", line 1: warning: useless declaration

"file", line 3: warning: useless declaration

using out of scope declaration: *name*

Type: Warning — *Options:* all

name was previously declared in a scope that is no longer active. In some ANSI C implementations, referring to such an object would yield an error; calling such a function would be interpreted as calling a function returning **int**. The compiler remembers the previous declaration and uses it. This warning tells what the compiler has done.

```
f(void){
    extern int i;
    double sin(double);
}
g(void){
    double d = sin(1.5);
    i = 1;
}
```

"file", line 6: warning: using out of scope declaration: sin

"file", line 7: warning: using out of scope declaration: i

void expressions may not be arguments: arg #*n*

Type: Error — *Options:* all

A function call contains an argument for which the expression type is **void**.

```
f(void){
    void v(void);
    g(v());
}
```

"*file*", line 3: void expressions may not be arguments: arg #1

void function cannot return value

Type: Warning — *Options:* all

A **return** was written statement with an expression, but the declared type of the function is **void**.

```
void v(void){
    return 3;
}
```

"*file*", line 2: void function cannot return value

void must be sole parameter

Type: Error — *Options:* all

Only the first parameter in a function prototype declaration may have **void** type, and it must be the only parameter.

```
int f(int,void);
```

"*file*", line 1: "void" must be sole parameter

void parameter cannot have name: *name*

Type: Error — *Options:* all

The parameter *name* was declared in a function prototype declaration that has **void** type.

```
int f(void v);
```

"*file*", line 1: void parameter cannot have name: v

\x is ANSI C hex escape

Type: Warning — *Options:* -Xt

In earlier releases, "\x" was equivalent to "x". However, in ANSI C, "\x" introduces a hexadecimal character escape. This diagnostic warns of the new meaning.

If valid hexadecimal characters follow “\x”, they are interpreted as part of the new escape sequence. Otherwise “\x” is treated as it was in previous releases.

```
int i = '\x';
```

"file", line 1: warning: \x is ANSI C hex escape

zero or negative subscript

Type: Warning, Error — *Options:* all

The size in an array declaration is zero or negative. The diagnostic is a warning if the size is zero and an error otherwise.

```
int ia[-5];  
int ib[0];
```

"file", line 1: zero or negative subscript

"file", line 2: warning: zero or negative subscript

zero-sized struct/union

Type: Error — *Options:* all

A structure or union was declared with the size of zero.

```
struct s { int ia[0]; };
```

"file", line 1: warning: zero or negative subscript

"file", line 1: zero-sized struct/union

Operator names

This section lists internal operator names that the compiler may use in error messages with definitions of these names.

,OP	The C “comma operator” (as distinct from the “,” that is used to separate function arguments).
ARG	A function argument, or a value passed to a function.
AUTO	An automatic variable that has not been allocated to a register.
CALL	A function call with arguments.
CBRANCH	A conditional branch. (This may be part of an if or loop statement.)
CONV	A conversion. It may have been explicit, in the form of a cast, or implicit, in the semantics of a C statement.
FCON	A floating-point constant.
ICON	An integer or address constant.
NAME	An object or function with extern or static storage class.
PARAM	A function parameter, a value that is received by a function.
REG	An object that has been allocated to a register.
RETURN	The operation that corresponds to a return statement.
STAR	The indirection operator “*”, as in *p.
STRING	A string literal.
U&	The “take address of” operator (as distinct from the bit-wise AND operation).
U-	The arithmetic negation operator (as distinct from subtraction).
UCALL	A function call with no arguments.
UGE	An unsigned “>=” comparison.
UGT	An unsigned “>” comparison.
ULE	An unsigned “<=” comparison.
ULT	An unsigned “<” comparison.
UPLUS	The ANSI C “unary +” operator.

Other error messages

The following messages may appear at compile time, but they are not generated by the compiler. Messages beginning with `Assembler:` are produced by `as`. Messages beginning with `ld:` are generated by `ld`, the link editor. Note that the format of the messages varies, and some of the messages are displayed over several lines.

```
Assembler: file.c
```

```
  aline n (cline n) : trouble writing; probably out of temp-file space
```

The file system may be low on space, or the temporary file or output file exceeded the current ulimit.

```
Assembler: file.c aline n (cline n)
```

```
  Cannot open Output File filename
```

The directory containing the source file is unwritable, or the file system containing source file is mounted read-only.

```
ld: Symbol name in file2.o is multiply defined. First defined in file1.o
```

A symbol name was defined more than once.

```
undefined                first referenced
symbol                   in file
```

```
sym1                    file1.o
```

```
ld fatal: Symbol referencing errors. No output written to a.out
```

A referenced symbol was not found. Compilation terminates.

Symbols, numbers

- + addition operator, 111
- & address operator, 109
- %= assignment operator, 115
- &= assignment operator, 115
- += assignment operator, 115
- = assignment operator, 115
- /= assignment operator, 115
- <<= assignment operator, 115
- = assignment operator, 115
- *= assignment operator, 115
- ^=assignment operator, 115
- |= assignment operator, 115
- & bitwise AND operator, 113
- ^bitwise exclusive OR operator, 113
- | bitwise inclusive OR operator, 113
- , comma operator, 115
- ? : conditional operator, 115
- decrement operator, 109
- / division operator, 110
- ... ellipsis notation, 104
- = = equality operator, 113
- \$ in yacc rules, 402
- ++ increment operator, 109
- * indirection operator, 109
- != inequality operator, 113
- << left shift operator, 112
- <= less or equal operator, 112
- < less than operator, 112
- && logical AND operator, 114
- ! logical negation operator, 109
- || logical OR operator, 114
- * multiplication operator, 110
- negation operator, 109
- ~one's complement operator, 109
- # preprocessing operator, 93
- ## preprocessing operator, 93
- % remainder operator, 110
- . structure member operator
- subtraction operator, 111
- + unary plus operator, 109

A

- abbreviations used in lex, 383
- aborted program debugging, sdb, 241
- absolute
 - address, sdb, 245
 - debugger. *See* adb
- actions
 - associated with yacc rules, 400, 402
 - default in yacc specification, 403
 - executed by parser, 397
 - in the middle of yacc rules, 404
- active functions, dbxtra, 209, 211
- adb
 - ? and / command display, 263
 - = command display, 262
 - backtrace, 269
 - breakpoints, 266
 - combining commands on a single line, 278
 - computing number, 280
 - core image file, 256
 - creating scripts, 278
 - current address, 259
 - current line change, 246
 - data formats, 261
 - deleting breakpoints, 268
 - displaying instructions and data, 257
 - exiting, 256
 - forming addresses, 257
 - forming expressions, 257
 - introduced, 255
 - killing a program, 268
 - memory maps, 275
 - operators, 260
 - options, 255, 265, 283
 - patching binaries, 283
 - program execution, 265
 - program preparation, 255
 - register names, 259
 - setting default input format, 279
 - single-stepping a program, 268
 - starting, 255
 - symbols, 257
 - using UNIX commands, 280
 - validating addresses, 277

address

- adb (*continued*)
 - variable value, 258
 - variables, 258
- address
 - binding, ld(CP), 127
 - physical, ld(CP), 126
 - validation, adb, 277
- admin command, SCCS, 342, 348, 349, 363, 364
- alias
 - command, dbxtra, 210
 - dbxtra and dbXtra, 233
- allocation
 - algorithm in ld(CP), 145
 - virtual address space, ld(CP), 126
- ambiguous grammar rules for yacc, 419
- ANSI implementation-defined behavior
 - arrays and pointers, 492
 - characters, 488
 - declarators, 494
 - environment, 486
 - floating point, 491
 - identifiers, 487
 - integers, 490
 - library functions, 498
 - locale-specific behavior, 508
 - preprocessing directives, 495
 - qualifiers, 494
 - registers, 492
 - statements, 495
 - structures, unions, enumerations, and bit-fields, 493
 - translation, 486
- a.out file
 - building, 445
 - compatibility, 473
 - differences with shared and non-shared libraries, 447
 - identification, 450
 - importing symbols, 464
 - incompatible libraries, 475
 - l option, 445
 - non-shared libraries, 447
 - referencing symbols, 445
 - shared libraries, 443, 444, 447, 451, 452, 458, 460, 468, 482
- a.out(FP), 37, 39
- appending to yytext with yymore, 389
- arbitrary value types in yacc specifications, 407
- archive libraries, 45
 - compatibility, 471
 - creating, 47
 - implementation, 34, 53
 - in make(CP), 326
 - linking with, 44, 45, 61
 - make(CP), 323
 - non-shared libraries, 444
 - searching, ld(CP), 143
 - shared library, 443
- ar(CP), 47
- argc and argv, 79, 80
- arithmetic
 - conversions, 84, 107
 - expression parsing, 421
 - types, 99
- arrays
 - declaration, 104
 - implementation-defined behavior, 492
 - initialization, 117
 - sdb, 244
- as(CP), 39
- asm, 87
- assembler, 34, 39
- assembly language
 - program debugger. *See* dbxtra
 - shared and non-shared libraries, 450
- #assert, 97
- assign command, dbXtra, 227
- assignment statement
 - ld(CP), 129, 139
 - source code, ld(CP), 129
- associativity rules for yacc grammar, 421
- auditing, SCCS, 373
- auto, 102

B

- backslash (\), 89
- backtrace display, adb, 269
- backward command, dbxtra, 210, 231
- basic types, 99
- binding address, ld(CP), 127
- bindkey command, dbXtra, 201, 202
- bit-fields, 100, 156
- bit-fields, implementation-defined behavior, 493
- blanks in yacc specifications, 399

branch
 libraries, 452
 SCCS, 345
 table, in shared libraries, 452, 453, 459
 #branch in shared libraries, 461
 break statement, 122
 breakpoint
 dbxtra, 214
 delete command, adb, 268
 delete command, sdb, 247
 display command, adb, 267
 display command, dbXtra, 197, 215
 event-ids, dbXtra, 206
 list command, sdb, 247
 management, sdb, 247
 sdb, 241
 set command, adb, 266
 set command, dbXtra, 207
 set command, dbxtra, 213
 set command, sdb, 247
 status, dbXtra, 197
 temporary, dbXtra, 223
 browse capabilities, dbXtra, 199
 browser. *See* cscope
 .bss, ld(CP), 127

C

C++
 debugging, overloaded functions and operators, 236
 debugging, usage requirements, 235
 debugging.cc command, 237
 debugging,specific features, 235
 execution capabilities, dbXtra, 199
 interaction mode, dbXtra, 194
 C code
 created from lex specifications, 394
 translating lex specifications into C, 395
 C language
 arrays, 492
 bit-fields, 493
 characters, 488
 comments, 91
 compilation modes and dependencies, 83, 87, 88, 98, 106, 112
 compiler, 83
 constants, 87
 conversions, 84, 106
 declarations, 99

C language (*continued*)
 declarations in yacc specifications, 407
 declarators, 494
 definitions, 105
 enumerations, 493
 environment, 486
 escape sequences, 89
 execution capabilities, dbXtra, 199
 expressions, 108
 floating point, 491
 identifiers, 86, 487
 implementation-defined behavior. *See* ANSI implementation-defined behavior
 initialization, 116
 integers, 490
 interaction mode, dbXtra, 195
 keywords, 87
 library functions, 498
 locale-specific behavior, 508
 operators, 93, 109
 phases of translation, 85
 pointers, 492
 portability, 123
 preprocessing, 91
 preprocessing directives, 495
 punctuators, 90
 qualifiers, 494
 registers, 492
 scope, 101
 shared and non-shared libraries, 450
 source level debugger. *See* dbxtra
 statements, 119, 495
 storage duration, 102
 string literals, 90
 structures, 493
 tokens, 86
 translation, 486
 types, 99, 103
 unions, 493
 C libraries
 linking with, 44
 partial contents, 67
 standard I/O, 76
 C Programmer's Productivity Tools (CPPT), 285
 call
 command, dbXtra, 215
 stack, dbxtra, 211
 stack movement, dbXtra, 225
 catch command, dbXtra, 221
 cb(CP), 13

cc command, C++ debugging, 237

C++/C toggle, dbXtra, 237

C/C++ toggle, dbXtra, 237

cc(CP)

- compilation modes and dependencies, 83, 87, 88, 98, 106, 112

- creating dynamically linked libraries, 47, 54, 59, 60

- D option, 467

- debugging option, 41

- dynamic linking options, 62

- header search option, 40

- I option, 466

- introduced, 33

- library linking option, 45, 61

- library search option, 49, 62

- optimizing option, 42

- profiling options, 42

- program naming option, 37

- static linking options, 49

cdc command, SCCS, 349, 367

cflow(CP), 472

char, 99, 106

character

- classification routines, 70

- constants, 89

- conversion routines, 71

- implementation-defined behavior, 488

checksum, SCCS, 372

child processes, debugging, 238

chkshlib(CP), 443, 473

chmod, SCCS, 348

classification routines, character, 70

.cnt files, 286

COFF link editor. *See* ld(CP)

comb command, SCCS, 349, 369

command language

- expressions, ld(CP), 127, 128

- symbols, ld(CP), 138

command line

- adb, 257, 258

- arguments, adb, 265

- arguments, sdb, 242

- options, dbXtra, 194

- sdb, 242

- syntax, dbXtra, 194

commands

- curses, dbxtra, 215

- execution, dbXtra, 216

- Motif, dbXtra, 215

- SCCS, 349

commands (*continued*)

- syntax, dbXtra, 201

- window, dbXtra, 196

comments, 38, 91

- in make(CP), 319

- in yacc specifications, 399

compile options

- adb, 255

- sdb, 241

compiler

- introduced, 19

- source code, 19

- writing with lex, 375

compiler diagnostics, 515

- error defined, 516

- error messages, 515

- fatal error defined, 516

- listed, 517

- message types, 516

- operator names in, 516, 599

- options, 516

- warning defined, 516

- warning messages, 515

conditional compilation, 95, 310

configurable

- keyboard interface, dbXtra, 198

- resources, dbXtra, 197

configured memory, ld(CP), 126

const, 57, 99, 103

constant expressions, 116

constants, 87

cont command, dbXtra, 222

context search, sdb, 245

context sensitivity in lex, 386

continuation lines, in make(CP), 319

continue statement, 122

conversion routines, characters, 71

conversions, 84, 106

coprocessors, sdb, 250

COPY section in ld(CP), 148

core

- dump examined, sdb, 242

- files, adb, 255

- files, dbXtra, 194

- files, dbxtra, 208

- files, sdb, 242

- files, shared libraries, 450

- image file, adb, 256

- image file, sdb, 241

- location, sdb, 245

coverage analysis with lprof, 285

CPPT

(C Programmer's Productivity Tools), 285

introduced, 285

lprof. *See* lprofCPPT cscope. *See* cscope, 300

CPU register, adb, 259, 269

cross-reference symbol table for cscope, 304

cscope

browser choice, 302

cross-reference symbol table, 300, 304

cscope.out cross-reference table, 305

editing, 300, 301, 310

environment variables, 301, 302, 303

escape to the shell, 313

examples of use, 311

introduced, 285

locating error message source, 305

setting terminal type, 301

stacking editor calls, 310

terminal information (terminfo)

database, 301

use, 302

curses

debugger. *See* dbxtra

display window, dbXtra, 226

interface introduced, dbXtra, 198

cxref, introduced, 17

D

.data

ld(CP), 127

shared libraries, 447

data segment, 34, 53, 55, 263

dbXtra

breakpoint display command, 197

breakpoint set command, 207

C++ interaction mode, 194

C interaction mode, 195

command line options, 194

compile options, 194

configurable resources, 197

core file, 194

cross reference, 227

curses interface, 198

debugging child processes, 238

edit commands, 201

help menu, 230

interface, 196

keyboard interface, 201

dbXtra (*continued*)

menu, 196

Motif interface, 196

object file, 194

options, 194

options menu, 228

program preparation, 194

sidebar command buttons, 197

source file, 195

starting, 194, 222

using, 215

windows, 199

dbxtra, 450

bindkey commands, 202

breakpoint, 214

compile options, 208

edit commands, 202, 207

help, 210

introduced, 22, 193

key binding, 207

program preparation, 194, 208

screen, 209

screen interface, 199

setting breakpoint, 213

source, 232

source code, 210

starting, 194, 208

stopping, 214

tutorial, 208

variables, 210

dbXtra(CP), 41

dbxtra(CP), 41

debugger

absolute. *See* adb; curses version. dbxtra

dbxtra, 193

Motif version. *See* dbXtra

shared libraries, 450

symbolic. *See* sdb

yacc, 409

debugging

child processes, 238

machine language, sdb, 250

running process, 239

declarations, 99

array, 104

declarators, implementation-defined

behavior, 494

default action in yacc specification, 403

#define, 34, 63, 93

- definitions
 - dbXtra, 195
 - function, 105
 - lex, 377
- definitions in lex, 382, 383
- delete breakpoint
 - adb, 268
 - interactively, sdb, 248
 - sdb, 247, 248, 250
- delimiters, in yacc, 400, 407
- delta
 - comments, SCCS, 364
 - numbering, SCCS, 345
- delta command, SCCS, 344, 349, 360, 361
- delta file, SCCS, 342, 344
- descriptive text, SCCS, 372
- directives, preprocessing, 34
- directories, sdb, 242
- disambiguating rules
 - lex, 385, 388
 - yacc, 420, 424
- disk storage space, 443
- display
 - C stack backtrace, adb, 269
 - command =, adb, 262
 - command /, adb, 263
 - command ?, adb, 263
 - command, dbXtra, 200, 226
 - instructions and data, sdb, 257
 - machine language statements, sdb, 250
 - memory maps, adb, 275
 - source command, sdb, 246
 - text, adb, 280
- displaying CPU registers, adb, 269
- double, 99, 107
- do-while statement, 121
- DSECT dummy section, ld(CP), 147
- dummy section, DSECT option, ld(CP), 147
- dump command, dbXtra, 228
- dump(CP), 450
- dynamic
 - analysis, SCCS, 285
 - dependency parameters, make(CP), 331
- dynamic linked libraries, options, 35
- dynamically linked libraries
 - creating, 47, 54
 - data segment, 53, 55
 - ELF, 35
 - guidelines for building, 54
 - implementation, 53
 - linking with, 45, 61

- dynamically linked libraries (*continued*)
 - text segment, 53, 55

E

- ECHO macro in lex, 381
- edit
 - command, dbxtra, 207
 - commands, dbXtra, 201, 202
- editor
 - calls stacked with cscope, 310
 - choice for cscope, 301
- EDITOR environment variable for cscope, 301
- #elif, 95
- ellipsis notation, 104
- #else, 95
- else statement, 120
- empty rules in yacc specification, 401
- #endif, 95
- end-marker in yacc specifications, 412
- entry point, changing, ld(CP), 142
- enumeration (enum), 101
- enumerations, implementation-defined
 - behavior, 493
- environment, implementation-defined
 - behavior, 486
- #error, 98
- error
 - handling in yacc, 424, 426
 - recovery, interactive in yacc, 426
 - recovery, YYERROR() function in yacc, 416
 - reporting in programs tracked with sdb, 241, 242
 - reporting in yacc, 409
 - token in yacc, 424
- error messages
 - adb, 256, 277
 - compiler, 515, 516
 - compiler fatal error, 516
 - cscope, 301
 - dbxtra, 219, 221, 232
 - help, SCCS, 349, 350
 - ld(CP), 134
 - locating source with cscope, 305
 - profiled program, lprof, 295
 - SCCS, 350, 352, 353, 356
 - sdb, 241, 242
 - yacc, 408, 409

- escape sequences, 89
- escape sequences in yacc specifications, 400
- event-id, dbxtra, 212
- events menu, dbXtra, 215
- executable files
 - adb, 255, 256
 - compiling, 39
 - creating, 37, 39
 - dbxtra, 208, 225
 - sdb, 242
- execute command
 - adb, 265, 268, 280
 - dbXtra, 195, 215
 - dbxtra, 215
 - sdb, 248, 249, 251
- execute menu, dbXtra, 222
- executing, program, adb, 265
- exit command
 - adb, 256
 - dbXtra, 215
 - dbxtra, 214
 - sdb, 251
- exit(S), 78
- #export, 458, 459
- expressions, 108
 - adb, 257, 259
 - dbXtra, 200, 215, 226, 227, 233, 234, 235, 236
 - dbxtra, 224
 - sdb, 246
- extern, 102
- external variable display, adb, 269

F

- fatal system error examined, adb, 256
- fclose(S), 78
- file, interdependencies. *See* make(CP)
- FILE structure, 77
- files
 - arguments in SCCS, 348
 - creating, in SCCS, 342
 - creating in SCCS, 363
 - delta file in SCCS, 344
 - end-of-file processing in lex, 391
 - g-file in SCCS, 343
 - menu, dbXtra, 215
 - nonrelocatable input file, ld(CP), 148
 - object, ld(CP), 127
 - output blocking, ld(CP), 148

- files (*continued*)
 - parameters in SCCS, 364
 - preparation for dbxtra, 208
 - protection in SCCS, 371
 - relocatable, 44
 - retrieving SCCS file with get, 343
 - SCCS, 371
 - delta file, 342
 - source file, shared libraries, 476
- find command, SCCS, 348
- .fini, ld(CP), 127
- finite-state machine, 413
- flags, SCCS, 372
- float, 99, 107
- floating point
 - constants, 88
 - implementation-defined behavior, 491
- floating types
 - conversion, 107
 - declaration, 99
- fopen(S), 77, 78
- for statement, 121
- fork(S), 289
- format
 - adb, 261, 262, 263, 264, 279
 - dbXtra, 235
 - SCCS, 372
 - specifier, sdb, 244
 - yacc specifications, 398
- forward command, dbxtra, 210, 231
- fpregs/regs command, dbXtra, 228
- frequency profiling with lprof, 285, 286
- fullscreen interface, dbxtra, 199
- func command, dbXtra, 225
- main, 105
- functions
 - declaration, 104
 - definition, 105
 - prototypes, 65, 102, 104, 105
 - prototypes, lint(CP), 155, 161

G

- g compile option, sdb, 241
- generic pointer, 104
- get command
 - options, SCCS, 343, 352, 353, 354, 355, 359
 - SCCS, 343, 344, 349, 350
- getc(S), 78
- getgrent(S) and shared library, 459

getopt(C)

- `getopt(C)`, 79
- `getopt(S)`, 81
- `g-file`, SCCS, 343
- global data, shared libraries, 459, 462, 480
- global symbols, 60
- `goto` statement, 122
- grammar rules
 - in `yacc`, 397, 400, 401
 - left associativity, `yacc`, 419
 - right associativity, `yacc`, 419
- `GROUP` directive, `ld(CP)`, 136
- grouping, output sections, `ld(CP)`, 135

H

- hardware register display, `dbXtra`, 228
- header files
 - including, 40, 94
 - lint, 159
 - standard place, 40
 - `string.h`, 65
- help
 - command, SCCS, 349, 350
 - `dbxtra`, 210
 - menu, `dbXtra`, 230
- hexadecimal escape, 89
- `#hide`, 458, 459, 461
- holes
 - creating within a section, `ld(CP)`, 140
 - in physical memory, `ld(CP)`, 141
- host shared library, 443, 451, 457, 468, 473, 474

I

- ID keywords, SCCS, 351
- `#ident`, 97
- identifiers
 - C language, 86
 - implementation-defined behavior, 487
 - qualification, `dbXtra`, 227
 - resolution, `dbXtra`, 233
- `#if`, 34, 95
- `if` statement, 119
- `#ifdef`, 95
- `ifile`, `ld(CP)`, 128
- `#ifndef`, 95
- `ignore` command, `dbXtra`, 221
- implicit rules, in `make(CP)`, 324
- imported symbol, 468

- `#include`, 34, 40, 63, 94
- include files, in `make(CP)`, 331
- incomplete types, 104
- incremental link editing, `ld(CP)`, 146
- info menu, `dbXtra`, 226
- INFO section in `ld(CP)`, 148
- `#init`, 461
- `.init`, initialized variable, `ld(CP)`, 127
- initialization
 - arrays, 117
 - C language, 116
- initialization and modification of file parameters, SCCS, 364
- initialized variable, `ld(CP)`, 127
- input
 - format, `adb`, 279
 - section, `ld(CP)`, 126
 - style for `yacc` specifications, 427
- input/output, 67, 76
- inserting commentary for the initial delta, SCCS, 364
- `int`, 99, 106
- integer, implementation-defined behavior, 490
- integral constants, 88
- integral types
 - conversion, 106
 - declaration, 99
 - initialization, 116
- interaction capabilities, `dbXtra`, 199
- interactive
 - debugging, `sdb`, 241
 - display, `sdb`, 241
- interactive error recovery in `yacc`, 426
- interface commands, `dbxtra`, 231
- internal rules of `make(CP)`, 335
- interpreting profiling output, 290
- interrupting program execution, `adb`, 267
- I/O routines for `lex`. *See* `lex`

J

- `j` flag, SCCS, 353
- `jump` command, `dbXtra`, 223

K

key binding, dbxtra, 207
 key letters that affect output, SCCS, 359
 keyboard
 interface, dbXtra, 198, 201
 translation, dbxtra, 231
 keyname/sequence editing command,
 dbxtra, 207
 keys command, dbxtra, 231
 keywords, 87

L

ld, shared libraries, 452
 LD_BIND_NOW, 53
 ld(CP), 39
 ADDR pseudo-function, 130, 134
 address, 126
 ALIGN pseudo-function, 130, 135
 allocation, 126, 145
 archive libraries, 143
 assignment statement, 129, 139
 binding address, 127
 .bss, 127
 C language operators in command lan-
 guage, 128
 changing entry point, 142
 command language expressions, 127, 128
 configured memory, 126
 constants in command language, 128
 COPY, 148
 .data, 127
 DSECT dummy section, 147
 dummy section, 147
 expressions, 128
 .fini, 127
 global symbols in command language,
 128
 GROUP directive, 136
 grouping output sections, 135
 holes, 137, 141
 ifile, 128
 incremental link editing, 146
 INFO, 148
 .init, ld(CP), 127
 initialized section holes, 140
 input section, 126
 introduced, 125
 -l option, 143
 location counter, 130

ld(CP) (*continued*)
 memory configuration, 126, 127, 131
 MEMORY directives, 131
 NEXT pseudo-function, 134
 NOLOAD, 148
 nonrelocatable input file, 148
 output, 126, 148
 OVERLAY, 148
 -r option, 146
 section, 126, 140
 SECTIONS directive, 132, 133, 134
 SECTIONS output alignment, 135
 SIZEOF pseudo-function, 130, 134
 source code and assignment statement
 interaction, 129
 symbol creating and defining, 138
 syntax for input directives, 149
 .text, 127
 -u option, 145
 unconfigured memory, 126
 ldd(CP), 52
 LD_LIBRARY_PATH, 50
 LD_RUN_PATH, 51
 left associativity rules for yacc grammar,
 421
 left recursion rules for yacc, 428
 length specifier, sdb, 244
 level numbering, delta numbering in SCCS,
 345
 lex
 actions, 380, 381
 compiling and linking, 395
 context sensitivity, 386, 387
 definitions, 377, 382, 383
 disambiguating rules, 385, 388
 end-of-file processing with yywrap, 391
 introduced, 375
 I/O routines, 388
 library default main() program, 395
 options, 395
 output file, 395
 pattern in specification, 377
 pattern searching, 377
 regular expressions, 377
 REJECT, 390
 reprocessing input with yymore, 389
 rules, 377
 running lexical analyzer program, 395
 specifications, 375, 377
 subroutines, 384
 symbol table, 393

lex (*continued*)

- tokens, 377, 391
 - UNIX systems, 394
 - used with make(CP), 396
 - used with yacc, 399, 410
 - user subroutines, 377
 - y.tab.h file, 393
 - yylen variable, 381
 - yyless, 390
 - yylex() lexical analyzer, 391
 - yytext character array, 381
 - yyval variable, 393
- lexical
- analysis, 375
 - analysis, yacc, 397, 410
 - analyzer yylex() generated by lex, 391
 - tie-ins, yacc, 429
- lex.yy.c file, 395
- .lib section, 35, 443
- libc, 465
- libmalloc, 465
- libraries
- archive, 34, 45
 - archive. *See* archive libraries
 - creating, 47, 54
 - dynamic. *See* dynamically linked libraries
 - dynamically linked, 43
 - header files, 63
 - host, 457, 468
 - lex, 395
 - libc, 44, 67, 76, 465
 - libgen, 75
 - libm, 46, 72
 - libmalloc, 465
 - linking with, 43, 61
 - lint(CP), 160
 - naming conventions, 61
 - networking, 444, 445
 - non-shared, 444
 - shared. *See* shared libraries
 - specifying on cc(CP) line, 19
 - standard place, 45
 - target, 457
 - yacc access, 409
- library functions, implementation-defined behavior, 498
- #line, 96
- line count data, 286

line mode

- dbXtra, 195
 - dbxtra, 231
- line number
- dbXtra, 194
 - dbxtra, 208
 - sdb, 241, 247
- line-by-line debugging, sdb, 241
- lines, newlines in yacc specifications, 399
- link editing, 39, 43
- dynamic, 34, 43
 - library linking options, 45, 61
 - multiply defined symbols, 60
 - quick reference, 61
 - shared libraries, 443
 - static, 34, 43
 - undefined symbols, 43
- link editor. *See* ld(CP)
- linking
- dynamic, 43
 - shared libraries, 452
 - static, 43
 - static vs. dynamic, 43
- lint(CP)
- command line, 159
 - consistency checks, 155
 - filters, 161
 - introduced, 153
 - introduced, 15
 - libraries, 160
 - message formats, 154
 - messages, 165
 - options and directives, 153, 162
 - portability checks, 156
 - suspicious constructs, 158
- literals in yacc specifications, 400
- locate-specific behavior,
- implementation-defined behavior, 508
- locating values in a file, adb, 283
- location
- counter, ld(CP), 130
 - display, sdb, 245
- long double, 99, 107
- long int, 99, 107
- look-ahead token, 413, 426
- lprof
- .cnt file, 286
 - coverage analysis, 285
 - creating profiled program, 286
 - data files in separate directory, 289
 - dynamic analysis, 285

lprof (continued)

- frequency profiling, 285, 286
- incompatibility with optimization, 296
- interpreting output, 290
- introduced, 286
- line count data, 286
- merging data files, 288
- non-terminating programs, 296
- profiled source listing, 290
- PROFOPTS environment variable, 286
- programs that fork(S), 289
- run-time data, 286
- specifying program and data file, 293
- summary option, 295
- test coverage improvement, 298
- timestamp override, 295
- unexecuted lines, 292
- used with program generators, 297
- used with shared libraries, 297

lprof(CP), 42

ls command, SCCS, 348

lvalues, 108

M

machine language

- debugging, *sdb*, 250
- statement display, *sdb*, 250

machine level

- debugger. *See* *dbxtra*
- instruction, *sdb*, 247

macro

- definitions, *make*(CP), 320
- ECHO in *lex*, 381
- expansion, 93
- translations, *make*(CP), 323

main() routine in *yacc*, 408

make(CP)

- archive libraries, 323, 326
- comments, 319
- continuation lines, 319
- creating new suffix rules, 330
- dependency, 319
- dynamic dependency parameters, 331
- environment variables, 332
- example *makefile*, 316, 334
- file interdependencies, 315, 316, 331
- implicit rules, 324
- Include files, 331
- internal rules, 335

make(CP) (continued)

- introduction, 315
- macro definitions, 320
- macro translations, 323
- maintaining parser code, 413
- makefile*, 315, 316, 319
- MAKEFLAGS, 332
- null suffix, 329
- recursive *makefiles*, 318
- suffix rules, 316, 323, 324, 329, 330
- Tilde (~) in SCCS filenames, 328
- timestamp information, 316
- used with *lex*, 396
- used with *yacc*, 413
- using shell commands, 322

makefile. *See* *make*(CP)

MAKEFLAGS. *See* *make*(CP)

malloc(S), 465

math library

- linking with, 46
- partial contents, 72

maximum offset, *adb*, 279

memory

- address, *adb*, 257
- configuration, *ld*(CP), 126, 127, 131
- map, *adb*, 275, 276, 277

MEMORY directives, *ld*(CP), 131

merging, *cscope*, 288, 295

messages

- lint(CP), 154, 165
- warning, *dbxtra*, 194
- warning, SCCS, 352

mkshlib(CP), 443, 454, 457, 462, 469, 481, 483

more command, *dbxtra*, 231

Motif interface, *dbXtra*, 196

Motif version debugger. *See* *dbXtra*

multibyte characters, 90

multi-line action in *lex*, 381

multiple function definitions and *cscope*, 310

multiply defined symbols, 60

multi-statement action in *lex*, 381

N

name, *yacc* specifications, 399

newlines in *yacc* specifications, 399

next command, *dbXtra*, 223

nexti command, *dbXtra*, 224

nm(CP)

nm(CP), 58
noload, 469
NOLOAD section in *ld(CP)*, 148
non-ASCII data file examined, *adb*, 255
non-shared libraries, 444
non-terminating profiled programs, 296
null suffix, in *make(CP)*, 329

O

#object, 461
object files
 data segment, 34, 53, 55
 dbXtra, 194
 dbxtra, 208
 ELF format, 45
 ld(CP), 127
 relocatable files, 44
 section, *ld(CP)*, 126
 text segment, 34, 53, 55, 57
objfile, *dbXtra*, 195
octal escape, 89
operators
 adb, 260
 in expressions, *dbXtra*, 234
operators (C language), 109
 additive, 111
 assignment, 115
 associativity and precedence, 116
 bitwise, 112, 113
 cast, 110
 comma, 115
 conditional, 115
 equality, 112
 logical, 114
 multiplicative, 110
 preprocessing, 93
 relational, 112
 structure
 unary, 109
optimizer, 42
optimizing, incompatibility with profiling, 296
options
 adb, 265, 283
 dbXtra, 194
 dbxtra, 194
 menu, *dbXtra*, 228
 sdb, 241, 242, 247

output
 dbXtra, 199
 file, blocking, 148
 section, holes, *ld(CP)*, 137
 section, *ld(CP)*, 126
 stopping, *dbxtra*, 210
 translations in *make(CP)*, 323
 width, *adb*, 279
OVERLAY section, *ld(CP)*, 148
overloaded functions and operators, C++
 debugging, 236

P

pagination, *dbxtra*, 231
paging
 dbxtra, 210
 dynamically linked libraries, 54, 58
parse trees in yacc rules, 406
parser
 arithmetic expressions, 421
 compiling and running, 413
 error recovery, 416
 operation, 414
 reset with *yverrok*, 426
patching binaries, *adb*, 283
path, *dbXtra*, 195
pattern in lex specification. *See* *lex*
pattern matching, *sdb*, 243
performance improvement with *prof* and *lprof*, 297
p-file, SCCS, 344
physical address, *ld(CP)*, 126
pointer
 declaration, 103
 implementation-defined behavior, 492
pointerr, initialization, 116
pointers, *sdb*, 244
portability
 C language, 123
 lint(CP) checks for, 156
 system calls, 71
position-independent code, 54
#pragma, 61, 97
precedence rules for yacc grammar, 421
preparing specifications for yacc, 427
preprocessing, 91
 directives, 34, 40, 92
 output, 38
 predefined names, 98

preprocessing (*continued*)
 tokens, 92
 preprocessing directives,
 implementation-defined behavior, 495
 primary expressions, 108
 print
 command, adb, 279
 command, dbXtra, 227
 command, dbxtra, 210
 stack trace, sdb, 243
 variables, dbxtra, 210
 printf(S), 450, 453, 458, 459
 prof, time-profiling, 286
 prof(CP), 42
 profile options. *See* PROFOPTS
 profiled program
 creating, 286
 running, 286
 profiler. *See* lprof
 profiling
 error messages, cscope, 287
 programs that fork(S), 289
 within a shell script, 289
 PROFOPTS, lprof
 environment variable, 286
 examples, 288
 options list, 287
 profiling within a shell script, 289
 turning off, 288
 program, generator. *See* lex
 programs
 creating profiled version, 286
 debugged with sdb, 241
 execution, adb, 265
 execution, dbXtra, 208
 execution, dbxtra, 208
 execution, sdb, 243
 generator and lprof, 297
 preparation for adb, 255
 preparation for dbXtra, 194
 preparation for dbxtra, 194, 208
 preparation for sdb, 241
 protection, SCCS, 371
 prototypes. *See* functions, prototypes
 prs command, SCCS, 349, 365
 pseudo-variable in yacc rules, 402, 405, 406
 pull-down menus, dbXtra, 196
 punctuators, 90

Q

qualified name, dbxtra, 212
 qualifiers, implementation-defined
 behavior, 494

R

recording changes, SCCS, 344
 recursion rules for yacc, 428
 reduce-reduce conflict in yacc, 420, 424
 region addresses, shared libraries, 454
 registers
 declaration, 102
 display, adb, 269
 implementation-defined behavior, 492
 manipulation, sdb, 250
 names, adb, 259
 value, adb, 259
 value change, sdb, 251
 regular expression, sdb, 246
 regular expressions in lex. *See* lex
 REJECT in lex, 390
 release numbering, delta numbering in
 SCCS, 345
 relocatable files, 44
 rerun command, dbXtra, 222
 reserved words in yacc, 430
 resetting parser with yyerror, 426
 retrieving
 different versions, SCCS, 352
 to make a delta, SCCS, 353
 return
 command, dbXtra, 224
 statement, 123
 right
 associativity rules for yacc grammar, 421
 recursion rules for yacc, 428
 rmdel command, SCCS, 349, 367
 rounding behavior, 107
 rules, yacc specification, 398, 400
 run command, dbXtra, 222
 running process, debugging, 239
 runtime analysis. *See* lprof
 run-time data file, 286

S

sact command, SCCS, 349, 366

scalar types, 99

SCCS

- access permission, 349
- auditing, 373
- beginners, 342
- body, 372
- branching, 345
- checksum, 372
- chmod, 348
- commands, 342, 343, 344, 348, 349, 352, 353, 355, 359, 360, 361, 363, 364, 365, 366, 367, 368, 369, 370
- concurrent edits of different SID, 356
- concurrent edits of same SID, 358
- creating a file using admin, 342
- creating of SCCS files, 363
- delta comments, 364
- delta file, 342, 344
- delta numbering, 345
- delta table, 372
- descriptive text, 372
- error messages, 350
- file arguments, 348
- filenames in make(CP), 328
- files, 371
- flags, 372
- formatting, 372
- get command, 350, 359
- get options, 355
- g-file, 343
- help command, 350
- ID keywords, 351
- initialization and modification of file parameters, 364
- inserting commentary for the initial delta, 364
- introduced, 341
- introduction, 11
- j flag, 353
- key letters that affect output, 359
- permission, 349
- p-file, 344
- protection, 371
- recording changes by using delta, 344
- retrieving a file, 343, 352
- SCCS identification string (SID), 342
- sccsdif command, 348, 349, 369
- SID, 345

SCCS (*continued*)

- terminology, 342
 - tree structure, 345
 - undoing a get -e, 354
 - updating a file, 353
 - user names, 372
 - x-files and z-files, 348
- scope, 101
- screen
- command, dbxtra, 231
 - dbxtra, 209
 - interface, dbxtra, 199
 - manager, dbXtra, 195
 - mode, dbxtra, 231
 - scroll command, dbxtra, 210
- script creation, adb, 278
- scroll
- command, dbxtra, 210
 - windows, dbXtra, 226
- sdb
- breakpoint management, 247
 - calling functions, 249
 - changing source file, 246
 - command line arguments, 242
 - compiling source program, 241
 - displaying variables, 243
 - exiting, 251
 - format specifier, 244
 - introduced, 241
 - length specifier, 244
 - machine language debugging, 250
 - options, 242, 247
 - program preparation, 241
 - register manipulation, 250
 - running a program, 248
 - session example, 251
 - shell command, 251
 - single stepping, 247
 - source file display, 245, 246
 - stack trace command, 243
 - startup routine, 243
 - structures, 244
- search
- command, adb, 278, 283
 - command, dbXtra, 224
 - path, dbXtra, 195
 - sdb, 245
- section
- allocation into named memory, ld(CP), 140
 - holes in physical memory, ld(CP), 141

section (*continued*)

- initialized holes, ld(CP), 140
- ld(CP), 126
- physical address, ld(CP), 126

section definition, ld(CP)

- COPY, 148
- INFO, 148
- NOLOAD, 148
- OVERLAY, 148

section type. *See* section definition

SECTIONS

- assignment statement, ld(CP), 139
- directive, ld(CP), 132
- file specification directive, ld(CP), 133
- loading to a specific address, ld(CP), 134
- output, ld(CP), 135

segment numbers, adb, 276

set breakpoint

- adb, 266
- dbxtra, 213
- sdb, 247

sh and ! commands, dbxtra, 231

shared libraries

- a.out file, 447, 450, 451, 452, 458, 460
- assembly language, 450
- benefits, 444
- #branch, 461
- branch table, 452, 453, 459, 462
- building, 454, 483
- building a.out file, 445
- building host libraries, 457
- building target libraries, 457
- C library, 444
- changing existing code, 459
- checking for different version, 473
- chkshlib(CP), 473
- code conventions, 450
- coding an application, 450
- compatibility, 462, 471, 473
- data symbol, 460
- debugging, 450
- description, 443
- example, 444, 476
- #export, 458, 459, 462, 483
- external symbol, 460
- getgrent(S), 459
- global data, 459, 462
- guidelines, 457
- #hide, 458, 459, 461, 462
- #hide linker, 483
- host, 443

shared libraries (*continued*)

- host library, 451
- implementing, 451
- importing symbols, 464, 468
- incompatible libraries, 475
- #init, 461
- lc_s option, 444
- ld (CP) link editor, 452
- .lib, 447
- libc, 464, 465
- libmalloc, 465
- link edit, 443
- lnsl_s option, 444
- malloc(S), 465
- members, 458
- memory, 443
- memory management unit (MMU), 454
- mkshlib(CP), 481, 483
- networking library, 444
- noload, 469
- #object, 461
- #objects noload, 482
- paging, 471
- printf(S), 453, 458, 459
- prof(CP), 472
- profiling, 297
- region addresses, 454, 480
- rewriting existing code, 457, 480
- routines, 458
- _s suffix, 444
- selecting contents, 456, 480
- source code, 450
- source files, 476
- space considerations, 446, 449
- specification file, 457, 462, 482
- stack variables, 459
- static data, 459
- static library symbols, 460
- static linking, 452
- strcmp(S), 449
- target library, 444, 451
- target library pathname, 456
- target pathname, 480
- tuning code, 471
- using, 483
- when to use, 446

shell

- commands in make(CP), 322
- commands, sdb, 241, 251
- execution, dbxtra, 231

shift-reduce conflict in yacc, 420, 424

- short int, 99, 106
- sidebar command buttons, dbXtra, 197
- signals, dbXtra, 221
- signed, 84, 99, 106
- single-stepping
 - adb, 268
 - sdb, 247
- size(CP), 53
- sizeof operator, 110
- skip command, dbXtra, 223
- source
 - code and assignment statement, ld(CP), 129
 - code control system. *See* SCCS
 - code, dbXtra, 194
 - code, dbxtra, 210
 - command, dbxtra, 232
 - file change, sdb, 246
 - file display and manipulation, sdb, 245
 - file display, sdb, 246
 - file in a different directory, 294
 - file, searching with cscope, 303
 - file, shared libraries, 478, 480
 - files, dbXtra, 195
 - file, shared libraries, 476
 - level debugger. *See* dbxtra
 - list, sdb, 242
 - listing for a subset of files, 294
 - menu, dbxtra, 224
 - programs debugged with sdb, 241
 - window, dbXtra, 196
- source code
 - analyzing, 13
 - archiving, 11, 18
 - beautifying with cb(CP), 13
 - compiling, 19
 - creating, 10
 - debugging, 22
 - testing, 21
 - using cflow, 16
 - using cscope, 14
 - using cxref, 17
 - using lint, 15
 - using make, 19
- specification file, 482
- specified name, dbXtra, 226
- stack trace
 - adb, 269
 - dbxtra, 211
 - sdb, 243
- stacking cscope and editor calls, 310
- start symbol in yacc specifications, 407
- starting state in lex, 387
- statements, 119
 - implementation-defined behavior, 495
- static, 102
- static data, 458, 460
- static link editor, shared libraries, 452
- static linking
 - default, 44
 - errors, 60
 - external references, 43
 - implementation, 53
- stdio.h header file, 63, 77
- step command, dbXtra, 223
- stepi command, dbXtra, 224
- stop, command, dbXtra, 217
- storage duration, 102
- strcmp(S), 64, 449, 450
- string.h. *See* header files
- strings
 - constants, 90
 - literals, 90
 - routines, 64, 69
- structure pointer operator
- structures, implementation-defined
 - behavior, 493
- structures (struct)
 - declaration, 99
 - initialization, 117
- subroutines
 - lex, 384
 - yacc. *See* yacc subroutines
- suffix rules, in make(CP), 316
- suffixes and transformation rules, in
 - make(CP), 324
- switch statement, 120
- symbolic debugger. *See* sdb
- symbols
 - global, 60
 - importing into shared libraries, 464, 468
 - ld(CP) command language, 138
 - left context in yacc specification, 405
 - names, adb, 257
 - sdb, 243
 - table, adb, 257
 - table, cscope, 304
 - table, dbXtra, 194, 195
 - table, lex, 393
 - terminal and non-terminal in yacc specification, 399
 - values in yacc rules, 402

syntax
 dbXtra, 194
 dbxtra, 194
 diagram for input directives, ld(CP), 149
 error messages in yacc, 409
 ld(CP), 129
 system calls, portability, 71

T

tabs in yacc specifications, 399
 target program, using make(CP) to
 produce, 316
 target shared libraries, 444, 451, 457, 473
 TERM environment variable for cscope, 301
 terminal
 and non-terminal symbols in yacc specification, 399
 information (terminfo) database, 301
 type setting for cscope, 301
 terminfo database, 301
 test coverage improvement with lprof, 298
 text
 display, adb, 280
 segment, 34, 53, 55, 57
 segment display, adb, 263
 .text, executable instructions
 ld(CP), 127
 shared libraries, 447
 time-profiling with prof, 286
 timestamp
 information, in make(CP), 316
 override with lprof, 295
 tokens, 86
 changing the value, 393
 declared in yacc specifications, 406
 in yacc, 399
 lex specification, 377
 lexical analysis, 397
 look ahead clearing with yyclearin, 426
 look ahead use in yacc, 413
 names, reserved, 412
 numbers in yacc, 412
 preprocessing, 92
 returning, 391
 use with lexical analyzer and parser, 391
 yacc errortoken, 424
 yyclearin, 426

trace command
 adb, 269
 dbXtra, 219
 dbxtra, 211
 sdb, 241, 243
 translation, implementation-defined
 behavior, 486
 tree structure, SCCS, 345
 trigraph sequences, 91
 tutorial, dbxtra, 208
 typedef, 102, 105
 incomplete, 106
 types
 basic, 99
 C language, 103
 casting, dbXtra, 235
 conversions, 84, 106
 incomplete, 104
 qualifiers, 99

U

unconfigured memory, ld(CP), 126
 #undef, 94
 undefined symbols, 43
 undisplay command, dbxtra, 232
 undoing a get -e, SCCS, 354
 unget command, SCCS, 349
 uninitialized variable, ld(CP), 127
 unions
 declaration, 100
 implementation-defined behavior, 493
 initialization, 117
 unsigned, 84, 99, 106
 up/down command, dbXtra, 225
 usage requirements, C++ debugging, 235
 use command, dbXtra, 224
 user
 defined subroutines. *See* yacc subroutines
 interface commands, dbxtra, 231
 names, SCCS, 372
 subroutines, lex, 377
 /usr
 ccs/lib, 45, 46
 include, 40
 lib, 45, 51

val

V

val command, SCCS, 348, 349, 370

validating addresses, adb, 277

value

left context in yacc specification, 405

stack in yacc declarations, 407

types in yacc specifications, 407

variable

adb, 258

assign, dbXtra, 227

contents, dbxtra, 211

current address, adb, 259

current value, adb, 258

dbXtra, 194

initialized, ld(CP), 127

last address, adb, 259

names, dbxtra, 208

printing, dbxtra, 210

sdb, 243

uninitialized, ld(CP), 127

value change, sdb, 251

value display, adb, 259

vc command, SCCS, 349, 370

version command, dbxtra, 232

VIEWER environment variable for cscope,
302

void, 99

volatile, 99

VPATH environment variable for cscope,
303

W

warning messages

compiler, 515

dbxtra, 194

SCCS, 352

weak symbols, 60

what command, SCCS, 348, 349, 368

whatis command, dbXtra, 226

when command, dbXtra, 216

where command, dbxtra, 211

which command, dbXtra, 227

while statement, 121

wide character constants, 90

wide characters, 90

window

command, dbXtra, 196

dbXtra, 199

scrolling, dbXtra, 226

window (*continued*)

sdb, 246, 251

source, dbXtra, 196

X

x-files and z-files, SCCS, 348

xref command, dbXtra, 227

Y

yacc

compiling and running the parser, 413

declarations, 398, 406, 407, 422, 423

delimiters, 398, 400

error. *See* yacc error

forced error recovery, 426

grammar rules. *See* yacc grammar rules

grammar use, 397

interactive error recovery, 426

introduced, 397

lexical analysis, 410

library, accessing, 409

maintenance using make(CP), 413

output file y.tab.c, 409

parser operation, 413

reserved words, 430

rules. *See* yacc rules

source files, 413

specifications. *See* yacc specifications

token names reserved, 412

tokens, 406

used with lex, 391, 399, 410

yychar, 409

yydebug, 409

YYERROR() parser error recovery, 416

yyparse() function, 409

yacc error

forced recovery, 426

handling, 424

interactive recovery, 426

recovery, reset with yyerrok, 426

reporting, 409

token, 424

yacc grammar rules

ambiguous, 419

associativity, 421

disambiguating rules, 420, 424

left associativity, yacc, 419

precedence, 421

- yacc grammar rules (*continued*)
 - reduce-reduce conflict, 420, 424
 - right associativity, yacc, 419
 - shift-reduce conflict, 420, 424
- yacc rules
 - actions, 400, 402, 404
 - default action, 403
 - empty rules, 401
 - escape sequences, 400
 - grammar, 400, 401
 - left recursion, 428
 - literals, 400
 - names, 399
 - parse trees, 406
 - pseudo-variable, 402, 405, 406
 - right recursion, 428
 - specification, 400
 - symbols, 399, 402, 405
 - terminal and non-terminal symbols, 399
 - tokens, 399
 - values, left context, 405
- yacc specifications
 - blanks, 399
 - comments, 399
 - delimiter `%%`, 398
 - examples, 430
 - format, 398
 - input style, 427
 - keywords, 406, 422, 423
 - newlines, 399
 - tabs, 399
- yacc subroutines
 - end-marker, 412
 - external variable `yyval`, 410
 - lexical analyzer, 410
 - `main()` routine, 408
 - token numbers, 412
 - user-defined, 408
 - `yerror` routine, 408
 - `yylex` routine, 408
- `y.tab.c` output file for yacc, 409
- `y.tab.h`, used with yacc, 412
- `y.tab.h` used with lex, 393
- `y.tab.h` used with yacc, 393
- `yychar` variable used in yacc, 409
- `yyclearin` token in yacc, 426
- `yydebug` used in yacc, 409
- `yyerrok`, parser reset in yacc, 426
- `YYERROR()` parser error recovery, 416
- `yerror` routine in yacc, 408
- `yyldeng` variable in lex, 381
- `yyless` function in lex, 390
- `yylex`, routine in yacc, 408
- `yylex()` lexical analyzer generated by lex, 391
- `yyval`
 - assignment, 410
 - variable used in yacc declaration, 407
 - variable used in yacc subroutines, 410
- `yymore` used to append to `yytext`, 389
- `yyparse()` function in yacc, 409
- `YYSTYPE` union declaration in yacc, 407
- `yytext` character array in lex, 381, 389
- `yyval`
 - variable, 393
 - variable used in yacc declaration, 407
- `yywrap` end-of-file processing in lex, 391

yywrap

31 January 1995

AU10503P000

SCO OpenServer™ Development System
Programming Tools Guide



AU10503P000