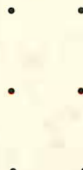


• • •
• •
•

SCO® UNIX® Development System

Programming Tools Guide



SCO[®] UNIX[®]
Development System
Programming Tools Guide



© 1983-1992 The Santa Cruz Operation, Inc.
© 1980-1992 Microsoft Corporation.
© 1989-1992 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, Santa Cruz, California, 95060, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

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. 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.

The following legend applies to all contracts and subcontracts governed by the Rights in Technical Data and Computer Software Clause of the United States Department of Defense Federal Acquisition Regulations Supplement:

RESTRICTED RIGHTS LEGEND: USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. "CONTRACTOR/SUPPLIER" IS THE SANTA CRUZ OPERATION, INC. 400 ENCINAL STREET, SANTA CRUZ, CALIFORNIA 95060, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.
DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Date: 10 December 1991
Document Version: 3.2.4D

Preface **1**

Creating programs	1
cc	1
ld	1
lex and yacc	2
m4	2
Maintaining program source files	2
make	2
SCCS	3
Using this guide	3
Documentation conventions	4
Development System documentation set	5
Using the Documentation	6
Commercial books and articles	7

Chapter 1

Compiling and linking C language programs **9**

Compiling simple programs	10
cc options	11
Specifying input files	11
File extensions	11
Specifying source files	12
Specifying output files	12
Naming object files	12
Naming the executable file	13
Linking	13
Linking with additional libraries	14
Error and warning messages	14
Setting the warning level	15
Checking syntax	16
Preparing for debugging	16
Optimizing	17

Standards conformance	19
ANSI	19
Portable Operating System for UNIX (POSIX)	20
X/Open Portability Guide 3 (XPG3)	20
System V Interface Definition Issue 2 (SVID)	21
Intel Binary Compatibility Standard 2 (IBCS2)	21
Compiling programs for XENIX	21
Compiling programs for DOS and OS/2	21
rcc	22

Chapter 2

The Link Editor

23

Sections	23
Memory configuration	24
Addresses	24
Binding	24
Object files	24
Link editor command language	25
Expressions	25
Assignment statements	26
Specifying a memory configuration	28
Section definition directives	30
Changing the entry point	40
Using archive libraries	41
Allocation algorithm	43
Incremental link editing	43
DSECT, COPY, NOLOAD, INFO, and OVERLAY sections	45
Output file blocking	46
Nonrelocatable input files	46
Syntax diagram for input directives	47

Usage	51
lint message types	53
Unused variables and functions	53
Set/used information	54
Flow of control	54
Function values	55
Type checking	56
Type casts	56
Non-portable character use	57
Assignments of longs to ints	57
Unusual constructions	57
Multiple uses and side effects	58

Introducing the C Programmer's Productivity Tools	61
lprof	62
Creating a profiled version of a program	62
Running the profiled program	62
The PROFOPTS environment variable	62
Examples of using PROFOPTS	63
Interpreting profiling output	66
Viewing the profiled source listing	66
Specifying program and data files to lprof	69
Files needed by lprof	69
Source listing for a subset of files	70
Summary option	71
Merging option	71
Cautionary notes on using lprof	72
Improving performance with prof and lprof	73
Improving test coverage with lprof	74
Using lprof with rcc	76
cscope	78
Configuring the environment	78

Using cscope	80
Running cscope	81
The cross-reference file	82
A tutorial example: locating the source of the error message	83
Conditional compilation directives	88
Examples of using cscope	89
Changing a text string	89
Adding an argument to a function	91
Changing the value of a variable	92

Chapter 5

make

93

Basic features	94
makefiles and substitutions	96
Dependency line syntax	96
Dependency information	96
Macro definitions	97
Executable commands	100
Output translations	101
Recursive makefiles	102
Suffixes and transformation rules	102
Implicit rules	102
Archive libraries	104
Tildes (~) in SCCS Filenames	106
The null suffix	107
Creating new suffix rules	108
include files	109
Dynamic dependency parameters	109
Environment variables	110
Suggestions and warnings	111
Internal rules	111

Chapter 6

Source code control system (SCCS)

117

SCCS for beginners	118
Terminology	118
Creating an SCCS file using admin	118
Retrieving a file by means of get	119
Recording changes by using delta	120
Additional information about get	120
Delta numbering	122
SCCS command conventions	124
x.files and z.files	124
SCCS commands	125
Error messages	126
The help command	126
The get command	126
The delta command	136
The admin command	139
The prs command	141
The sact command	142
The rmdel command	143
The cdc command	143
The what command	144
The sccsdiff command	145
The comb command	145
The val command	146
The vc command	146
SCCS files	147
Protection	147
Formatting	148
Auditing	149

Chapter 7

Shared libraries

151

What is a shared library?	151
Building an a.out file	153
Deciding whether to use a shared library	154
Space considerations	154

Saving space	154
Increase space usage in memory	157
Coding an application	157
Identifying <i>a.out</i> files that use shared libraries	158
Debugging <i>a.out</i> files that use shared libraries	158
Implementing shared libraries	159
The host library and target library	159
The branch table	160
Building a shared library	161
The building process	162
Guidelines for writing shared library code	165
Choosing library members	166
Include large, frequently used routines	166
Exclude infrequently used routines	166
Exclude routines that use much static data	167
Exclude routines that complicate maintenance	167
Include routines the library itself needs	167
Changing existing code for the shared library	167
Minimize global data	168
Using the specification file for compatibility	170
Importing symbols	172
Providing compatibility with non-shared libraries	179
Tuning the shared library code	180
Checking for compatibility	181
An example	185
Summary	192

Chapter 8

lex

193

Writing lex programs	195
The rules section	195
The definitions section	200
The subroutines section	201
Advanced lex usage	202
Disambiguating rules	203
Context sensitivity	203
lex I/O routines	206
Routines to reprocess input	207
Using lex with yacc	209

Using lex under UNIX systems	212
Running lex	212
Compilation	213
Execution	214
Using make with lex	214

Chapter 9

yacc

215

Basic specifications	216
The rules section	218
Terminal and non-terminal symbols	218
Rules	218
Actions	220
The declarations section	224
Tokens	224
The start symbol	225
C declarations	225
Support for arbitrary value types	225
Other declarations	226
The subroutines section	227
Lexical analysis	227
lex and yylval	227
Token numbers	229
The end marker	229
Reserved token names	229
The yacc environment	230
Compiling and running the parser	231
Compiling	231
Running the parser	231
Parser operation	232
The <i>shift</i> action	232
The <i>reduce</i> action	233
Ambiguity and Conflicts	237
Disambiguating rules	238
Precedence	239
Assigning a precedence to grammar rules	241
Error handling	242
The error token	243
Interactive error recovery	244

Hints for preparing specifications	245
Input style	245
Left recursion	246
Lexical tie-ins	246
Reserved words	247
Examples	248
A simple example	248
An advanced example	252

Chapter 10

m4: A macro processor **261**

Invoking m4	262
Defining macros	262
Quoting	264
Changing the quoting marks	265
undefine	265
ifdef	265
Using arguments	266
Using built-in arithmetic functions	267
Manipulating files	268
Using system commands	269
Using conditionals	269
Manipulating strings	270
len	270
substr	271
index	271
translit	271
Printing	272

Appendix A

ANSI implementation-defined behavior **273**

Translation	274
Identifying diagnostics	274
Environment	275
Arguments to main()	275
Interactive devices	275

Identifiers	275
Significant characters without external linkage	275
Significant characters with external linkage	275
Significance of character case	276
Characters	276
Source and execution character sets	276
Multi-byte shift states	276
Bits per character	276
Mapping character sets	276
Constants with unrepresented characters and escape sequences	276
Constants with multiple or wide characters	277
Locale used for multi-byte conversion	277
Range of char values	278
Integers	278
Integer range and representation	278
Demotion of integers	278
Signed bitwise operations	279
Sign of division remainder	279
Right shift of negative-valued signed integer	279
Floating point	279
Floating-point range and representation	279
Converting an integer to a floating-point	279
Converting a floating-point to a narrower floating-point	280
Arrays and pointers	280
Largest array size	280
Casting pointers	280
Pointer subtraction	280
Registers	281
Using registers	281
Structures, unions, enumerations, and bit-fields	281
Improper access to a union	281
Padding and alignment of members of structures	281
Sign of bit-fields	281
Order of allocation of bit-fields	282
Alignment of bit-fields	282
The type of values of an enumerated type	282
Qualifiers	282
Access to volatile objects	282
Declarators	283
Maximum number of declarators	283

Statements	283
Maximum number of case values	283
Preprocessing directives	283
Character constants and conditional inclusion	283
Locating includable source files	283
Including files with quoted names	283
Character sequences	284
Pragmas	284
Definitions for date and time	287
Library functions	288
Expanding the NULL macro	288
Diagnostic printed by the assert function	288
Character testing	288
Math functions and domain errors	288
Underflow of floating-point values	289
Domain errors and the fmod function	289
The signal function	289
Default signals	289
Signal blocking	290
The SIGILL signal	290
Terminating new-line characters	290
Space characters before a new-line character	290
Null characters appended to a binary stream	290
File position in append mode	290
Writing on text stream	290
File buffering	291
The existence of zero-length files	291
Composing valid file names	291
File access limits	291
Removing open files	291
Renaming with a name that exists	291
Output of pointer values	291
Input of pointer values	292
Reading ranges	292
File position errors	292
Messages generated by the perror function	292
Allocating zero memory	292
The abort function and open and temporary files	292
The exit function	293
Environment names	293
The system function	293

The strerror function	293
The time zone	297
The clock function	297
Locale-specific behavior	297
Content of execution character set	297
Direction of printing	297
Decimal point character	297
Character testing and case mapping	297
Collation sequence	299
Time and date formats	299
C6.0 Implementation limits description	300
Environmental limits	300
Translation limits	302

Appendix B

Compiler exit codes and error messages **305**

Compiler exit codes	305
Command-line error messages	305
Command-line fatal-error messages	306
Command-line error messages	306
Command-line warning messages	308
Compiler error messages	309
Fatal-error messages	310
Compilation-Error messages	315
Warning messages	331

Preface

This guide explains how to use the UNIX Development System to create and maintain C programs. The system provides a broad spectrum of programs and commands to help you design and develop application and system software.

The following sections introduce the programs and commands of the UNIX Software Development System.

Creating programs

The C programming language can meet the needs of most programming projects. The UNIX Development System provides a full set standards-conforming libraries that will support you in the development of portable software.

cc

You can use the `cc` command to compile and link C programs. This command also accepts and processes object files, library archives, and assembly language files.

ld

The 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.

lex and yacc

You can create source files for lexical analyzers and parsers using the program generators **lex** and **yacc**. You use lexical analyzers in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. You use parsers in programs to convert meaningful sequences of tokens and values into actions. The **lex** program generates lexical analyzers, written in C, from given specification files. The **yacc** program generates parsers, written in C, from given specification files. You can use **lex** and **yacc** together to make complete programs.

m4

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files, using the **m4** macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file.

Maintaining program source files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS (Source Code Control) commands.

make

The **make** program is the UNIX program maintainer. It automates the steps required to create executable programs and provides a mechanism for ensuring that programs are up-to-date.

You can use the **make(CP)** command to simplify the tasks of maintaining, updating, and regenerating your executable programs. 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.

make executes the instructions in the *makefile* to update one or more target files. It checks all of the components of the specified target file to ensure that only the portions that have been changed since the target file was last created are compiled. After compiling all updated source files, **make** can call the linker to create the executable file. **make** can be used to run UNIX shell commands so that it can take advantage of other operating system features.

make follows certain filename extension conventions that direct it to call other processors such as **lex**(CP) and **yacc**(CP). Files that need to be processed by one of these utilities can be included in the *makefile*.

make can extract files from SCCS when it updates your target file(s). See the following section for more information on SCCS.

SCCS

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.

Refer to the SCCS chapter in this guide for more information on using SCCS.

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, "Compiling and Linking Your Program," explains how to compile and link using the **cc** command.

Chapter 2, "The Link Editor," contains information on the Link Editor and the command language which can be used to modify the default behavior of **ld**(CP).

Chapter 3, "lint," examines C language source files to detect bugs and obscure code constructions.

Chapter 4, "C Programmer's Productivity Tools," teaches you how to use **scope**, a browser, and **lprof**, a line profiling tool.

Chapter 5, "make," helps you keep track of file-to-file relationships, the order of command executions, and general file maintenance.

Chapter 6, "Source Code Control System (SCCS)," explains how to control and maintain all versions of a project's source files using the SCCS commands.

Chapter 7, "Shared Libraries" describes the C shared Libraries.

Chapter 8, "lex," explains how to create lexical analyzers using the program generator. **lex**.

Chapter 9, “yacc,” explains how to create parsers using the program generator, yacc.

Chapter 10, “m4,” explains how to imbed m4 macros in your files and preprocess them.

Documentation conventions

SCO documents use font changes and other typographic conventions to distinguish text elements. The following table shows these conventions:

Font conventions

Example	Entity
cc or cc(CP)	command. The “CP” indicates the manual page section in which the command is documented.
<i>libc.a</i>	filename
<i>action</i>	placeholder
⟨Esc⟩	keyname
open or open(S)	system calls, library routines, kernel functions, C keywords. The “S” indicates the manual page section in which the command is documented.
buf	structure
b.errno	structure member
\$HOME	environment or shell variable
SIGHUP	named constant
login	output
“clobber”	jargon
“adm3a”	data value
date	user input

Development System documentation set

The contents of the SCO UNIX System V/386 Development System documentation set are illustrated here. In addition to these books, one set of *Release and Installation Notes* is shipped with each copy of the Development System software.

Overview	<i>Developer's Overview</i>	
Reference	<i>Programmer's Reference Manual (2 volumes)</i>	<i>Encyclopedia</i>
Guides	<i>Programming Tools Guide</i>	<i>Developer's Topics</i>
	<i>Debugging Tools Guide</i>	<i>User Interfaces Guide</i>
Optional	<i>Device Driver Writer's Guide</i>	<i>Macro Assembler Writer's Guide</i>

The books included with the Development System are described here.

Developer's Overview

introduces the Development System facilities and gives general information about developing software to run on SCO UNIX systems and the supported cross-development environments

Programmer's Reference Manual

a two-volume set that includes manual pages for the entire Development System, including sections (CP), (DOS), (FP), (S), and (XNX). See the "Manual pages" article in the *Encyclopedia* for a description of these manual page sections.

Encyclopedia

contains articles that give background information about system internals, descriptions of facilities, and other general issues. Articles are arranged alphabetically.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include extensive examples that illustrate features that SCO added to the porting base.

Programming Tools Guide

provides generally useful information about programming tools and their use in developing software. It also provides implementation-specific details about the ANSI-conforming C compiler that is provided with the Development System.

Debugging Tools Guide

provides generally useful information about the debugging tools, and their use in tracking down and eradicating problems in C and assembly language programs. In addition, it includes information on using dbXtra to debug C and C++ programs in a Motif-based windowing environment.

User Interfaces Guide

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

In addition, the following books are available for Development System customers who need them:

Device Driver Writer's Guide

Provides guide and reference information describing how to write device drivers for SCO UNIX systems. This book is sold separately.

Macro Assembler Writer's Guide

Provides information on the assembly language for the Intel 286, 386, and 486 processors. Developers who are writing device drivers or applications that have tight performance requirements may want to write portions in assembly language. Return the coupon enclosed with the Development System documentation to receive a copy.

Using the Documentation

We recommend the following approach to using this documentation set. In general, we suggest the following:

1. All users should read the *Developer's Overview* thoroughly, soon after installing the software. Experienced users may skim some sections, but this book gives a comprehensive overview of the Development System's capabilities.
2. Use the *Encyclopedia* to look up topics about which you need more information. The *Encyclopedia* gives both summary and background information on many topics, and includes cross-references to sources of more in-depth information.
3. The *Programmer's Reference Manual* contains complete reference information about all capabilities provided in the Development System. New users are warned, however, that the manual pages provide reference material that may be difficult to utilize if they do not understand basic concepts.

The *Programmer's Reference Manual* contains separate sections for Development System commands (CP), system services (S), Development System file formats (FP), and other topics. The pages in each section of the *Programmer's Reference Manual* are arranged alphabetically, although some related facilities are grouped together on a page. If you cannot locate the correct manual page, use the permuted index to find the one that contains the information you want. The "Manual pages" article in the *Encyclopedia* explains how to use the permuted index.

4. The *Programming Tools Guide* and *Debugging Tools Guide* are mainly for users who have not used one of these tools before, or who want to reacquaint themselves with some aspect of that tool's operation. These books are good places to look when the *Encyclopedia* and *Programmer's Reference Manual* are giving you lots of detail, but not enough guidance. Nonetheless, even experienced programmers will discover helpful hints and practical tips on the tools discussed in these books.

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. We have not attempted to replicate guide information on all of these topics; manual pages are provided, and many articles in the *Encyclopedia* list additional sources of information on their subject matter. In addition, the *Encyclopedia* includes a large "Bibliography" that will interest users of the Development System.

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

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

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

See the "Bibliography" in the *Encyclopedia* for full citations if you are not familiar with these books.

Chapter 1

Compiling and linking C language programs

The Development System provides two C language compilers: `cc`(CP) is a fully ANSI-compliant implementation, based on the Microsoft C compiler Version 6.0, while the AT&T compiler `rcc`(CP) is available for programmers who need to produce “K&R” (Kernighan and Ritchie) code. Most of the information in this chapter deals with the `cc` command. See `rcc`(CP) for specific information on this command.

The `cc` command accepts C source files, object files, assembly-language files and library files; it compiles and links these files to create an executable file. For further information, see the “Specifying input files” and “Specifying output files” sections in this chapter.

`cc` also controls the linking process. You can alter the way in which files are linked by using command line options to the `cc` command. For information on linker options, see the “Linking” section.

The parser called by `cc` can be used to check the syntax of your C language source files without compiling them. See the section called “Checking syntax” for more information.

The `cc` command automatically produces optimized machine language code. You do not have to specify optimizing instructions to `cc` unless you want to change the way `cc` optimizes, request more sophisticated optimizations, or disable optimization altogether for ease of debugging. For more information, see the “Optimizing” and “Preparing for debugging” sections in this chapter.

The `cc` command can ensure that the source code conforms to the ANSI, POSIX, XPG3, IBCS2 and SVID standards for C language compilers. For further information, see the sections pertaining to each standard.

You can use `cc` to compile programs to run under SCO XENIX, or to incorporate functions from the XENIX libraries. For more information, see the "Compiling programs for XENIX" section.

You can also compile programs for DOS or OS/2 systems with the `cc` command. For more information, see the "Compiling programs for DOS and OS/2" section.

There are several other programming tools that you may want to use for various development tasks, and these are discussed at the end of this chapter in the "Using other programming tools" section.

Refer to the `cc(CP)` manual page for detailed information on `cc` functionality.

Compiling simple programs

You can compile a simple C language source file like the one in the following example to produce an executable program using `cc` without specifying any options.

```
#include <stdio.h>

main()
{
    printf("This is a test\n");
}
```

To compile and link this program, enter this command:

`cc test.c`

`cc` echoes the filename to the screen and then returns you to the system prompt. The executable file produced by `cc` is called *a.out*. To test the program, enter the command:

`a.out`

You will see the following response on your screen:

This is a test

cc options

`cc` offers a large number of command options to control the compiler. You can use command options to rename your executable file, save the generated object files, modify or disable optimization, and so on. Options consist of a dash (-) followed by one or more letters. Refer to the `cc(CP)` manual page for detailed information on the command line options.

Options can appear anywhere on the `cc` command line. In general, an option applies to all files that follow it on the command line, and does not apply files preceding it. Except where specifically noted in the manual page, options do not affect any object files specified on the command line.

Specifying input files

The `cc` command can process source files, object files, and library files. It is a common practice to use the file extension to indicate the type of file, although this is not mandatory when using `cc`.

File extensions

`cc` uses the filename extension to determine what kind of processing the file requires, as shown in Table 1.1.

Table 1-1 File extensions

Extension	Action taken by <code>cc</code>
<code>.c</code>	compiled as a C source file
<code>.o</code>	linked as an object file
<code>.a</code>	passed to the linker and searched as a library file, unless the <code>-c</code> option, which suppresses the linking stage, is specified
<code>.asm</code>	passed to <code>masm</code> as an OMF assembly-language source file
<code>.s</code>	passed to <code>as</code> as a COFF assembly-language source file
none	linked as an object file
anything else	linked as an object file, unless the <code>-Tc</code> option (see below) is used

Specifying source files

To indicate to `cc` that a file is a C source file even though it does not have the `.c` extension, use the `-Tc` command line option. This tells `cc` that the file that follows the option is a C source file. One or more spaces can appear between `-Tc` and the filename. A separate `-Tc` option must appear before each source file that has an extension other than `.c`, as shown below:

```
cc main.c -Tc test.prg -Tc collate.prg print.prg
```

This example compiles three source files: `main.c`, `test.prg`, and `collate.prg`. Because `print.prg` is specified without a `-Tc` option, `cc` treats it as an object file. After compiling the three source files, `cc` links the object files `main.o`, `test.o`, `collate.o`, and `print.prg`, creating the executable file, `a.out`.

Specifying output files

Normally, after the linker creates the executable file from the object files and libraries, it removes the generated object files. If the linker is unable to create the executable file, the object files are not removed from the system. Object files which were specified on the `cc` command line are not removed at the end of the link phase.

Naming object files

When `cc` compiles C source files, it gives each object file created the base name of the corresponding source file plus the `.o` extension. (The base name consists of all characters in the filename up to the final dot.) The `-Fo` option lets you specify alternate object filenames or cause object files to be created in a different directory. Object files specified on a `-Fo` option will not be removed at the end of the link phase.

The `-Fo` option is used as follows:

```
cc -Foobjfile.o file1.c
```

The object filename argument must appear immediately after the option, with no intervening spaces; the corresponding source file must follow the object filename, separated from it by whitespace. In the preceding example, `file1.c` is compiled into the object file `objfile.o`.

NOTE You are free to supply any name and extension you like for `objfile`. However, it is recommended that you use the conventional `.o` extension because the linker uses `.o` as the default extension when processing object files. A `.o` extension is added if you do not specify one.

If you specify a directory (which must end with a forward slash) following the `-Fo` option, `cc` creates the object files in the specified directory and gives them the default filename (the basename with the extension `.o`). Otherwise, the object files is created in the current directory. Here is an example of naming object files:

```
cc -Foobject1/ this.c that.c -Fo/src/newthose those.c
```

In this example, the first `-Fo` option causes `object1/this.o` to be created as a result of compiling `this.c`. The compiler also creates `object1/that.o` as a result of compiling `that.c`. The second `-Fo` tells the compiler to create `/src/newthose.o` from `those.c`).

Naming the executable file

If the `cc` command successfully compiles your source files, it calls the linker to create the executable file. By default, `cc` names the executable file `a.out`. You can specify an alternate name by using the `-o` command line option. This option lets you specify the name of the executable file or cause it to be created in a different directory. The `cc` command creates only one executable file, so you can specify the `-o` option anywhere on the command line. If more than one `-o` option appears, the last one on the command line is used. Here is an example:

```
cc -o process ./*.c
```

This example compiles all source files in the current directory with the extension `.c`. The resulting object files are linked and the executable file is named `process`.

Linking

After compiling the source files specified on the command line, `cc` calls the linker, `ld`, to create the executable file. The `ld` command combines the object files generated by `cc` with any object files specified on the command line, resolves any external references, and copies the result to the executable file. You can specify linker options after the `-link` option, which must appear at the end of the `cc` command line. These are passed to the linker at the end of the compilation. As well, any options not understood by the compiler will be passed on to the linker.

When an object file is linked, the linker tries to resolve any external references in the object file by searching the library file(s). Any definitions required from the standard C library are automatically linked in without your having to specify this library on the command line. There are a number of standard libraries: the memory-model (`-M`) option you specify on the `cc` command line determines which are used. If you use the default small memory-model (`-Ms`), `cc` uses the libraries `libc.a` and `libintl.a`.

The linker called by `cc` is `/bin/ld`, which produces executable files in Common Object File Format (COFF) by default. If the object files specified on the command line are in Object Module Format (OMF), the linker creates a XENIX executable file. You must use one of the `cc` command line options `-xenix`, `-xout`, or `-x2.3` to produce OMF files. You cannot link OMF and COFF files together in one executable.

Linking with additional libraries

The `-l` option may be used to specify additional libraries. The `-l` option and its argument must appear *after* the file or files which require the library. If you specify additional libraries to `cc` (such as the `math`, `curses`, and `lex` libraries), the linker searches these libraries before it searches the default library to resolve external references in the object files. To compile a program with the `lex` library, use the `l` argument after the `-l` option, as shown below:

```
cc lex.yy.c -ll
```

If a library name includes a path specification, the linker searches only that path for the library. If you specify only a library name (without a path specification), the linker searches for the specified library file in the locations listed below, and in the order given:

- the current directory
- any paths specified, in their order of appearance on the command line
- the default library location, `/lib`

If a library name is specified without an extension, the linker automatically assumes the `.a` extension. To link with a library file that has any other extension, specify the complete library name, including the extension.

Error and warning messages

Error and warning messages can appear at different stages of program development and usage:

- During compilation, the compiler generates a broad range of error and warning messages to help locate errors and potential problems in the C source code. Compiler error and warning messages begin with the letter `C`, followed by a four-digit error number.
- During linking, the linker generates error messages.
- During program execution, the system displays runtime error messages. This category includes messages about core dumps, segmentation violations, and floating-point exceptions (errors generated by an 8087, 80287, or 80387 coprocessor).

Compiler error messages are sent to standard error, which is usually the terminal. The general form of error messages is as follows:

```
filename(line number): error or warning number: message
```

For a full list of all possible error and warning messages, see “Appendix E.”

Compiler error and warning messages are numbered in the following sequence:

- Fatal compiler errors are numbered from C1000.
- Other compiler errors are numbered from C2000.
- Compiler warnings are numbered from C4000.

You can redirect the messages to a file by using the Bourne shell or Korn shell redirection symbols at the end of your command line:

```
cc file.c -o file.o 2> file.err
```

If you are using the C shell, you can redirect the messages to a file by using standard redirection syntax:

```
cc file.c -o file.o >& file.err
```

Setting the warning level

You can suppress warning messages produced by the compiler by using the **-W[0123]** or **-w** (warning) options. Compiler warning messages are any messages numbered C4000 and up. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend. The **-W[0123]** and **-w** options apply only to source files specified on the command line: they do not apply to object files.

The **-W0** and **-w** options turn off warning messages. These options are useful when you compile programs that deliberately include questionable statements. The **-W0** or **-w** option applies to the remainder of the command line or until the next **-W[0123]** option on the command line.

The **-W1** option (the default) causes the compiler to display most warning messages.

The **-W2** option causes the compiler to display more warning messages. Level 2 warnings may or may not indicate serious problems. They include the following:

- use of functions with no declared return type
- failure to put **return** statements in functions with non-**void** return types
- data conversions that cause loss of data or precision

The **-W3** option displays the highest level of warning messages, including warnings about the uses of non-ANSI features and extended keywords and about function calls before the appearance of function prototypes in the program.

You can also use the **-WX** option to make any warning message function as a fatal error. In this case, you would only create an executable file when no warning or error messages are generated.

Checking syntax

The **cc** command can be used to check the syntax of source code. This can be useful in detecting typing errors, mismatched braces and parentheses, and other problems.

The **-Zs** option tells the compiler to perform a syntax check on the source files that follow it on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file. The compiler will display error messages if the source file has syntax errors, but will not produce object files, object listings, or executable files.

The **-Fs** option directs **cc** to generate a source listing that shows these error messages in the locations they occurred.

Preparing for debugging

CodeView, or **sdb**, you must create an executable file with full symbolic-debugging information and source-code line numbers so that the debugger can match the object code to the relevant source code. The **-g** option to **cc** causes it to incorporate the information necessary for debugging in the executable file.

The **-Od** option tells the compiler not to perform most optimizations: some simple optimizations will still be performed. Without the **-Od** option, **cc** optimizes by default. This is the only **-O** option that should be used when you plan to use a symbolic debugger with your object file, because optimization usually involves rearrangement of instructions in the object code. This could make it difficult to understand the order of execution when debugging.

If the **-g** option is specified with no explicit **-O** options, all optimizations involving code rearrangement are suppressed, although the compiler may continue to perform simple optimizations. If any explicit **-O** options are specified, all requested optimizations are performed. Other optimization options are discussed in the section on "Optimizing."

Optimizing

The optimizing capabilities of the C compiler can reduce the storage space or execution time required for your program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default: you can use the **-O** options to exercise control over the optimizations performed.

Optimizing for speed

When you do not specify a **-O** option, **cc** automatically uses **-Ot**, causing program-execution speed to be favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps slower) and faster (but perhaps larger) code, the compiler generates faster code.

Optimizing for code size

To cause the compiler to favor smaller code size instead of the default faster code, use the **-Os** option.

Disabling optimization

The **-Od** option turns off all optimizations, except for some optimizations within basic blocks (blocks of instructions that have a single entry point and a single exit point). This is useful in the early stages of program development to avoid spending the time optimizing code that will later be changed. Because optimization may involve rearrangement of instructions, you should specify the **-Od** option when you use a debugger with your program or when you want to examine an object-file listing. Note that turning off or restricting optimization of a program usually increases the size of the generated code.

Achieving consistent floating-point results

The **-Op** option is useful when floating-point results must be consistent within a program. This option changes the way in which the program handles floating-point values.

Ordinarily the compiler stores each floating-point value in an 80-bit register. In subsequent references to that value, the compiler reads the value from the register. When the final value is written to memory, it is truncated, because floating-point types are allocated fewer than 80 bits of storage (32 bits for **float** type and 64 bits for **double** type). Therefore, the value stored in the register may actually be more precise than the same value stored in a floating-point variable.

If you use the **-Op** option, when floating-point values are referenced, the compiler reloads them from floating-point variables rather than from registers. Because the value is truncated each time it is written to memory, over the

course of the program a value which was maintained in the register may become quite different from what it would have been had it been loaded from memory each time the variable was referenced. Using **-Op** gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more control over the truncation (and hence the consistency) of floating-point values.

Wide return values

Many programmers assume that if the C language source code does not have accurate function declarations in scope, the functions will return type **int** values anyway. This can cause problems if a function in your program actually returns a type **char** or type **short** value, due to the possibility that the high-order part of the **int** is non-zero. Therefore, the **cc** command “widens” the return value by default for 386 compilations.

You can use the **-Oh** option to **cc** to ensure that the **char** or **short** return values are not “widened” to **int**. This option applies only to 386 compilations, because the compiler does not “widen” return values when the target platform is not a 386.

Producing maximum optimization

The **-Ox** option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using **-Otlige** on the command line.

Unsafe optimizations

Some optimizations are not always safe, that is, they rely on assumptions about the code which will not be true of all C code. This does not imply a bug in the optimization: rather, certain more “aggressive” optimization procedures will be carried out on the assumption that if the user specifies particular optimization options, the code satisfies the criteria for safe optimization. In particular, the **-Oa** option, which relaxes alias checking, the **-Oc** option, which enables optimization of block-level common subexpressions, and the **-Oe** option, which enables global register allocation, should be used with caution. The **-Ox** option, which causes maximum optimization, will turn on some optimizations which are possibly unsafe.

Standards conformance

The `cc` command conforms to the ANSI, POSIX, XPG3, and SVID standards.

ANSI

For more information, see Appendix A, “ANSI Implementation defined limits.”

Enabling or disabling language extensions

The C compiler is fully compliant with the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These additional features are enabled when the `-Ze` (default) option is in effect and disabled when the `-ansi` option is in effect. They include the following:

- the `cdecl`, `far`, `fortran`, `huge`, `near`, and `pascal` keywords
- use of `casts` to produce values, as in this example:

```
int *p;
((long *)p)++;
```

The preceding example could be rewritten to conform with ANSI C as shown here:

```
p = (int *)((char *)p + sizeof(long));
```

- use of trailing commas (,) without ellipses (,...) in function declarations to indicate variable-length argument lists, such as:

```
int printf(char *,);
```

- use of bit fields with base types other than `unsigned int` or `signed int`

Use the `-ansi` option if you are porting your program to other environments. This ensures that the code conforms to the ANSI standard. The `-ansi` option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed previously.

Predefined macro names

The C compiler supports all of the predefined macro names found in the ANSI standard for the C language. These provide a convenient means for obtaining the date and time of the compilation and for indicating whether the compiler purports to conform fully to the ANSI standard.

The following list explains each of these macro names:

<code>__DATE__</code>	date of compilation, expressed as a string literal in the form: Mmm [d]d yyyy
<code>__FILE__</code>	filename, expressed as a string literal
<code>__LINE__</code>	number of the line that the <code>__LINE__</code> macro appears on
<code>__STDC__</code>	integer constant 0. If equal to 1, this macro indicates full conformity with the ANSI C language standard.
<code>__TIME__</code>	time of compilation, expressed as a string literal in the form: hh:mm:ss

The `__TIMESTAMP__` macro name offers a capability not found in the ANSI standard. It gives the date and time of last modification of the source file, expressed as a string literal in the form:

Ddd Mmm [d]d hh:mm:ss yyyy

The time and date provided by `__TIMESTAMP__` indicate the actual timestamp of the source file, whereas `__DATE__` and `__TIME__` indicate the time of compilation.

The following code fragment uses three predefined macros with the `message` `#pragma` to display informational messages at the time of compilation.

```
#pragma message("Compilation date: " __DATE__)
#pragma message("Compiling: " __FILE__)
#pragma message("Last modification: " __TIMESTAMP__)
```

Here is the output you might see from the preceding code fragment:

```
Compilation date: Feb 11 1992
Compiling: sample.c.
Last modification: Mon Feb 10 12:02:51 1992
```

Portable Operating System for UNIX (POSIX)

You can force the `cc` command to compile only POSIX-conforming code by using the `-posix` command line option.

X/Open Portability Guide 3 (XPG3)

You can force the `cc` command to compile only XPG3-conforming code by using the `-xpg3` command line option.

System V Interface Definition Issue 2 (SVID)

You can force the `cc` command to compile only SVID-conforming code by using the `-svid` command line option.

Intel Binary Compatibility Standard 2 (IBCS2)

You can force the `cc` command to compile only IBCS2-conforming code by using the `-iBCS2` command line option.

Compiling programs for XENIX

By default, `cc` produces object and executable files in the COFF format, which is the same format used by the AT&T development system. `CP>-xenix` The `-xenix` option causes `cc` to produce OMF object and executable files, which are compatible with the XENIX System V Development System tools. When the `-xenix` option is used with any of the options that produce assembly-language output, the warning message normally issued ("`masm directives`") is suppressed.

The `-xout` option causes `cc` to produce OMF object and executable files that include the functionality of SCO UNIX System V/386 Release 3.2. Programs generated using this option may not run properly under XENIX.

The `-x2.3` option is equivalent to the `-xenix` option, but it also includes the extended functions available with XENIX System V/386 release 2.3. When used with the memory model option `-M2`, which enables 80286 code generation, this option produces XENIX System V/286 2.3 compatible files.

SCO UNIX System V can execute either COFF or OMF programs.

Compiling programs for DOS and OS/2

The `cc` command is capable of creating object code that executes in the DOS or OS/2 environments. The `-dos` option instructs the compiler to use the set of libraries in `/usr/lib/dos` and to use a different linker, `dosld(CP)`.

The `-os2` option instructs the compiler to use the libraries in `/usr/lib/os2` and to call the `os2ld` linker command.

Programs compiled with `-dos` or `-os2` cannot run in the UNIX System V environment, and many UNIX System V system calls are not supported in DOS or OS/2.

There is a variety of **-FP** options that can be used with **-dos** or **-os2** to control floating-point operations. See the *DOS and OS/2 Development Guide* for more information.

NOTE If either **-dos** or **-os2** are specified on the **cc** command line, it overrides any other options specified on the command line, and the resulting executable file is in DOS or OS/2 format. If both are present, the first one to appear will be used.

rcc

The AT&T compiler **rcc** is provided in the Development System to ensure that pre-ANSI C source programs can be compiled. The **rcc** command produces COFF files only.

rcc accepts input files with **.c** extensions as C language source files; files with **.o** extensions are assumed to be COFF object files. The **rcc** command calls the AT&T assembler (**as**) to assemble any files that have the extension **.s**. Input files with **.i** extensions are assumed to be preprocessed C source files. Files with any other extension or with no extension are passed to the linker as object files.

For more information on this compiler and the available options, see the **rcc(CP)** manual page.

Chapter 2

The Link Editor

This chapter contains information on the link editor, `ld`(CP), and the command language which can be used to direct its operations.

`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. These will 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.

The command language supported by `ld` allows the user a greater degree of control over object files and their memory locations.

The discussion that follows relies heavily on the reader having knowledge of the COFF file format. In particular, familiarity with the concept of a **section** is essential to understanding this material. Refer to the SCO UNIX System V/386 Development System Encyclopedia for more information.

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, setting aside 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 compiler (which generates an OMF file and converts it to COFF using `cvtomf`) and by `ld`. `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 shared libraries. The `.fini` section, if it exists, contains finalization code used in shared libraries. Sections with other names than these may be present in object files that were created by `ld`.

Link editor command language

This section gives information on the `ld` command language. 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.

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*. 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.

Expressions

Expressions in the link editor command language may contain global symbols, constants, and most of the basic C language operators. (See the section “Syntax diagram for input directives.”) 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.

The following names are reserved and hence unavailable as symbol or section names:

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 Figure 2-1, in order of precedence from highest to lowest:

symbol
! ~ - (UNARY Minus)
* / %
+ - (BINARY Minus)
>> <<
== != > < <= >=
&
&&
= += -= *= /=

Figure 2-1 Operator Symbols

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 `ld` directives.

Syntax

The syntax of the assignment statement is:

```
symbol op expression;
```

op may be one of the one of the operators "=", "+=", "-=", "*=", or "/=". Assignment statements must be terminated by a semicolon.

Interaction with source code

Assignment 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 symbol's 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"). 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." 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 uppercase 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 which do not mention that portion of memory.

Section definition directives

The purpose of the **SECTIONS** directive is to describe how input sections 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 file name 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 **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 of determining 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 **crt1.0** (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".

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 must satisfy a reference that is known to be unresolved at the time the library is searched. 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 `rou1` 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 rou1 -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 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 `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:

- 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 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>
<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> }

(Continued on next page)

(Continued)

Directives	Expanded Directives
	[<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
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> .
<assign_op>	= += -= *= /=
<end>	; ,
<expr>	<expr> <binary_op> <expr> <term>
<binary_op>	* / % + - >> << == != > < <= >=

(Continued on next page)

(Continued)

Directives	Expanded Directives
	&
	&&
<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
	-o<wht_space><filename>
	-r
	-s
	-t
	-u<wht_space><name>
	-z
	-H
	-L<path_name>
	-M
	-N

(Continued on next page)

(Continued)

Directives	Expanded Directives
	-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 3

lint

lint examines C language source files to detect bugs and obscure code constructions. It enforces the type rules of C more strictly than the C compiler does. **lint** also enforces any portability restrictions involved in moving programs between different machines or operating systems. In addition, **lint** can detect legal C constructs that are wasteful or error-prone. **lint** accepts multiple input files and library specifications and checks them for consistency. Two versions of **lint** are currently available. These are **lint** and **rlint**. **lint** is based on the the first pass of the Microsoft C compiler. This understands both the ANSI and Microsoft dialects of the C language. **rlint** is based on the first pass of the AT&T C compiler. All references to **lint**, in this chapter will also apply to **rlint** unless explicitly stated otherwise.

Usage

The syntax for **lint** is:

lint [*options*] *files* ... *libraries* ...

The syntax for **rlint** is:

rlint [*options*] *files* ... *libraries* ...

The arguments are as follows:

options	flags to change the default settings for lint or to suppress certain types of messages displayed by lint
files	one or more files on which lint is to be run
libraries	additional libraries to be used, other than the Standard C Library which is used by default

The names of files written in C must end with the suffix *.c*. Files ending with *.ln* are either **lint** library files or output files from **lint**'s first pass. If **rlint** is used, the equivalent files would end with *.rln*.

For the full list of options, refer to the **lint**(CP) and **rlint**(CP) manual pages. Multiple options can be combined into a single argument, such as **-ab** or **-xha**.

Lint accepts the option **-la** lint library file of the form *x.ln* found in the directory */usr/lib*. The **lint** math library */usr/lib/lib-lm.ln* is called with the option **-lm**. The library */usr/lib/lib-lm.rln* is referenced if **-lm** is used with **rlint**. The **-p** option is used to call the portable lint library in place of the standard C lint library used by default. **lint** libraries are not actual libraries but are library description files corresponding to the equivalent C libraries. These description files all begin with the comment:

```
/* LINTLIBRARY */
```

A series of dummy function definitions follow the comment. The critical parts of these definitions are: the declaration of the function return type; the number and types of arguments to the function; and whether the dummy function returns a value, for example:

```
int putchar(c) int c; { return (c); }
```

In this example, **int putchar** indicates that the function **putchar** returns an integer. The function name is followed by a list of argument type definitions separated by semi-colons (;). In the above example, **int c** indicates that the single argument **c** must be an integer. Finally, **return (c)** indicates that the function returns its argument.

lint libraries are processed like ordinary source files. However, functions that are defined in a library, but not used in a source file, do not result in warning messages. **lint** does not simulate a full library-search algorithm, and it will print messages if the source files contain a redefinition of a library routine.

lint checks programs against the standard C lint library by default. To suppress checking against the standard C lint library or the portable lint library the **-n** option can be used.

lint message types

The following paragraphs describe the major categories of messages printed by lint.

Unused variables and functions

As programs evolve, variables and arguments to functions may cease to be used. It is not uncommon for external variables or even entire functions to become unnecessary but remain in the source. These errors rarely cause programs to fail, but they are inefficient and make the source harder to understand.

lint prints messages about defined but unused variables and functions. These messages can be suppressed with the **-u** or **-x** option.

A function can be written that does not use all of its arguments. Normally lint will produce warning messages that indicate which arguments are not used in a function. If the program is still being developed it may be desirable to suppress these messages. This can be done with the **-v** option to lint. This suppresses all messages about unused arguments except those arguments which are also declared to be register variables. This is an active (and preventable) waste of the register resources of the machine.

The comment `/* ARGSUSED */` can be placed before a function in the source code to suppress messages about unused arguments. This has the same effect as using the **-v** option with lint but applies only for the single function.

The comment `/* VARARGSn */` can be used before functions in sources files and in lint library files to indicate that the function can take a variable number of arguments. The value *n* indicates the number of arguments that lint should perform type checking on, for example:

```
/* VARARGS1 */
int printf(s) const char *s; { return (0); }
```

This entry in the standard C lint library file indicates that **printf** can take multiple arguments and that lint should only perform type checking on the first argument. If `/* VARARGS */` is used without an *n* value then no argument type checking is done.

When lint is applied to some, but not all, files of a collection that are to be loaded together, it issues warnings about unused or undefined variables. Functions and variables that are defined may not be used in the files checked by lint; conversely, functions and variables defined elsewhere may be used. The **-u** option suppresses these spurious messages.

If a program is divided into many files it is likely that some of these files will have external variable declarations. If external variables are declared but unused in any of these files then *lint* will produce a warning message. These warning messages can be suppressed by use of the **-x** option to *lint*.

Set/used information

lint attempts to detect cases where a variable is used before it is set. It detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a “use,” because the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, because the true flow of control need not be discovered. *lint* can print error messages about program fragments that are legal but are considered bad stylistically. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. *lint* does deal with initialized automatic variables.

The set/used information also allows recognition of those local variables that are set but never used. These are frequent sources of inefficiencies and may also be symptomatic of bugs.

Flow of control

lint attempts to detect unreachable portions of a program. It prints messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom, and to recognize the special cases **while(const)** and **for(;;)** as infinite loops. *lint* also attempts to warn about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered bad style. Use the **-b** option to suppress warnings about unreach code.

The *lint* command has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code that *lint* does not detect. The most serious effects of this are in the determination of returned function values, (see the next section, “Function Values”). If you think that a particular place in the program is unreachable in a way that is not apparent to *lint*, the comment can be added to the source code at the appropriate place:

```
/* NOTREACHED */
```

This informs *lint* that a portion of the program cannot be reached, and to ignore it (that is, do not print a warning).

Programs generated by **lex** and **yacc** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up **lint**'s output. We recommend invoking **lint** with the **-b** option when dealing with such input.

Function values

Sometimes functions return values that are never used; sometimes programs incorrectly use function values that have never been returned. **lint** addresses these problems in a number of ways.

Locally, the appearance of both of these lines within a function definition is cause for alarm:

```
return( expr );
```

```
return;
```

lint returns the message:

```
function name has return(e) and return
```

The difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f (a)
{
    if (a) return (3);
    g ();
}
```

If **a** tests false, **f** calls **g** and then returns with no defined return value; this triggers a message from **lint**. If **g**, like **exit**, never returns, the message is still produced, even though nothing is wrong. The comment */*NOTREACHED*/* in the source code suppresses this warning. In practice, this feature has detected some potentially serious bugs.

Globally, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
(void) fprintf(stderr, "File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (such as not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected.

Type checking

lint enforces the C type-checking rules more strictly than compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments
- at the structure-selection operators
- between the definition and uses of functions
- in enumerations

A number of operators have an implied balancing between operand types. The assignment, conditional, and relational operators have this property. The argument of a `return` statement and expressions used in initialization suffer similar conversions. In these operations, `char`, `short`, `int`, `long`, `unsigned`, `float`, and `double` types may be freely intermixed. Pointer types must agree exactly, except that arrays of `x`'s can be intermixed with pointers to `x`'s.

The type-checking rules for structure references also require that the left operand of the `"->"` be a pointer to structure, the left operand of the `"."` be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function-argument and return-value matching. The types `float` and `double` may be freely matched, as may the types `char`, `short`, `int`, and `unsigned`. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must be the same types as their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are `"="`, initialization, `"=="`, `"!="`, function arguments, and return values.

If you want to turn off strict type-checking for an expression, the comment `/*-NOSTRICT*/` should be added to the source code immediately before the expression. This comment prevents strict type-checking for only the next line in the program.

Type casts

The C type cast feature was introduced as an aid to portability. Consider this assignment, where `p` is a character pointer:

```
p = 1;
```

lint warns you about the assignment. In this assignment, a cast is used to convert the integer to a character pointer:

```
p = (char *)1;
```

The programmer obviously had a strong motivation for doing this and has clearly signaled those intentions. Nevertheless, lint prints a warning message about it.

Non-portable character use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, lint prints messages about certain comparisons and assignments being illegal or non-portable. For example, this code fragment works on one machine but fails on machines where characters always take on positive values:

```
char c;
...
if( (c = getchar( )) < 0 ) ...
```

The real solution is to declare *c* as an integer, because *getchar* is actually returning integer values. In any case, lint will print the message:

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the fields may be too small to hold the values. This is especially true because bit fields on some machines are considered signed quantities. While it may seem logical that a 2-bit field declared type *int* cannot hold the value 3, the problem disappears if the bit field is declared to be type *unsigned*.

Assignments of longs to ints

Bugs can arise from the assignment of a *long* to an *int*, which truncates the contents. This can happen in programs that have been incompletely converted to use *typedefs*. When a *typedef* variable is changed from *int* to *long*, the program can stop working because some intermediate results may be assigned to *ints*, which are truncated. The *-a* option can suppress messages about the assignment of *longs* to *ints*.

Unusual constructions

Several legal but unusual constructions are detected by lint. It is hoped that the messages encourage clearer style, and even point out bugs. The *-h* option can suppress these checks. For example, in the following statement, the *** does nothing.

```
*p++ ;
```

lint returns the message:

```
null effect
```

The following program fragment results in a test that will never succeed:

```
unsigned x ;
if( x < 0 ) ...
```

Similarly, the next two tests are equivalent, although the intended action may differ:

```
if( x > 0 ) ...

if( x != 0 ) ...
```

lint prints the following message in these cases:

```
degenerate unsigned comparison
```

Suppose a program contains something similar to this:

```
if( 1 != 0 ) ...
```

lint prints this message because the comparison of 1 with 0 gives a constant result:

```
constant in conditional context
```

Another construction detected by **lint** involves operator precedence. Bugs that arise from misunderstanding operator precedence can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the following statements probably do not act as intended:

```
if( x&077 == 0 ) ...
```

```
x<<2 + 40
```

The best solution is to use parentheses in such expressions, and **lint** encourages this with an appropriate message.

Multiple uses and side effects

With complicated expressions, the best order in which to evaluate subexpressions may be machine-dependent. For example, on machines in which the stack runs backwards, function arguments are probably best evaluated from right to left. On machines with a stack running forward, left to right evaluation seems most efficient.

Function calls embedded as arguments of other functions may or may not be treated comparably to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

The C language leaves the order of evaluation of complicated expressions up to the local compiler, so that efficiency of C on a particular machine is not unduly compromised. In fact, various C compilers have considerable differences in the order in which they evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example:

```
a[i] = b[i++];
```

This statement causes **lint** to call attention to this condition by printing:

```
warning: i evaluation order undefined
```


Chapter 4

C Programmer's Productivity Tools

Introducing the 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** 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** 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=[y n]	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=[y n]	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=[y n]	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=dirname	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=filename	<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 runs 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]                        sub1();
[12]                )

                sub1()
[15]                (

                /* The following declaration *is* an executable
                statement. */

[20]                int i = 0;
[22]                if (i > 0) {
                        /* The next line is never executed. */
[U] [25]                        sub2();
[U] [26]                }
[U] [27]                else (
[28]                        sub3();
[29]                )
[30]                )

                sub2()
[U] [33]                (

                        /* do nothing */

[U] [35]                )

                sub3()
[38]                (

                        /* do same as sub2() */

[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

23 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 from (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 **-I** 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 covered	lines covered	total lines	function 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 undocumented 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.

Using *lprof* with *rcc*

The examples given so far in this chapter illustrate using **lprof** with **cc**. This tool may be used with **rcc** also. As with **cc**, profiled programs must be compiled with the **-ql** option. All the **lprof** options remain the same. There will, however, be some differences in behavior.

- Values are reported for **goto** labels and **case** labels when using **cc** even though these statements do not correspond to executable code. These values are not meaningful.
- Similarly, values are reported for a line containing only the closing bracket of an **if** statement, and for a line that contains an **else** statement. These values are carry-overs from previous lines.

- The number of times a function was called and exited are marked differently. If `gcc` is being used, the beginning and ending curly braces are marked:

```

                                foo(i)
                                int i;
[6]      {
...
[6]      }
```

If `cc` is used, the first formal parameter declaration and the ending curly brace are marked:

```

                                foo(i)
[6]      int i;
                                {
...
[6]      }
```

- If a **break** or **continue** statement is executed inside a loop, `lprof`, used with `cc`, will mark the closing curly brace as having been executed a smaller number of times than the beginning curly brace, as shown in the next example:

```

                                while(i)
[10]     {
...
                                if (exp) break;
...
[9]      }
```

If `lprof` is used with `gcc`, the beginning and ending curly brace will both be marked the with the actual number of executions of the first curly brace:

```

                                while(i)
[10]     {
...
                                if (exp) break;
...
[10]     }
```

Similarly, functions that use `setjmp()` and `longjmp()` may have their start- and end-points marked with a different number of executions.

These differences occur because of differences in the way `cc` and `gcc` generate code and line number information.

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 4-1 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 the section “Examples of using **cscope**” later in this chapter.

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 4-2 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(nsrcdirs * 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 4-3 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 5

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), documented in the *Programmer's Reference*, 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 the on- and off-line manual pages and the *Programmer's Reference*.

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 interfile 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.s abc.a
2              cc x.o y.o z.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
```

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.s* assembly source 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** 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. (For information about macros, see 'Description files and substitutions' later in this chapter.) 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:

```
cleanup:
    rm *.o
    du
```

If the following command is issued, all object files are deleted.

make cleanup

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 /usr/fred/lib/abc.a
      cc -o menu x.o y.o z.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 end in a *.s*.

By knowing and using the default suffix rules, or by creating new ones, the complexity of the description files can be reduced. For more information on suffixes, see ‘Suffixes and transformation rules’, later in this chapter.

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 metacharacters 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 the backslash 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:

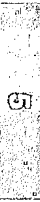
```
2 = xyz
abc = -ll -ly -lm
LIBES =
```

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 is used by **make**. 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.



\$*, **\$@**, **\$?**, and **\$<** are four special macros that change values during the execution of the command. 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 $@
. . .
```

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.

```
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, 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

cleanup:
        -rm *.o gram.c
        -du

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

        # print files that are out-of-date
        # with respect to "print" file.

print: $(FILES)
        pr $? | $(LP)
        touch print

```

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.

Extensions of \$, \$@, and \$<*

A makefile may also use the following related macros: `$(@D)`, `$(@F)`, `$(*D)`, `$(*F)`, `$(<D)`, and `$(<F)`. The internally generated macros `$*`, `$@`, and `$<` are useful generic terms for current targets and out-of-date relatives. The following related macros are also available: `$(@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` command.

For example, the following command can be used:

```
cd $(<D);$(MAKE) $(<F)
```

Executable commands

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
```

In general, any shell command can be used in a makefile, but there are a few 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. For example:

```
test:
    echo "start"
    for i in 1 2 3;do \
        echo "start" \
        echo "starting..."; \
        echo "...done"; \
    done
    echo "end"
```

The middle four lines comprise a single shell command. The backslashes suppress the normal interpretation of a newline, resulting in the four lines being considered a single line.

A further restriction results from the fact that each command line is executed in a separate shell. This does not allow for carry over command executions. Consequently, a `cd` command remains in effect within a single line. For example:

```
target1 : comp1 comp2
        cd newdir
        ls
```

The **ls** command lists the directory where **make** was invoked, not *newdir*. To achieve the desired result, try:

```
target1 : comp1 comp2
        cd newdir; ls
        .
        .
        .
```

In this example, the shell is invoked on the entire command line, and the **cd** remains in effect when the **ls** is executed.

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 there is no such name, **make** prints a message and stops.

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.

Recursive makefiles

Another feature of **make** concerns the environment and recursive invocations. For testing purposes, **make -n ...** can be executed and everything that can be done is printed, including output from lower-level invocations of **make**. For example:

```
make -n
```

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.

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** flag 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 is as follows:

.o	object file
.c	C source file
.c~	SCCS C source file
.f	FORTTRAN source file
.f~	SCCS FORTRAN source file
.s	assembler source file
.s~	SCCS assembler source file
.y	yacc source grammar
.y~	SCCS yacc source grammar
.l	lex source grammar
.l~	SCCS ex source grammar
.h	header file
.h~	SCCS header file
.sh	shell file
.sh~	SCCS shell file

Figure 5-1 summarizes 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.

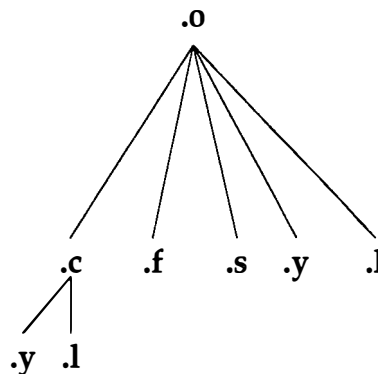


Figure 5-1 Summary of default transformation path

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 5-1.

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**, **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**, **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*, and *.s~.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 are internally defined:

.c~	SCCS C source file
.f~	SCCS Fortran source file
.sh~	SCCS shell file
.c~.a:	transforms an SCCS C source file to a library (archive) file
.c~.c:	transforms an SCCS C source file to a C source file
.c~.o:	transforms an SCCS C source file to an object file
.f~.a:	transforms an SCCS Fortran source file to a library (archive) file
.f~.f:	transforms an SCCS Fortran source file to a Fortran source file
.f~.o:	transforms an SCCS Fortran source file to an object file
.s~.a:	transforms an SCCS assembler source file to a library (archive) file
.s~.s:	transforms an SCCS assembler source file to an assembler source file
.s~.o:	transforms an SCCS assembler source file to an object file
.y~.c:	transforms an SCCS yacc source file to a C source file
.y~.o:	transforms an SCCS yacc source file to an object file
.l~.l:	transforms an SCCS lex source file to a lex source file
.l~.o:	transforms an SCCS lex source file to an object file
.h~.h:	transforms an SCCS C header file to a C header 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~:
.f:
.f~:
.sh:
.sh~:
```

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 .a
```

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:

```
CMDS = 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 **CMDS** is used. For example, the following compiles all the components previously defined in the target (**CMDS**) with a two-line entry in the makefile:

```
$(CMDS) : $$@.c
          $(cc) -o $? -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, then the priority of assignments is rearranged to:

- 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

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.

Internal rules

The standard set of internal rules used by **make** is reproduced in the examples shown on the following pages.

Example 5-1 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
#

MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=

```

Example 5-1 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 5-1 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 $(LDFLAGS)
$(AR) $(ARFLAGS) $@ $*.o
-rm -f $*.o

.f~.a:
$(GET) $(GFLAGS) $<
$(F77) $(F77FLAGS) c $*.f $(LDFLAGS)
$(AR) $(ARFLAGS) $@ $*.o
-rm -f $*.[fo]

```

Example 5-1 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 5-1 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

```

Chapter 6

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 and why changes were made to a file, and the person making them

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. By using this approach, SCCS can maintain successive versions of the file, while consuming only a minimal amount of increased disk space. This allows for the efficient use of resources and easy regeneration of previous versions of a file.

This chapter covers the following topics:

- SCCS for beginners: how to create, retrieve, and update versions of 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 by in chapter.

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 under the 'Delta Numbering' section of this chapter.

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** (administer) 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** command under 'SCCS commands.' 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 g.file name by deleting the prefix *s*. (the file is now called *g.lang*) from the name of the SCCS file. Thus, the original *lang* file has been recreated.

An **ls(C)** command (documented in the *User's Reference*) lists both *lang* and *s.lang* files in the directory. SCCS retains the *s.lang* for other users.

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 **-e** option to the **get** 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** command.

The **get -e** command prints additional status information including 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 (documented in the *User's Reference*). 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 version 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. This evolutionary process is represented in Figure 6-1.

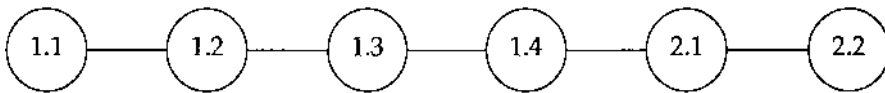


Figure 6-1 Evolution of an SCCS file

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 6-1. 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 6-2:

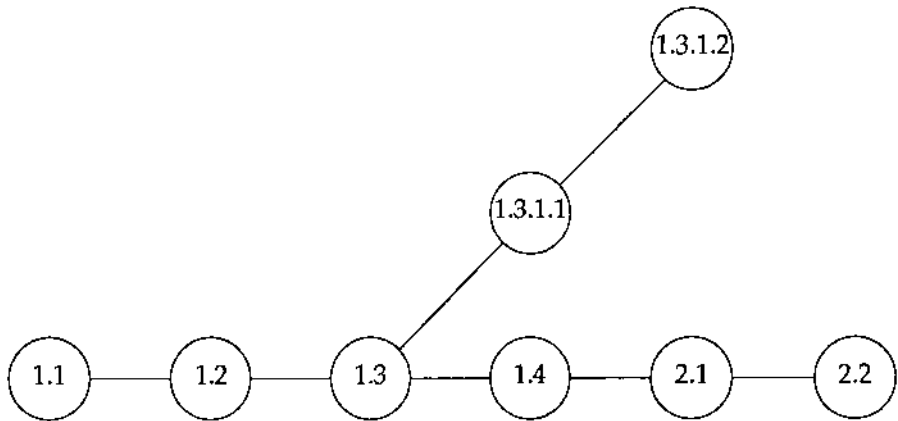


Figure 6-2 Tree structure with branch deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 6-3. 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.

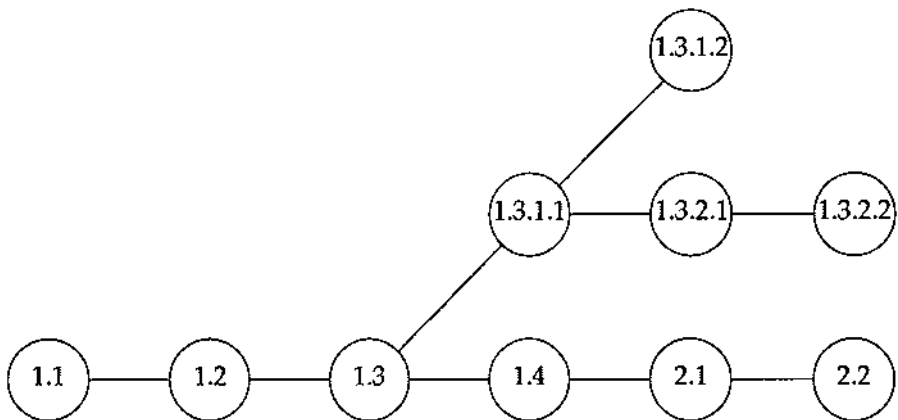


Figure 6-3 Extended branching concept

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, that 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. 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)`, which are also documented in the *User's Reference*.

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' and documented in the *Programmer's Reference*.

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)` in the *Programmer's Reference*.

There is distinction between real user (see `passwd(C)`) 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.

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.* of the SCCS filename with *x.* The x.file is created in the same directory as the SCCS file, given the same mode (see `chmod(C)` in the *User's Reference*), 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. 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 in the *Programmer's Reference*.

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
scsdiff	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** 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.

The help command

The **help** facility provides an explanation of the error messages. For example:

```
get lang
```

This error message is displayed:

```
ERROR [lang]: not an SCCS file (co1)
```

For more information on this error message, use the **help** facility in conjunction with the error code (co1), for example:

```
help co1
```

This provides the following explanation of why *get lang* produced an error message:

```
co1:
"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.

The 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, and the version retrieved, the module's name within the g.file. This information eventually appears 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');
```


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 **-i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately 20 ID keywords provided, see **get(CP)** in the *Programmer's Reference*.

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.

```
get -r1.3 s.abc
```

This command retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3  
64 lines
```

A branch delta can be retrieved similarly:

```
get -r1.5.2.3 s.abc
```

This produces (for example) on the standard output:

```
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.

This command omits the level number:

```
get -r3 s.abc
```

This causes the retrieval of the trunk delta with the highest level number within the given release. The output of the above command might be:

```
3.7  
213 lines
```

If the given 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. For details on numbering by delta's see the 'Delta numbering' section.

```
get -r9 s.abc
```

Executing the above might produce:

```
7.6  
420 lines
```

This indicates that trunk delta 7.6 is the latest version of file *s.abc* below release 9.

get -r4.3.2 s.abc

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.) This might result in the following output:

```
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. For example, if the most recent delta in release 3 is 3.5, and the following command is issued:

get -r3 -t s.abc

It might produce:

```
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
```

Retrieval with intent to make a delta

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 'The **admin** command' 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 flags in the SCCS file, representing start and end of range.

- It checks whether the R is locked against editing. The lock is a flag in the SCCS file.
- It checks whether multiple concurrent edits are allowed for the SCCS file by the **-j** flag in the SCCS file.

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, when there is really no editing that needs to be done at the time. In such cases, the **unget** 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)** in the **Programmer's Reference** 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 **get** and executed with the **-e** key. 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. See 'Protection' under 'SCCS files' for a discussion of how different users are permitted to use SCCS commands on the same files.

Table 6-1 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 6-1 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)
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.(mS+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 6-4:

- * 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** key letter is effective only if the **-b** flag (discussed under **admin**(CP) in the *Programmer's Reference*) is present in the file. An entry of **-** means 'irrelevant'.
- ** The release hR is the highest existing release that is lower than the specified, nonexistent release R.
- ‡ This case applies if the **-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 **-j** flag is set in the SCCS file. 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 (whose format is described under **get(CP)** in the *Programmer's Reference*) showing the deltas used in constructing a particular version of the SCCS file.

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)**, which is documented in the *User's Reference*. 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)** in the *Programmer's Reference* for a full description of additional key letters.

The 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)**, documented in the *User's Reference*, 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, 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 has a **-v** flag, **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)** in the *User's Reference*.)

NOTE The **delta -m** command is allowed only if the SCCS file has a **-v** flag.

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 a **-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 **-i** flag in the SCCS file (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 described earlier under 'SCCS command conventions.' 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)**, documented in the *User's Reference*.

The *admin* command

The **admin**(CP) command, documented in the *Programmer's Reference*, 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' under 'SCCS files' later in this chapter.)

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** flag 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 **-i** flag (not to be confused with the **-i** key letter), 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 **-v** flag must be set by means of **-f**. The **-v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary. (See **sccsfile(F)** in the *Programmer's Reference*.) 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)** in the *Programmer's Reference* 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 -fm modname 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.

The 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)** in the *Programmer's Reference*.

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 the following output:

```
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 the following output:

```
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**.

The 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. It is described in the **Programmer's Reference**.

The rmdel command

The **rmdel**(CP) command, documented in the *Programmer's Reference*, 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 6-3, 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 **-l** flag is set, the release must not be contained in the list. (See **admin**(CP) in the *Programmer's Reference*.) 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).

The cdc command

The **cdc**(CP) command, documented in the *Programmer's Reference*, 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 and deletion of old ("!" prefix) MR numbers. 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 the -v flag has been set.)

```
comments?
```

**deleted wrong MR number and
inserted correct MR number**

NOTE An MR (Modification Request) is described in 'The **delta** command' section of this guide.

The *what* command

The **what**(CP) command, described in the *Programmer's Reference*, 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) in the *Programmer's Reference*.) 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%";
```

When the following command is used, the resulting *g.file* is 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.

The sccsdiff command

The **sccsdiff**(CP) command, documented in the *Programmer's Reference*, 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, documented in the *User's Reference* (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), which is discussed in the *User's Reference*.

The comb command

The **comb**(CP) command, documented in the *Programmer's Reference*, lets the user try to reduce the size of an SCCS file. It generates a shell procedure (see **sh**(C) in the *User's Reference*) 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 6-3, 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 (see `get(CP)` in the *Programmer's Reference* for its syntax) of deltas the user wants preserved. All other deltas are discarded.

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.

The val command

The **val(CP)** command (see the *Programmer's Reference*) 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** flag, respectively. See **admin(CP)** in the *Programmer's Reference* for descriptions of these flags.

The **val** command treats the special argument **-** differently from 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.

The 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. A complete description of **vc** may be found in the *Programmer's Reference*.

SCCS files

The following topics are discussed in this section:

- protection mechanisms used by SCCS files
- format of SCCS files
- recommended 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
- 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 SCCS files do so by creating a copy of the file (see 'SCCS command conventions'). 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)** in the *User's Reference*.) 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	This is a line containing the logical sum of all the characters of the file (not including the checksum itself).
Delta table	This provides information about each delta, such as type, SID, date and time of creation, and commentary.
User names	This is the list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	These are indicators that control certain actions of SCCS commands.
Descriptive text	This is usually a summary of the contents and purpose of the file.
Body	This is the text administered by SCCS, intermixed with internal SCCS control lines.

Details on these file sections can be found in **sccsfile(F)**. The checksum is discussed next under "Auditing."

Since SCCS files are ASCII files, they can be processed by non-SCCS commands like `ed(C)`, `grep(C)`, and `cat(C)`, all documented in the *User's Reference*. 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 Care should be exercised 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, described in the *User's Reference*, 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 7

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,” 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,” describes how to build a shared library. Specifically, this part describes how to use the tool **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.

NOTE Shared libraries are a new feature of UNIX System V Release 3.0 and later. An executable object file that needs shared libraries will not run on previous releases of UNIX System V.

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 ... -llibrary_file ...
```

To direct a search of the networking library, for example, you use the following command line.

```
cc file.c -o file ... -lnsl_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 the section "Referencing Symbols in a Shared Library from Another Shared Library" later in this chapter.

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" later in this section.

Space considerations

The following section outlines the benefits and drawbacks of using a shared library. Covered are:

- 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 between a shared library and an non-shared library.

A host shared library resembles an 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 an non-shared library and a shared library.

Although these similarities exist, a shared library differs significantly from an 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 an 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. (See the section "Importing Symbols" later in the chapter for more details on imported symbols.) 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 7-1 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 7-2 depicts the process images in memory of these two files when they are executed.

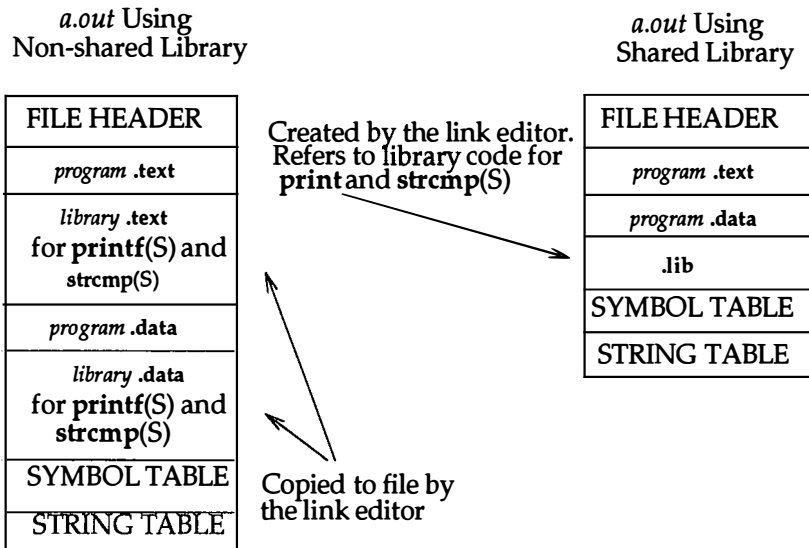


Figure 7-1 *a.out* files created using a non-shared library and a shared library

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 7-2. This separation enables all processes using the same shared library to reference a single copy of the code.

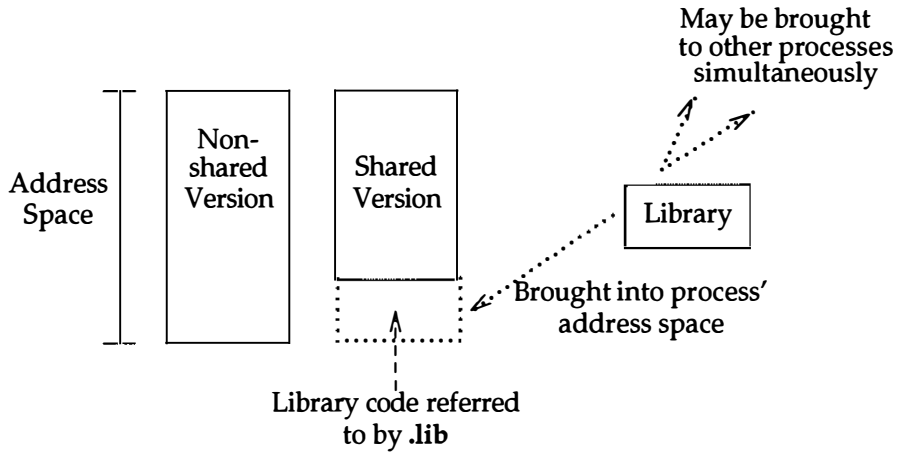


Figure 7-2 Processes using an archive and a shared library

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.

Use the following two points when using either an 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 (documented in the *Programmer's Reference*) 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 are 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 7-1 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 ran 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 7-3 shows two *a.out* files executing a call to `printf(S)`. The process on the left was built using an non-shared library. It already has a copy of the library code defining the `printf(S)` 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.

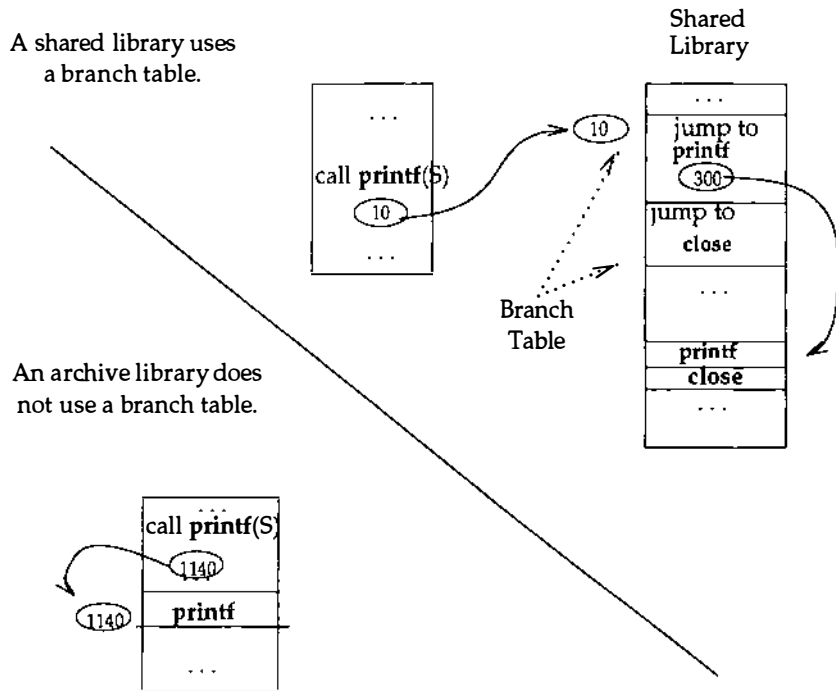


Figure 7-3 A branch table in a shared library

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)."

Building a shared library

This part 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
0xA000000 . . . 0xA3C00000	Reserved for AT&T UNIX System Shared C Library AT&T Networking Library	/shlib/libc_s /shlib/libnsl_s
0xA400000 0xA4400000 0xA4800000 0xA4C00000	Generic Database Library	Unassigned
0xA500000 0xA5400000 0xA5800000 0xA5C00000	Generic Statistical Library	Unassigned
0xA600000 0xA6400000 0xA6800000 0xA6C00000	Generic User Interface Library	Unassigned
0xA700000 0xA7400000 0xA7800000 0xA7C00000	Generic Screen Handling Library	Unassigned
0xA800000 0xA8400000 0xA8800000 0xA8C00000	Generic Graphics Library	Unassigned
0xA900000 0xA9400000 0xA9800000 0xA9C00000	Generic Networking Library	Unassigned
0xAA000000 0xAB000000 . . 0xAFC00000	Scan Code library Libprot library	/shlib/libsc_s /shlib/protlib_s
0xB0000000 . . . 0xBFC00000	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 doesn't 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 doesn't 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 don't 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's a good idea to include large, frequently used routines in a shared library but to exclude smaller routines that aren't used as much. For

general guidelines see the section "Choosing Library Members" in this chapter. Also see the guidelines in the following sections: "Importing Symbols," "Referencing Symbols in a Shared Library from Another Shared Library," and "Tuning the Shared Library Code."

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 the section "Changing Existing Code for the Shared Library" in this chapter.

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 section "An example" for a sample specification file. Also, see the section "Using the Specification File for Compatibility" in this chapter. The contents and format of the specification file are given by the directives list. A description of each directive is provided in the **mkshlib**(CP) manual page. References to these directives are made throughout this chapter. Prior review of the **mkshlib**(CP) manual page 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**. These commands all are documented in the *Programmer's Reference*.

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. (See “Use **#hide** and **#export** to Limit Externally Visible Symbols” in the “Using the Specification File for Compatibility” section for more details).

- 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 (which are documented in the *Programmer’s Reference*) are good examples.

NOTE Since the **printf(S)** family of routines is used frequently, we included **printf(S)** and related routines when we built 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” in the “Tuning the Shared Library Code” section.

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)` (documented in the *Programmer's Reference*), 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 don't 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(S)` 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. You should 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. See "Use `#hide` and `#export` to Limit Externally Visible Symbols" in the section "Using the Specification File for Compatibility" later in this chapter for further tips.

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

```
...
int head = 0;
...
func()
{
    ...
    p = "hello";
    ...
}
...
int tail = 0;
...
```

Broken Successor

```
...
int head = 0;
...
func()
{
    ...
    p = "hello, world";
    ...
}
...
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 (e.g. tables for switch statements).

Adding new external variables to a shared library may change the addresses of static symbols, but this doesn't 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. The exported data symbols should go 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) man page).

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 which 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 7-4, the symbols `_libc.ptr1` and `_libc.ptr2` are imported from user's code and the symbol `_libc_malloc` from the library itself.

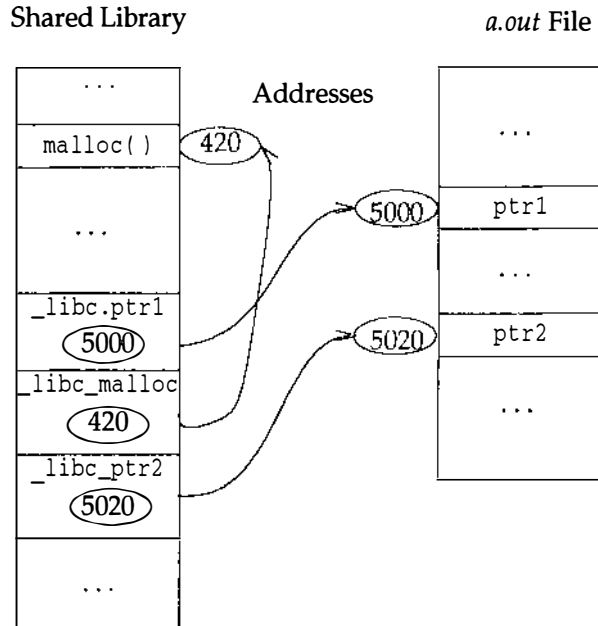


Figure 7-4 Imported symbols in a shared library

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 don't 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 an 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.

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)` has been 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 would be 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 an *a.out* File" 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, we 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, such as UNIX System V Release 3.0.

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

To begin, profile the code that might go into the shared library (see the `prof(CP)` command in the *Programmer's Reference*).

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" earlier in this chapter 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

The key is to 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

Better performance results 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. A command, **chkshlib**(CP) (documented in the *Programmer's Reference*), 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), documented in the *Programmer's Reference*.

* 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	file1 can [may not] execute using file2
file2,3,... (if any) are targets	file1 can [may not] execute using file3
file1 is executable	file1 may [may not] have been produced using file2
file2,3 are hosts	file1 may [may not] have been produced using file3
file1 is host	file1 can [may not] produce executables which will run with file2
file2 (if any) is target	
file1 is target	file2 can [may not] produce executables which will run with file1
file2 is host	
both files are targets or	file1 can [may not] replace file2
both files are hosts	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)`, documented in the *Programmer’s Reference*.

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 choice. Unfortunately, you might not be able to find those *a.out* files, let alone force their owners to rebuild them with your new library.

So your second choice is to give a different target pathname to the new version of the library. The host and target pathnames are independent; you don’t 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:

logd	floating-point logarithm to a given base; defined in the file <i>log.c</i>
polyd	evaluate a polynomial; defined in the file <i>poly.c</i>
maux_stat	return usage counts for the other two routines in a structure; defined in <i>stats.c</i> ,

The external variable:

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 <i>errno</i> in the C library),
----------------	--

And the header file:

<i>maux.h</i>	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.

```

/* 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) {
        mauxerr = 1;
        return(0);
    }
    else
        mauxerr = 0;
    return(logx/logbase);
}

```

```

/* poly.c */

#include "maux.h"
#include <math.h>

/* Evaluate the polynomial
 *       $f(x) := a[0] * (x^n) + a[1] * (x^{n-1}) + \dots + a[n];$ 
 * Note that there are N+1 coefficients!
 * This uses Horner's Method, which is:
 *       $f(x) := (((a[0]*x) + a[1])*x) + a[2]) + \dots) + a[n];$ 
 * 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);
}

```



```

/* 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);
}

/* 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

To begin, 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    0xB0a0000

```

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" of 'The building process' earlier in this chapter. 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, we 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.

We 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, we need to define the imported symbol pointers somewhere. We have already created a file for global data `maux_defs.c`, so we will add the definitions to it.

```
/* Data fileiaux_defs.c */

intiauxerr = 0;
double (*_libiaux_log)() = 0;
char * (*_libiaux_malloc)() = 0;
```

Finally, we observe that there are floating-point operations in the code, and we remember that the routines for these cannot be imported. (If we tried to write the specification file and build the shared library without taking this into account, **mkshlib** would give us errors about unresolved references.) This means we 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 **libiaux**:

```
1  ##
2  ## libiaux.sl - libiaux specification lfile
3  #address .text 0xB0680000
4  #address .data 0xB06a0000
5  #target /my/directory/libiaux_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     _libiaux_malloc malloc
22     _libiaux_log    log
```

Briefly, here 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, we have to compile the **.o** files as we would for any other library:

```
cc -ciaux_defs.c poly.c log.c stats.c
```

Next, we 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** will have to 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**, you would issue the following command line:

```
cc x.c libmaux_s.a -lm -lc_s
```


This command line causes the following:

- the imported symbol pointer reference to `log` is resolved from `libm`
- the 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.

Summary

This chapter describes the UNIX System shared libraries and explains how to use them. It also explains how to build your own shared libraries. Using any shared library almost always saves disk storage space, memory, and computer power; and running the UNIX System on smaller machines makes the efficient use of these resources increasingly important. Therefore, you should normally use a shared library whenever it's available.

Chapter 8

lex

The software tool *lex* 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 8-1. 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.

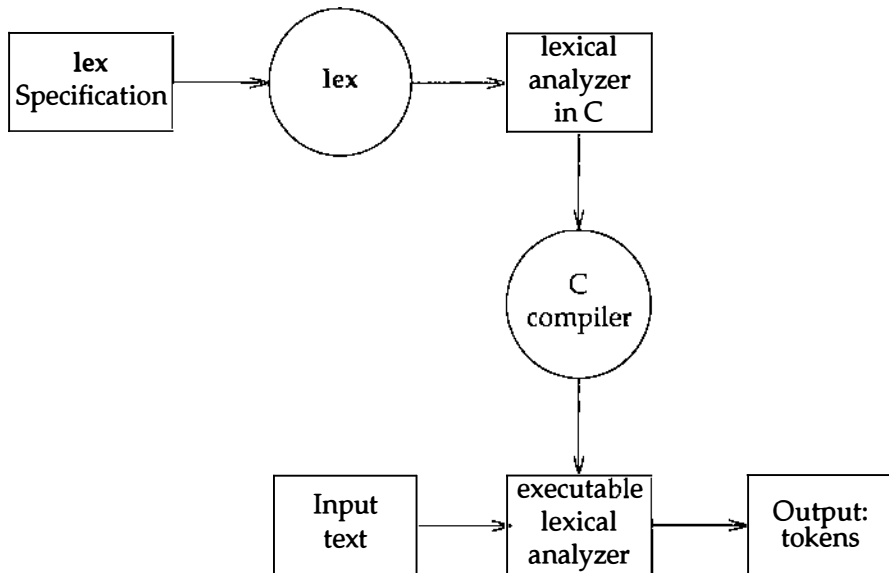


Figure 8-1 Creation and use of a lexical analyzer with `lex`

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.

The 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` and `sed`. 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:

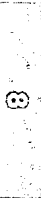
```
[^abc]
```

- 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`.

The 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`. `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}      printf("%s", yytext);
```

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 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:

```
DO50k = 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 example that follows, assume that you are dealing with a programming language in which programs start with the keyword **#go** and end with the keyword **#stop**, and which has a 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, if any, state 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 the section on "Disambiguating rules".

Rule five (line 10) in the specification deactivates the state `PROC` 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[yytextlen-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[yytext-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 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,

```
BiiiBjjjB
```

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. If the input text consists of the “word” *BoomBoxB*, then 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.

yylless

The function **yylless()** resets the end point of the current token. **yylless()** takes a single integer argument: **yylless(*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 **yylless(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

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`. `lex` can conveniently be used with `yacc` to develop compilers. (For a more complete discussion, see the chapter on `yacc` in this guide.) Whether or not you plan to use `lex` with `yacc`, this section 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 be ended 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 **yylval** should be used for this purpose. Parsers generated by **yacc** look in that variable for the value of the token. **yylval** 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 which 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 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 the 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 500. 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 700
```

This entry tells *lex* to make the table large enough to handle as many as 700 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 2000, the designated parameter is *a*, for example:

```
%a 2800
```

You may refer to the *Programmer's Reference* page on *lex* for a list of all the options available with the *lex* command.

Compilation

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.

Execution

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` 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.c` (or if `x.c` 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`. (See the chapter on `make` in this guide for more information.)

Chapter 9

yacc

yacc is a “parser generator”; It generates C code that can be compiled to produce a parser. A parser 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 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 ;
```


The symbols to the right of the colon will have been defined as tokens, defined in other rules in the specification, or else will be 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 semi-colon serve as punctuation in the rule and have no significance in evaluating the input. Input such as the following would be matched by this rule:

April 16, 1961

This chapter discusses the following topics:

- preparing a yacc specification
- the parser operation
- ambiguities
- operator precedences in arithmetic expressions
- error detection and recovery
- the operating environment and special features of the parsers yacc produces
- suggestions to improve the style and efficiency of the specifications
- examples of yacc usage

Basic specifications

A yacc 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 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 following example is a complete yacc specification:

```
%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 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.

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 the chapter on lex in this guide.

NOTE While the lexical analyzer may be included as a routine defined the specification file, it would be 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.

The rules section

This section describes the grammar rules and actions that appear in the rules section of a yacc specification.

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 some 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 (discussed later in this chapter) 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. Uppercase 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 escapes are recognized. The following escapes are understood by yacc.

```

'\n'    newline
'\r'    return
'\''    single quote (')
'\'\'    backslash (\)
'\t'    tab
'\b'    backspace
'\f'    form feed
'\nnn'  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 and thereby avoid rewriting 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 will be 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 specif-

ication. 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 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 that define 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 doing input and output, calling subroutines, modifying variables, and so on.

The following are examples of grammar rules with actions:

```
A      :  '(' B  ')'
```

```
      {
```

```
        hello( 1, "abc" );
```

```
      }
```

```
      ;
```

and:

```
XXX    :  YYY ZZZ
```

```
      {
```

```
        (void) 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. In the case of 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 on the left of the colon, and will be that symbol's value when it appears on 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 ')' ;
```

One would expect the value returned by this rule to be 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 ;
}
```

The 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 came 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 via 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 which 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 );
          }
```

The declarations section

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 the section “Precedence.”

Every name that appears in the rules section, but is not defined in the declarations section, is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

The 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 value 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 `yyval` 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 nonterminal 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 will be delayed until the section "Precedence".

The subroutines section

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.

Lexical analysis

The user must supply a lexical analyzer to read the input stream and pass tokens to the parser. This analyzer may also have 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 `yylex` 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` knows 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 the earlier section “Support for arbitrary value types.”

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 yyval

The `lex` 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 by calling a function `yylex`.

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);
        }

```

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 that has 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 nonnegative 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. The potential for duplication exists here, so care must be taken to make sure that all token numbers are distinct.

The end marker

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.



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 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 triviality 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*, *codeview*, or *dbxtra*).

Compiling and running the parser

This section explains how to create and run the parser once yacc is run to produce the C code.

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.c*, or if *x.c* 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 which yacc uses to go from the specification to the C code for the parser is complex and will not be discussed here. The parser itself is relatively simple and understanding how it works will make treatment of error recovery and ambiguities easier. (Understanding this section is not essential to being able to use yacc, but it will likely prove helpful.)

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**, **accept**, **error**, and **goto**. 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.

The 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 *yylval* is pushed on to the value stack.

The reduce action

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 on 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 will be 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.

The 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;
           }
           ;
```

The 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 will be 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 `yyperror()` 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 that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user rather 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 which do not have 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 which 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 to resolve 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 to allow 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 followed the specification just given would structure 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, though 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.

The 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 is 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 most likely unacceptable, 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 the action after **error** were to call some sophisticated resynchronization routine that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token

returned by the lexical analyzer would presumably be the first token in a legal statement. The old illegal token would have to be discarded and the **error** state reset. An action similar to the following could 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 is, consider writing the sequence specification using an empty rule.

```
seq : /* empty */
     | seq item
     ;
```

Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if the parser is asked to decide between empty sequences that could satisfy more than one rule, if 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

This section contains two extended examples. They display many of the features of yacc discussed in this chapter.

A 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, */
    /* yynval = 0 through 25 */
    /* returns DIGIT for digit, yynval = 0 through 9 */
    /* all other characters are returned immediately */
    int c;
        /*skip blanks*/
    while ((c = getchar()) == ' ')
        ;

        /* c is now nonblank */

    if (islower(c))
    {
        yynval = c - 'a';
        return (LETTER);
    }
    if (isdigit(c))
    {
        yynval = c - '0';
        return (DIGIT);
    }

    return (c);
}

```

An 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 10

m4: A macro processor

The **m4**(CP) macro processor allows the user to define and process strings of characters called *macros*. Macros are strings in a file which get replaced by some other text when the file is processed. The substituted text will often be another literal string. Macros can also be defined in terms of a set of programming constructs that **m4** supports, and so macros may act as functions. These functions can:

- accept arguments
- perform arithmetic
- check conditions
- perform file manipulation
- do string processing

The **m4** macro processor may be used to enhance the functionality of some other application, such as a compiler or word processor.

The basic operation of **m4** is to copy its input to its output. As the input is read, each alphanumeric string (that is, string of letters and digits) is checked. If the string is the name of a macro, the name is replaced by its definition. **m4** then rereads the resulting string and may perform further manipulations on it. When macros are called with arguments, the arguments are collected and substituted in the correct position in the defining text before **m4** rescans the text.

The **m4** macro processor provides a collection of about twenty built-in macros. In addition, the user can define new macros. This chapter describes some of the most commonly used built-in macros and explains how you can define your own macros. Built-in and user-defined macros work the same way, except that some of the built-in macros have side effects on the state of the process. For more information about the built-in macros, see **m4**(CP).

Invoking m4

To invoke **m4**, use a command of the form:

```
m4 [filenames]
```

Filename arguments are processed in order. If there are no arguments, or if a dash (-) appears in the argument list, then the standard input is read. The processed text is written to the standard output, and can be redirected as shown by the following command:

```
m4 file1 file2 - > outputfile
```

The dash in the above example indicates that the standard input is processed after *file1* and *file2*.

Defining macros

The built-in function **define** is used to define new macros. The following statement defines the name **name** as **contents**:

```
define(name,contents)
```

All subsequent occurrences of **name** will be replaced by **contents**. **name** must be alphanumeric and must begin with a letter. (The underscore (_) is regarded as a letter.) The string **contents** could be any text, including text that contains balanced parentheses, and it may stretch over multiple lines. Here is a short example of a file which is to be processed by **m4**:

```
define(Bob,Robert)
define(Sandy,Sandra)
define(I,James)
define(Julia,her mother)
Bob and I had lunch with Sandy and Julia.
```

If this file were processed by **m4**, the output would be:

```
Robert and James had lunch with Sandra and her mother.
```

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis it is assumed to have no arguments.

A macro name is only recognized as such if it is surrounded by non-alphanumeric characters. A slightly modified version of the previous example will demonstrate this:

```
define(Bob,Robert)
define(Sandy,Sandra)
define(I,James)
define(Julia,her mother)
Bob and Ian had lunch with Sandy and Julia.
```

The output will be:

```
Robert and Ian had lunch with Sandra and her mother.
```

Notice that the letter “I” in the word “Ian” is not replaced by the string “James”. This is because one of the characters surrounding “I” is an alphanumeric character.

Macro names or arguments can also be defined in terms of other names or arguments. The following statements define macros M and N each to be 100:

```
define(N, 100)
define(M, N)
```

Since N is defined to be 100, and M is defined to be N, M is defined to be 100. This illustrates two aspects of **m4**’s behavior:

- macro names can be expanded into their defining text at any time after they are defined
- **m4** expands macro names into their defining text at the first opportunity

The first fact implies that the definition of N may be used anywhere in the text that follows it, even inside another **define** statement. The second fact implies that the second **define** statement above is equivalent to:

```
define(M,100)
```

As soon as **m4** reads the name N, it substitutes the text ‘100’. Therefore, even if N subsequently changes, M does not.

If this is not the desired result, it may be avoided by interchanging the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now M is defined as the string “N”, so when M is read later, **m4** will first replace it with the string “N”. Then, because **m4** rereads substituted text, the name N will be replaced by ‘100’. The net effect is that M will be replaced by whatever the value of N is at that time.

Quoting

The more general solution to the problem described in the last section is to delay the expansion of the arguments of **define**. This is done by *quoting* them. Any text surrounded by a left single quotation mark (') and a right single quotation mark (') is not expanded immediately, but has the marks stripped off. In the second statement below, the punctuation marks are removed from the string "N" as the argument is being collected, but the *macro* N is not expanded. The effect is that M is defined as the string "N", not as 100:

```
define(N, 100)
define(M, 'N')
```

The general rule is that **m4** strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. Given the definitions in the last example, if you want the letter "M" to pass through **m4** unchanged and appear in the output, it must be quoted. For example:

```
"Come in, Mr Bond," 'M' said.
```

The following definitions show another application of quoting:

```
define(N, 100)
.
.
.
define(N, 200)
```

The "N" in the second definition is evaluated as soon as it is seen: that is, it is replaced by 100. This makes the second definition equivalent to:

```
define(100, 200)
```

Obviously, this statement doesn't have the desired effect. Moreover, it will be ignored by **m4**, since only names (alphanumeric strings starting with a letter) can be defined. To really redefine N, evaluation must be delayed by quoting, as shown below:

```
define(N, 100)
.
.
.
define('N', 200)
```

When using **m4**, it is often wise to quote the first argument of a define statement.

Changing the quoting marks

If the single quote and grave marks used for quoting are not convenient for some reason, the quoting marks can be changed with the built-in function **changequote**. For example, the following statement defines the new quotation marks to be the left and right square brackets:

```
changequote([, ])
```

The original characters can be restored by calling **changequote** with no arguments.

undefine

The built-in function **undefine** removes the definition of a macro or built-in function. The following statement removes the definition of **N**:

```
undefine('N')
```

undefine, as in the following statement:

```
undefine('define')
```

Notice that the macros and functions being undefined must be quoted. Once a built-in function is removed, its value cannot be retrieved.

ifdef

The built-in function **ifdef** determines whether a macro is currently defined. This function can take three arguments, but the third argument may be null. The first argument is a macro name, the second and third are strings (which may contain macros). If the macro given as the first argument is defined, the function returns the second argument (that is, the function call is replaced by the text of the argument). If the macro is undefined, the third argument is returned. If the third argument is null, the function returns a null string.

As an example, suppose that one of the macros **xenix** and **unix** is defined in a particular version of a program. To define a macro **system** the system being used, you could use the following statements:

```
ifdef('xenix', 'define(system,1)' )
ifdef('unix', 'define(system,2)' )
```

Again, note the quotation marks in this example.

Here is an example of an **ifdef** call with three arguments:

```
ifdef('xenix', on XENIX, not on XENIX)
```

Using arguments

The simplest form of macro definition involves replacing a string with another fixed string. User-defined macros can also have arguments, so different invocations can have different results. Within the replacement text for a macro, any occurrence of `$n` will be replaced by the *n*th argument when the macro is actually used. For example, the `replicate` macro, defined in the following example, generates code to double its argument. It uses the `eval` macro, explained in the section “Using built-in arithmetic functions”.

```
define(replicate, eval($1+$1))
Six of one and half-a-dozen of the other gives replicate(6)
```

When `m4` processes the input file above, the output is:

```
Six of one and half-a-dozen of the other gives 12
```

Only the first nine macro arguments, `$1` through `$9`, are accessible. The argument `$0` is the macro name itself. Arguments which are referred to in a macro definition but not supplied when the macro is called are replaced by null strings. The following macro simply concatenates up to nine arguments:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

It may be called with fewer than nine arguments, as in the call:

```
cat(x, y, z)
```

The result of this call will be the string `xyz`. The arguments `$4` through `$9` are null, since no corresponding arguments were provided.

Leading unquoted spaces, tabs, and newlines that occur during argument collection are discarded. All other white space is retained. The following statement defines “a” to be *b c*:

```
define(a, b c)
```

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. Therefore, in the statement below there are only two arguments: the second is literally `( b,c)`.

```
define(a, (b,c))
```

A literal comma or parenthesis can be inserted in an argument list by quoting it.

Using built-in arithmetic functions

The **m4** processor provides built-in functions for doing arithmetic on integers. There are two simple functions, **incr**, which returns the result of incrementing its numeric argument by 1, and **decr** which returns the result of decrementing by 1. To perform the equivalent of the assignment statement '**M=N+1**', use the following statements:

```
define(N, 100)
define(M, 'incr(N)')
```

Note that **incr(N)** is not equivalent to '**N=N+1**', that is, it does not actually change the value of **N**.

The more general mechanism for arithmetic is the built-in function **eval**, which supports ordinary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

Precedence	Operator	Name
1	+	Arithmetic Addition
	-	Arithmetic Negation
2	**	Exponential
	^	Exponential
3	*	Multiplication
	/	Division
	%	Remainder
4	+	Addition
	-	Subtraction
5	= =	Equal
	!=	Not Equal
	<	Less than
	<=	Less than or Equal to
	>	Greater than
	>=	Greater than or Equal to
6	!	Logical NOT
7	&	Logical AND
	&&	Logical AND
8		Logical OR
		Logical OR

Parentheses can be used to group operations where needed. All the operands of an expression given to **eval** must ultimately evaluate to numbers. The numeric value of a relation that evaluates to *true*, for example, '**1>0**', is 1; the value of a relation which evaluates to *false* is 0. The precision of **eval** depends on the implementation.

To set the value of the macro **M** to $2^{**}N+1$, use the following statements:

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

It is advisable to quote the defining text for a macro unless it is very simple (for example, just a number).

eval may appear on its own in the input. For example:

```
2+2=eval(2+2)
```

Manipulating files

A new file can be included in the input at any time by using the built-in function **include**. The following example inserts the contents of the file *boiler* into the file being processed:

```
include(boiler)
```

The **m4** processor replaces the function call with the contents of the file. Files containing a set of **define** statements are often included in this way.

It is a fatal error if **include** cannot access the file named. To avoid such errors, the alternate form **sinclude** can be used. If **sinclude** (silent include) cannot find a file, no messages are issued and processing continues.

The output of **m4** can be diverted to files during processing and the collected material output later. For this purpose, **m4** maintains ten output streams, called *diversions*. These are numbered 0 through 9, and the files diverted into are referred to by their diversion number. Diversion 0 is the standard output. The following statement puts all subsequent output at the end of a temporary file referred to as 1:

```
divert(1)
```

Diverting to this file is stopped by another **divert** macro. A **divert** call which has an argument in the range 2 through 9 will stop diverting to the current file and start diverting to another file; a call which has no arguments or an argument of 0 will cause output to be sent to the standard output. Diverting to a diversion whose name is not between 0 and 9, inclusive, causes the diverted text to be discarded.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion. This is done using the **undivert** macro. Diversion *n* may be brought back with a call of the form

```
undivert(n)
```

The macro call is replaced by the contents of the diversion. **undivert** can take multiple arguments, separated by commas; the diversions are brought back in the order specified in the argument list. The following macro call will bring back all the diversions, in numeric order:

```
undivert
```

When a diversion is “undiverted”, it is emptied of all text.

The built-in **divnum** function returns the number of the currently active diversion. This value is 0 during normal processing.

Using system commands

Any program on the local operating system can be run from a file being processed by **m4** by way of the built-in **syscmd** macro. For example, the following statement runs the UNIX System **date** command:

```
syscmd(date)
```

The macro **syscmd** can be used to create a file which is subsequently included using the **include** command.

Using conditionals

There is a built-in macro called **ifelse** that enables conditional testing. The behavior of this macro is somewhat unusual, so the description and examples below should be read carefully. **ifelse** takes three or more arguments. In the simplest case, where only three arguments are present, the values of the first two arguments are compared, and if they are the same string, then the macro returns the value of the third argument. For example:

```
define(Bill,William)
ifelse(William,Bill,"Same guy--book him!")
```

In this example, the first two arguments are the same after macro substitution takes place, so the third argument will be output. If the first two arguments are not the same, then a null string is returned.

If exactly four arguments are present, then if arguments one and two are not identical, the fourth argument is returned. For example, you could define a macro called **compare** that compares two strings and returns 1 if they are the same, and 0 if they are not:

```
define(compare, 'ifelse($1, $2, 1, 0)')
```

Note the quotation marks, which prevent the immediate evaluation of **ifelse**. If the fourth argument is missing, it is treated as empty.

If there are exactly five arguments to `ifelse`, the fifth argument is ignored. If there are more than five arguments, and if the first two arguments are not identical, then the result will be as if `ifelse` were called with the fourth and remaining arguments. For example:

```
ifelse(a,A,1,b,b,2)
```

This expression will return 2. *a* and *A* are not identical, so the result is the same as if the following call had been made:

```
ifelse(b,b,2)
```

Since the first two arguments are the same in this case, the value 2 is returned. This method of evaluation works with even longer argument lists. Consider this example:

```
ifelse(a,A,1,b,B,2,c,c,3,d)
```

Since the first two arguments are not the same, the result is the same as that of the macro call:

```
ifelse(b,B,2,c,c,3,d)
```

Again, the first two arguments are not the same, and so the result of this macro call is now the same as the result of:

```
ifelse(c,c,3,d)
```

This macro call will return 3. The `ifelse` macro thus provides a limited form of multiple-decision capability.

Manipulating strings

A number of `m4` built-in macros for performing string manipulation are discussed in this section.

len

The built-in `len` macro takes a string as its argument and returns its length. The following statement will return a value of 6:

```
len(abcdef)
```

All characters within the parentheses are counted, so the following statement will return a value of 5:

```
len((a,b))
```

substr

The built-in **substr** macro returns substrings of strings. The macro call has the form:

```
substr(s,i,n)
```

This call returns the substring of *s* that starts at position *i* and is *n* characters long. The first character in the string is considered to be in position zero. The following example returns the string "hor ses":

```
substr('toti ehor sesto',6,7)
```

If *n* is omitted, the remainder of the string will be returned.

index

The built-in macro **index** takes two strings as its arguments and returns the starting position at which the second string appears in the first string. It returns -1 if the second string does not appear in the first. The following macro call returns "2":

```
index('toti ehor sesto','ti e')
```

As with the **substr** macro, the first character of a string is in position zero.

translit

The built-in **translit** macro performs character transliteration. The macro call takes the form:

```
translit(s,f,t)
```

The arguments *s*, *f*, and *t* are all strings; the macro returns a string gotten by replacing all characters in *s* that appear in *f* with the corresponding character in *t*. The following macro call replaces each of the letters in *aeiou* with the corresponding digit:

```
translit(questionable, aeiou, 12345)
```

The returned value will be *q52st34n1bl2*. If *t* is shorter than *f*, characters that don't have an entry in *t* are replaced by the null string. *f* are replaced with the null string. Therefore, the following statement deletes vowels from the first argument:

```
translit(questionable, aeiou)
```


Printing

The built-in macro **errprint** writes its arguments to the standard error. The statement below to print the message "fatal error":

```
errprint('fatal error')
```

The **dumpdef** macro is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, all definitions are printed. Otherwise, the definitions named as arguments are printed.

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. The standard is detailed in ANSI Document Number X3J11/90-013. A list of the characteristics and extensions that must be documented is provided in an appendix to the ANSI document, subsection “F.3 Implementation-Defined Behaviour”. This list serves as a guide to documenting implementation-defined behaviour. Note that it is not an exhaustive list but a guide provided as a convenience.

According to the ANSI standard, each implementation of an ANSI-conforming implementation of the C language must be accompanied by a document that defines all implementation-defined characteristics and extensions. This appendix provides this information for the Microsoft C Compiler, version 6. It 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

The compiler generates diagnostic messages in the following form:

filename (line-number) : diagnostic level error source error number message

filename and *line-number* indicate the location of the offending statement.

diagnostic level indicates whether the message is an error or a warning.

error source indicates which program in the compilation process generated the error as follows:

- C a compiler error
- D a command line error

The leading digit of *error number* indicates the severity of the message as follows:

- 1 a fatal error
- 2 a non-fatal error
- 4 a warning

message is an explanation of the error or warning.

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 full pathname of the program being executed.

argv[1] through **argv[c]** 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 Microsoft C Compiler recognizes up to 254 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 Microsoft C Compiler recognizes up to 254 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 Microsoft C compiler, the source and execution character sets are the same, the standard 8-bit character set.

Multi-byte shift states

Multi-byte characters are used by some C language implementations to represent foreign-language characters not represented in the standard ASCII set. The Microsoft C compiler supports multi-byte characters but does not implement shift states; only the “C” locale is implemented. Consequently, no useful operations can be performed on multi-byte characters.

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 nongraphic 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

Escape sequence	ASCII value	Character Name
\'	39	single quote
\"	34	double quote
\?	63	question mark
\\	92	backslash
\a	7	bell
\b	8	backspace
\f	12	formfeed
\n	10	new-line
\r	13	carriage return
\t	9	horizontal tab
\v	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 left side. For example, if

```
int i = 'abcd';
```

then the value of `i` in hex is 65666768, the individual values for "a", "b", "c", and "d" concatenated together. Again, if

```
int ii = 'abcdef';
```

then the value of `ii` in hex is 67686970, the individual values for "c", "d", "e", and "f" concatenated together. Note that the excess characters on the left, "a" and "b", 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 “C6.0 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) 0xabcdfe;
```

then the value of `c` is `0xfe`.

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 Microsoft C Compiler are:

`float` 4 byte IEEE format

`double` 8 byte IEEE format

`long double` 8 byte IEEE format

|| **NOTE** The section “C6.0 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

The compiler will place the first 3 variables declared with the **register** storage-class specifier into registers as long as the variable is accessed and it is of the correct type. Three registers in the i386 are used for register variables: EBX, ESI, and EDI. These registers can hold values of types **long**, **int**, and **short**.

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 6.0 compiler aligns the whole structure on a 4-byte boundary by default (see “Pragmas” below). 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

```
zzyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxx
44444444333333332222222211111111
```

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. 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 on 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, normally */usr/include*. The **-X** option disables the search through the default path.

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, normally */usr/include*. The **-X** option disables the search through the default path. 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 file name.

Pragmas

The behavior of each recognized **#pragma** directive in C 6.0 is listed below:

alloc_text

Places the specified **functions** in the given text segment.

```
#pragma alloc_text(text_segment, function [, function]...)
```

comment

Places the comment **string** into the comment segment of the object file.

```
#pragma comment(comment_type, "string")
```

comment_type is one of: **compiler**, **lib**, **exestr**, or **user**.

code_seg

Allocates a segment named **segment** in **_TEXT** with class name **name1name2....**

```
#pragma code_seg("segment", "name1" [, "name2"]...)
```

check_stack

Tells the compiler to turn stack probes **on** or **off**. The default is **off**.

```
#pragma check_stack(on|off)
```

check_pointer

Turns pointer checking **on** or **off**. The default is **off**.

```
#pragma check_pointer(on|off|)
```

This pragma is recognized, but performs no useful function under UNIX.

data_seg

Allocates a segment named "segment" in `_DATA` with class name `name1name2...`

```
#pragma data_seg("segment", "name1" [, "name2"]...)
```

function

Tells the compiler to generate code for function calls instead of using the inline (intrinsic) form of the **function(s)** listed.

```
#pragma function(function1[,function2]...)
```

intrinsic

Tells the compiler to generate inline code for the listed **function(s)**.

```
#pragma intrinsic(name)
```

loop_opt

Turns loop optimization on and off.

```
#pragma loop_opt(on|off)
```

The default is off. Optimization can be toggled for the complete compilation using the `-Ol` option.

message

Print "string" on the standard output.

```
#pragma message("string")
```

optimize

If **string** is empty, the compiler uses default values. If **on** or **off** is omitted, the setting for the given options are toggled.

```
#pragma optimize("string", on|off)
```

string is one or more of:

- a** Assume no aliases in the following code.
- c** Enable local common subexpression.
- e** Enable global register allocation.
- g** Enable global common subexpressions.
- l** Optimize loops.
- n** Disable unsafe loop optimizations.
- p** Ensure floating-point consistency.

- t** Optimize for speed.
- w** Assume no aliases except across function calls.

plmn

This pragma is recognized but performs no useful function under UNIX. **on** or **off** toggles last setting.

```
#pragma plmn(on|off)
```

plmf

This pragma is recognized but performs no useful function under UNIX. **on** or **off** toggles last setting.

```
#pragma plmf(on|off)
```

pagesize

Sets the number of lines on the source listing page to a number greater than 15 and less than or equal to 255. The default is 63.

```
#pragma pagesize(integer)
```

linesize

Sets the listing line size in a source listing to a number greater than 79 and less than or equal to 132. The default is 79.

```
#pragma linesize(integer)
```

page

When compiler is generating source listing, generate **integer** number of blank pages. The number of blank pages may be greater than or equal to 1 and less than or equal to 127. The default is to generate one page.

```
#pragma page(integer)
```

skip

When compiler is generating source listing, generate **integer** number of blank lines. The number of blank lines may be greater than or equal to 1 and less than or equal to 127. The default is to generate one blank line.

```
#pragma skip(integer)
```

pack

Instructs the compiler to pack structs on 1, 2, 4, or 8 byte boundary (defaults to 4).

```
#pragma pack(1|2|4|8)
```

switch_check

Controls whether or not code is generated to perform range checking on switch expressions.

```
#pragma switch_check(on|off|)
```

on or off toggles the last setting.

segment

Allocates a segment named **identifier** in one of **_TEXT**, **_DATA**, **_BSS**, **CONST**.

```
#pragma segment(_TEXT|_DATA|_BSS|CONST, identifier, identifier, ...)
```

searchlib

This pragma is recognized, but performs no useful function under UNIX.

```
#pragma searchlib ...
```

same_seg

Tells compiler to assume the listed "far" variables are allocated in the same segment. Must be used with the **-ND** command line option.

```
#pragma same_seg(variable1, variable2,...)
```

subtitle

Sets a subtitle, **string**, on the source listing page.

```
#pragma subtitle("string")
```

title

Sets a title, **string**, on the source listing page.

```
#pragma title("string")
```

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,

```
((void *) 0);
```

Diagnostic printed by the `assert` function

The `assert` function prints the diagnostic

```
"Assertion failed: <expression>, <file name>, <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`, `isctrl`, `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. The values returned by the mathematics functions on the occurrence of domain errors are presented in the following table:

Table A-2

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`.

`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	Number	Function
SIGINT	02[0]	interrupt
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGABRT	06[0]	used by abort, replaces SIG10T
SIGFPE	08[1]	floating point exception
SIGSEGV	11[1]	segmentation violation
SIGTERM	15[0]	software termination signal

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.

Writing on text stream

Writing to a text stream causes the associated file to be truncated beyond that point.

File buffering

Files are line 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 file names

The rules for composing valid file names are:

- The maximum file name length is 14 characters under SCO UNIX 3.2.2.
- The maximum file name length is 255 characters under SCO UNIX 3.2.4.
- All ASCII characters are acceptable with the exception of `"/"`, `"."` and `".."`.

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.

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 macro `errno` is set to 2 by the `fgetpos` or `ftell` function on failure. A 9 or a 22 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. This is different behaviour from the behaviour of the memory allocation functions in `libmalloc.a` which always 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.

Environment names

The environment list is altered by the function `putenv`. `putenv("string");` where `string` is of the format: `variable=value`.

NOTE Since the underlying operating system is POSIX and X/OPEN compatible, the size of the environment is determined by a manifest defined by POSIX and X/OPEN: `ARG-MAX`.

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:

0. Error 0
1. Not owner
2. No such file or directory
3. No such process
4. Interrupted system call
5. I/O error
6. No such device or address
7. Arg list too long
8. Exec format error
9. Bad file number
10. No child processes
11. No more processes
12. Not enough space
13. Permission denied
14. Bad address
15. Block device required
16. Device busy
17. File exists

18. Cross-device link
19. No such device
20. Not a directory
21. Is a directory
22. Invalid argument
23. File table overflow
24. Too many open files
25. Not a typewriter
26. Text file busy
27. File too large
28. No space left on device
29. Illegal seek
30. Read-only file system
31. Too many links
32. Broken pipe
33. Argument out of domain
34. Result too large
35. No message of desired type
36. Identifier removed
37. Channel number out of range
38. Level 2 not synchronized
39. Level 3 halted
40. Level 3 reset
41. Link number out of range
42. Protocol driver not attached
43. No CSI structure available
44. Level 2 halted
45. Deadlock situation detected/avoided
46. No record locks available
47. Error 47
48. Error 48
49. Error 49
50. Bad exchange descriptor
51. Bad request descriptor
52. Message tables full
53. Anode table overflow
54. Bad request code
55. Invalid slot
56. File locking deadlock
57. Bad font file format
58. Error 58
59. Error 59
60. Not a stream device

61. No data available
62. Timer expired
63. Out of stream resources
64. Machine is not on the network
65. Package not installed
66. Object is remote
67. Link has been severed
68. Advertise error
69. Srmount error
70. Communication error on send
71. Protocol error
72. Error 72
73. Error 73
74. Multihop attempted
75. Error 75
76. Error 76
77. Not a data message
78. Filename too long
79. Error 79
80. Name not unique on network
81. File descriptor in bad state
82. Remote address changed
83. Can not access a needed shared library
84. Accessing a corrupted shared library
85. .lib section in a.out corrupted
86. Attempting to link in more shared libraries than system limit
87. Can not exec a shared library directly
88. Error 88
89. Function not implemented
90. Error 90
91. Error 91
92. Error 92
93. Error 93
94. Error 94
95. Error 95
96. Error 96
97. Error 97
98. Error 98
99. Error 99
100. Error 100
101. Error 101
102. Error 102
103. Error 103

- 104. Error 104
- 105. Error 105
- 106. Error 106
- 107. Error 107
- 108. Error 108
- 109. Error 109
- 110. Error 110
- 111. Error 111
- 112. Error 112
- 113. Error 113
- 114. Error 114
- 115. Error 115
- 116. Error 116
- 117. Error 117
- 118. Error 118
- 119. Error 119
- 120. Error 120
- 121. Error 121
- 122. Error 122
- 123. Error 123
- 124. Error 124
- 125. Error 125
- 126. Error 126
- 127. Error 127
- 128. Error 128
- 129. Error 129
- 130. Error 130
- 131. Error 131
- 132. Error 132
- 133. Error 133
- 134. Error 134
- 135. Structure needs cleaning
- 136. Error 136
- 137. Not a name file
- 138. Not available
- 139. Is a name file
- 140. remote i/o error
- 141. reserved for future use
- 142. Error 142
- 143. Error 143
- 144. Error 144
- 145. Directory not empty

Errors 33 and 34 are available as the manifest constants `EDOM` and `ERANGE` respectively.

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.

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 behaviour 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-4

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-5

Function	Case Mapping Functions	
	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.

C6.0 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-6 limits.h

Variable	Limit
CHAR_BIT	8
CHAR_MAX	SCHAR_MAX
CHAR_MIN	SCHAR_MIN
INT_MAX	2147483647
INT_MIN	-2147483648
LONG_MAX	2147483647L
LONG_MIN	-2147483648L
MB_LEN_MAX	1
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-7 float.h

Variable	Limit
FLT_RADIX	2
FLT_ROUNDS	1
FLT_MANT_DIG	24
DBL_MANT_DIG	53
LDBL_MANT_DIG	64
FLT_DIG	6
DBL_DIG	15
LDBL_DIG	19
FLT_EPSILON	((float)1.192092896e-07)
DBL_EPSILON	2.2204460492503131e-16
LDBL_EPSILON	5.4210108624275221706e-20L
FLT_MIN_EXP	(-125)
DBL_MIN_EXP	(-1021)
LDBL_MIN_EXP	-16381
DBL_MIN	2.2250738585072014e-308
FLT_MIN	((float)1.17549435e-38)
LDBL_MIN	3.3621031431120935063e-4932L
FLT_MIN_10_EXP	(-37)
DBL_MIN_10_EXP	(-307)
LDBL_MIN_10_EXP	(-4931)
FLT_MAX_EXP	128
DBL_MAX_EXP	1024
LDBL_MAX_EXP	16384
FLT_MAX	((float)3.40282347e+38)
DBL_MAX	1.7976931348623147e+308
LDBL_MAX	1.189731495357231765e4932L
FLT_MAX_10_EXP	38
DBL_MAX_10_EXP	308
LDBL_MAX_10_EXP	4932

Translation limits

This section describes the translation limits defined for this environment. The following translation limits are defined for this implementation:

Table A-8

Item	Actual	Minimum Acceptable
nesting levels of compound statements, iteration control structures, and selection control structures.	30	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	254	31
significant initial characters in an external identifier	254	6
external identifiers in one translation unit	1021	511
identifiers with block scope declared in one block	1024	127
macro identifiers simultaneously defined in one translation unit	≥ 4096	1024
parameters in one function definition	31	31
arguments in one function call	31	31
parameters in one macro definition	127	31
arguments in one macro invocation	127	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 #include'd files	9	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	15	15
functions that can be registered by atexit().	32	32

Appendix B

Compiler exit codes and error messages

This appendix describes the exit codes returned by the compiler. It also lists error messages you may encounter as you develop a program, and gives a brief description of actions you can take to correct the errors.

Compiler exit codes

All the programs in the C Compiler package return an exit code (sometimes called an “errorlevel” code) that can be used by other programs such as **make**. If the program finishes without errors, it returns a code of 0. The code returned varies depending on the error encountered.

Code Meaning

- | | |
|---|---|
| 0 | No fatal error |
| 2 | Program error (such as compiler error) |
| 4 | System level error (such as out of disk space or compiler internal error) |

Command-line error messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

<i>command line fatal error D1xxx: messagetext</i>	(fatal error)
<i>command line error D2xxx: messagetext</i>	(error)
<i>command line warning D4xxx: messagetext</i>	(warning)

If possible, the compiler continues operation, printing a warning message. In some cases, command-line errors are fatal and the compiler terminates processing.

Command-line fatal-error messages

The following messages identify fatal errors. The compiler driver cannot recover from a fatal error; it terminates after printing the error message.

D1000 UNKNOWN COMMAND LINE FATAL ERROR

The compiler detected an unknown fatal-error condition.

D1001 could not execute *filename*

The compiler could not find the given file in the current working directory or any of the other directories named in the PATH variable.

D1002 too many open files, cannot redirect *filename*

No more file descriptors were available to redirect the output of the -P option to a file.

D1003 no more processes, try later

Can't fork any more processes. Process table is full.

Command-line error messages

When the compiler driver encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

D2000 UNKNOWN COMMAND LINE ERROR

The compiler detected an unknown error condition.

D2001 too many symbols predefined with -D

Too many symbolic constants were defined using the -D option on the command line.

The limit on command-line definitions is normally 16; you can use the -U or -u option to increase the limit to 200.

D2002 a previously defined model specification has been overridden

Two different memory models were specified; the model specified later on the command line was used.

D2003 missing source file name

You did not give the name of the source file to be compiled.

D2007 bad *option* flag, would overwrite *string1* with *string2*

The specified *option* was given more than once, with conflicting arguments *string1* and *string2*.

D2008 too many *option* flags, *string*

Too many letters were given with the specified *option* (for example, with the -O *option*).

- D2009 unknown option *character* in *optionstring*
One of the letters in the given option was not recognized.
- D2012 too many linker flags on command line
You tried to pass more than 128 separate options and object files to the linker.
- D2013 incomplete model specification
Not enough characters were given for the *-Astring* option. The *-Astring* option requires all three letters (to specify the data-pointer size, code-pointer size, and segment setup).
- D2014 *-ND* not allowed with *-Ad*
You cannot rename the default data segment unless you give the *-Auxx* option (*SS != DS*, load *DS*) on the command line.
- D2016 *-Gw* and *-ND name* are incompatible
You tried to rename the default data segment to the given *name* when you specified the *-Gw* option. Renaming the default data segment is illegal in this case because the *-Gw* option requires the *-Awxx* option.
- D2017 *-Gw* and *-Au* flags are incompatible
You tried to specify the *-Auxx* option (*SS != DS*, load *DS*) with the *-Gw* option. Specifying *-Auxx* with *-Gw* is illegal because the *-Gw* option requires the *-Awxx* option.
- D2019 cannot overwrite the source file, *name*
You specified the source file as an output-file *name*. The compiler does not allow the source file to be overwritten by one of the compiler output files.
- D2020 *-Gc* option requires extended keywords to be enabled (*-Ze*)
The *-Gc* option and the *-Za* option were specified on the same command line. The *-Gc* option requires the extended keyword *cdecl* to be enabled if library functions are to be accessible.
- D2021 invalid numerical argument *string*
A non-numerical *string* was specified following an option that required a numerical argument.
- D2022 cannot open help file, *cc.hlp*
The *-help* option was given, but the file containing the help messages (*cc.hlp*) was not in the default directory (*/usr/lib/286*) or in any of the directories specified by the *PATH* environment variable.
- D2024 : *-Gm* and *-ND* are incompatible options
You compiled with both the *-Gm* and *-ND* compiler options. These options are incompatible because *-Gm* indicates that string literals and *near const* data items should be allocated in the *CONST* segment, while the *-ND* option attempts to allocate the same items in a different, named segment.

Command-line warning messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking.

D4000 UNKNOWN COMMAND LINE WARNING

An unknown warning has been detected by the compiler.

D4001 listing has precedence over assembly output

Two different listing options were chosen; the assembly listing is not created.

D4002 ignoring unknown flag *string*

One of the options given on the command line was not recognized and is ignored.

D4003 80186/286 selected over 8086 for code generation

Both the **-G0** option and either the **-G1** or **-G2** option were given; **-G1** or **-G2** takes precedence.

D4004 optimizing for time over space

This message confirms that the **-Ot** option is used for optimizing.

D4006 only one of **-P/-E/-EP** allowed, **-P** selected

Only one preprocessor output option can be specified at one time.

D4007 **-C** ignored (must also specify **-P** or **-E** or **-EP**)

The **-C** option must be used in conjunction with one of the preprocessor output flags, **-E**, **-EP**, or **-P**.

D4008 non-standard model -- defaulting to small model libraries

A nonstandard memory model was specified with the **-Astring** option. The library search records in the object model were set to use the small-model libraries.

D4009 threshold only for far/huge data, ignored

The **-Gt** option cannot be used in memory models that have near data pointers. It can be used only in compact, large, and huge models.

D4011 preprocessing overrides source listing

Only a preprocessor listing was generated, since the compiler cannot generate both a source listing and a preprocessor listing at the same time.

D4012 function declarations override source listing

The compiler cannot generate both a source-listing file and the function prototype declarations at the same time.

D4013 combined listing has precedence over object listing

When **-Fc** is specified along with either **-Fl** or **-Fa**, the combined listing (**-Fc**) is created.

- D4014 invalid value *number* for *string*. Default *number* is used
 An invalid value was given in a context where a particular numerical value was expected.
- D4017 conflicting stack checking options - stack checking disabled
 Both the **-Ge** and the **-Gs** flags are given in one compile command (**-Ge** enables stack checking, **-Gs** disables it).

Compiler error messages

The error messages produced by the Microsoft C compiler fall into three categories:

1. Fatal-error messages
2. Compilation-error messages
3. Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the file name and line number where the error occurs.

Fatal-error messages

Fatal-error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename(line) : fatal error C1xxx: messagetext

After the compiler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Compilation-error messages

Compilation-error messages identify actual program errors. These messages appear in the following format:

filename(line) : error C2xxx: messagetext

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename(line) : warning C4xxx: messagetext

You can use the **-W** option to control the level of warnings that the compiler generates. This option is described in the “Compiling and linking C language programs” chapter of this guide.

Fatal-error messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

C1000 UNKNOWN FATAL ERROR Contact SCO Technical Support

An unknown error condition has been detected by the compiler.

C1001 Internal Compiler Error (compiler file *filename*, line *n*) Contact SCO Technical Support

The compiler detected an internal inconsistency. The *filename* refers to an internal compiler file, not your source file.

C1002 out of heap space in Pass 2

The compiler has run out of dynamic memory space. This usually means that your program has many symbols and/or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into subexpressions.

C1003 error count exceeds *n* ; stopping compilation

Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.

C1004 unexpected end-of-file found

This message appears when you have insufficient disk space for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file. This message can also occur when a comment does not have a closing delimiter (**/*), or when an *#if* directive occurs without a corresponding closing *#endif* directive.

C1005 string too big for buffer

A string in a compiler intermediate file overflowed a buffer.

- C1006 write error on compiler-generated file
The compiler was unable to create the intermediate files used in the compilation process.
The following conditions commonly cause this error:
1. A system file or the inode table is full at time of compilation
 2. Not enough space on a device containing a compiler intermediate file
- C1007 unrecognized flag *string* in *option*
The *string* in the command-line *option* was not a valid option.
- C1008 no input file specified
The compiler found no input file on the command-line.
- C1009 compiler limit : macros nested too deeply
The expansion of a macro exceeds the available space. Check to see if the macro is recursively defined, or if the expanded text is too large.
- C1010 compiler limit : macro expansion too big
The expansion of a macro exceeds the available space.
- C1011 compiler limit : macro definition too big
The expanded macro definition is larger than the internal buffer size used to save it. This size is currently: (24*1024) - 5.
- C1012 unmatched parenthesis - missing *character*
The parentheses in a preprocessor directive were not matched; *character* is either a left or right parenthesis.
- C1013 cannot open source file *filename*
The given file either did not exist, could not be opened, or was not found. Make sure your environment settings are valid and that you have given the correct path name for the file.
- C1014 too many include files
Nesting of **#include** directives exceeds 10 levels.
- C1016 **#if[n]def** expected an identifier
You must specify an identifier with the **#ifdef** and **#ifndef** directives.
- C1017 invalid integer constant expression
The expression in an **#if** directive must evaluate to a constant.
- C1018 unexpected **#elif**
The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.
- C1019 unexpected **#else**
The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

- C1020 unexpected `#endif`
An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.
- C1021 invalid preprocessor command *string*
The characters following the number sign (#) do not form a valid preprocessor directive.
- C1022 expected `#endif`
An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.
- C1023 cannot open source file *filename*
Unable to open *filename* for reading. See open(S) for possible reasons as to why the open might fail.
- C1024 cannot open include file *filename*
Cannot find or open include file *filename* in the list of search directories.
- C1026 parser stack overflow, please simplify your program
Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, try to simplify your program.
- C1027 DGROUP data allocation exceeds 64K
More than 64K of variables was allocated to the default data segment. For compact-, medium-, large-, or huge-model programs, use the **-Gt** option to move items into separate segments.
- C1032 cannot open object listing file *filename*
One of the following statements about the file name or path name given (*filename*) is true:
1. The given name is not valid.
 2. The file with the given name cannot be opened for lack of space.
 3. A read-only file with the given name already exists.
- C1033 cannot open assembly-language output file *filename*
One of the conditions listed under error message C1032 prevents the given file from being opened.
- C1035 expression too complex, please simplify
The compiler cannot generate the code for a complex expression. Break the expression into simpler subexpressions and recompile.
- C1036 cannot open source listing file *filename*
One of the conditions listed under error message C1032 prevents the given file from being opened.

- C1037 cannot open object file *filename*
One of the conditions listed under error message C1032 prevents the given file from being opened.
- C1039 unrecoverable heap overflow in Pass 3
The post-optimizer compiler pass overflowed the heap and could not continue. Try recompiling with the **-Od** option (see “Compiling and linking C language programs”) or try rewriting the function containing the line that caused the error.
- C1040 unexpected end-of-file in source file *filename*
The compiler detected an unexpected end-of-file condition while creating a source listing or mingled source/object listing. This error probably occurred because the source file was edited during compilation.
- C1041 cannot open compiler intermediate file - no more files
The compiler could not create intermediate files used in the compilation process because no more file descriptors were available.
- C1042 cannot open compiler intermediate file - no such file or directory
The compiler could not create intermediate files used in the compilation process because the */tmp* directory did not exist.
- C1043 cannot open compiler intermediate file
The compiler could not create intermediate files used in the compilation process. The exact reason is unknown.
- C1044 out of disk space for compiler intermediate file
The compiler could not create intermediate files used in the compilation process because no more space was available. To correct the problem, make more space available on the disk and recompile.
- C1045 floating point overflow
The compiler generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:


```
float fp_val = 1.0e100;
```

In this example, the double-precision constant **1.0e100** exceeds the maximum allowable value for a floating-point data item.
- C1046 compiler limit: *struct/union* nested too deeply
The *struct/union* is nested too deeply. Try simplifying the definition.
- C1046 bad *option flag*, would overwrite *flag* with *string*
A string *option* is being set, but setting *flag* with *string* would overwrite *flag*.
- C1047 too many *option* flags, *string*
The *option* appeared too many times. The *string* contains the occurrence of the option that caused the error.

- C1047 limit of -D/-I exceeded at *string*
The -D/-I options appeared too many times. The *string* contains the occurrence of the option that caused the error.
- C1048 Unknown option *character* in *optionstring*
The *character* was not a valid letter for *optionstring*.
- C1049 invalid numerical argument *string*
A numerical argument was expected instead of *string*.
- C1050 *segmentname* code segment too large
A code segment grew to within 36 bytes of 64K during compilation. A 36-byte pad is used because of a bug in some 80286 chips that can cause programs to exhibit strange behavior when, among other conditions, the size of a code segment is within 36 bytes of 64K.
- C1052 compiler limit : #if/#ifdef blocks nested too deeply
You have exceeded the maximum nesting level for #if/#ifdef directives.
- C1053 compiler limit : struct/union nested too deeply
Structure and union definitions were nested to more than 10 levels.
- C1054 compiler limit : initializers nested too deeply
The compiler limit on nesting of initializers was exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.
- C1055 compiler limit : out of keys
The maximum number of keys was exceeded.
- C1056 compiler limit : out of macro expansion space
The compiler has overflowed an internal buffer during the expansion of a macro; reduce the complexity of the macro.
- C1057 unexpected end-of-file in macro expansion; (missing ?)
The compiler has encountered the end of the source file while gathering the arguments of a macro invocation. Usually this is the result of a missing closing parenthesis () on the macro invocation.
- C1059 compiler is out of near heap space
The compiler has run out of storage for items that it stores in the near (default data segment) heap. This usually means that your program has too many symbols or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.

- C1060 compiler is out of far heap space
The compiler has run out of storage for items that it stores in the **far** heap. Usually this is the result of too many symbols in the symbol table.
- C1064 compiler limit : token overflowed internal buffer
You defined more than 10 distinct text segments with the **alloc_text** pragma.
- C1068 cannot open file *stdout*
Can't open the standard output file for writing. Check if current directory is read-only or if *stdout* file exists and is read only.
- C1070 mismatched **#if/#endif** pair in file *filename*
Mismatched **#if/#endif** pair in file *filename*.
- C1071 unexpected end-of-file found in comment
An unterminated comment was found.
- C1072 *filename* :cannot read file
Some problem occurred while trying to read the file. See read(S).
- C1090 *namedata* allocation exceeds 64K
Segment *name* is bigger than 64K.
- C1100 invalid number: *num*, in **#line** preprocessing directive
The **linenumber** is negative or too big.
- C1126 *name* :automatic allocation exceeds %s
There are more than *num* bytes of automatic (local) variables in the code.
- C1127 *Segment* :segment redefinition
Segment *name* was redefined.
- C1501 compiler limit : too many temporary variables
This is an internal limit. Ran out of temporary variables.

Compilation-Error messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

- C2000 UNKNOWN ERROR Contact SCO Technical Support
The compiler detected an unknown error condition.
- C2001 newline in constant
A new-line character in a character or string constant was not in the correct escape-sequence format (**\n**).

- C2003 expected defined id
The identifier to be checked in an **#if** directive was not enclosed in parentheses.
- C2004 expected defined(id)
An **#if** directive caused a syntax error.
- C2005 #line expected a line number, found *token*
A **#line** directive lacked the required line-number specification.
- C2006 #include expected a filename, found *token*
An **#include** directive lacked the required filename specification.
- C2007 #define syntax
A **#define** directive caused a syntax error.
- C2008 *character* : unexpected in macro definition
The given character was used incorrectly in a macro definition.
- C2009 reuse of macro formal *identifier*
The given identifier was used twice in the formal-parameter list of a macro definition.
- C2010 *character* : unexpected in macro formal-parameter list
The given character was used incorrectly in the formal-parameter list of a macro definition.
- C2011 *identifier* : *identifier* type redefinition
The given macro definitions exceeded 256 bytes.
- C2012 missing name following <
An **#include** directive lacked the required filename specification.
- C2013 missing >
The closing angle bracket (>) was missing from an **#include** directive.
- C2014 preprocessor command must start as first non-white-space
Non-white-space characters appear before the number sign (#) of a preprocessor directive on the same line.
- C2015 too many characters in constant
A character constant containing more than one character or escape sequence was used.
- C2016 no closing single quotation mark
A character constant was not enclosed in single quotation marks.
- C2017 illegal escape sequence
The character or characters after the escape character (\) did not form a valid escape sequence.

- C2018 unknown character *0xcharacter*
The given hexadecimal number does not correspond to a character.
- C2019 expected preprocessor command, found *character*
The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.
- C2020 *member*: *keyword* member redefinition
Member *member* was redefined. Classification is *keyword*.
- C2021 expected exponent value, not *character*
The given character was used as the exponent of a floating-point constant but was not a valid number.
- C2022 *number*: too big for char
The *number* was too large to be represented as a character.
- C2023 divide by 0
The second operand in a division operation (/) evaluated to zero, giving undefined results.
- C2024 mod by 0
The second operand in a remainder operation (%) evaluated to zero, giving undefined results.
- C2025 *identifier*: enum/struct/union type redefinition
The given *identifier* had already been used for an enumeration, structure, or union tag.
- C2026 *identifier*: member of enum redefinition
The given *identifier* had already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.
- C2027 use of undefined type *type*
The member *type* was not defined.
- C2028 struct/union member must be inside a struct/union
Structure and union members must be declared within the structure or union. This error may be caused by an enumeration declaration that contains a declaration of a structure member, as in the following example:

```
enum a {
    january,
    february,
    int march;    /* structure declaration:
                  ** illegal
                  */
};
```

- C2030 *identifier* : struct/union member redefinition
The *identifier* was used for more than one member of the same structure or union.
- C2031 *identifier* : function cannot be struct/union member
The given function was declared to be a member of a structure. To correct this error, use a pointer to the function instead.
- C2032 *identifier*:function cannot be member of *keyword tagname*
Member/fields cannot be functions. Use a pointer to function instead.
- C2033 *identifier* : bit field cannot have indirection
The given bit field was declared as a pointer (*), which is not allowed.
- C2034 *identifier* : type of bit field too small for number of bits
The number of bits specified in the bit field declaration exceeded the number of bits in the given base type.
- C2035 struct/union *identifier* : unknown size
The given structure or union had an undefined size.
- C2036 *struct/union identifier keyword* : unknown size
The given *identifier* has an unknown size.
- C2037 left of *character* specifies undefined struct/union *identifier*
The expression before the member-selection operator (-> or .) identified a structure or union type that was not defined.
- C2038 *identifier* : not struct/union member
The given identifier was used in a context that required a structure or union member.
- C2039 *identifier* : not *keyword* member
The *identifier* is not a struct/union member.
- C2041 illegal digit *digit* for base *base*
A number of given *base* has an illegal digit in it.
- C2042 signed/unsigned keywords mutually exclusive
The signed and unsigned keywords may not appear in the same declaration.
- C2043 illegal break
A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.
- C2044 illegal continue
A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.
- C2045 *identifier* : label redefined
The given label appeared before more than one statement in the same function.

- C2046 illegal case
The **case** keyword may appear only within a **switch** statement.
- C2047 illegal default
The **default** keyword may appear only within a **switch** statement.
- C2048 more than one default
A **switch** statement contained more than one **default** label.
- C2049 case value *number* already used
A duplicate case value was used.
- C2050 nonintegral switch expression
A switch expression was not integral.
- C2051 case expression not constant
Case expressions must be integral constants.
- C2052 case expression not integral
Case expressions must be integral constants.
- C2054 expected (to follow *identifier*
The context requires parentheses after the function *identifier*.
- C2055 expected formal-parameter list, not a type list
An argument-type list appeared in a function definition instead of a formal-parameter list.
- C2056 illegal expression
An expression was illegal because of a previous error. (The previous error may not have produced an error message.)
- C2057 expected constant expression
The context requires a constant expression.
- C2058 constant expression is not integral
The context requires an integral constant expression.
- C2059 syntax error: *token*
The given *token* caused a syntax error.
- C2060 syntax error : end-of-file found
The end of the file was encountered unexpectedly, causing a syntax error. This error can be caused by a missing closing curly brace (}) at the end of your program.
- C2061 syntax error : identifier *identifier*
The given *identifier* caused a syntax error.
- C2062 type *type* unexpected
The given *type* was misused.
- C2063 *identifier* : not a function
The given *identifier* was not declared as a function, but an attempt was made to use it as a function.

- C2064 term does not evaluate to a function
An attempt was made to call a function through an expression that did not evaluate to a function pointer.
- C2065 *identifier* : undefined
The given *identifier* was not defined.
- C2066 cast to function type is illegal
An object was cast to a function type.
- C2067 cast to array type is illegal
An object was cast to an array type.
- C2068 illegal cast
A type used in a cast operation was not a legal type.
- C2069 cast of void term to nonvoid
The void type was cast to a different type.
- C2070 illegal sizeof operand
The operand of a **sizeof** expression was not an identifier or a type name.
- C2071 *class* : illegal storage class
The given storage *class* cannot be used in this context.
- C2072 *identifier* : initialization of a function
An attempt was made to initialize a function.
- C2075 *identifier* : array initialization needs curly braces
The braces (**{}**) around the given array initializer were missing.
- C2076 *identifier* : struct/union initialization needs curly braces
The braces (**{}**) around the given structure or union initializer were missing.
- C2077 nonscalar field initializer *identifier*
An attempt was made to initialize a bit field member of a structure with a non-integral value.
- C2078 too many initializers
The number of initializers exceeded the number of objects to be initialized.
- C2079 *expression* uses undefined struct/union
The given identifier was declared as a structure or union type that had not been defined.
- C2082 redefinition of formal parameter *identifier*
A formal parameter to a function was redeclared within the function body.
- C2083 *keyword* comparison illegal
Cannot compare structs/unions.

C2084 function *identifier* already has a body

The given function had already been defined.

C2085 *identifier* : not in formal-parameter list

The given parameter was declared in a function definition for a nonexistent formal parameter.

C2086 *identifier* : redefinition

The given *identifier* was defined more than once.

C2087 *identifier* : missing subscript

The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension, as in the following example:

```
int func(a)
    char a[10][];          /* Illegal */
    {
        .
        .
        .
    }
```

```
int func(a)
    char a[][5];           /* Legal */
    {
        .
        .
        .
    }
```

C2088 *op* : illegal for *struct/union*

This is an illegal operation for a *struct/union*.

C2089 *tagname* : *struct/union* too large

The *struct/union* is larger than the maximum allowable size.

C2090 function returns array

A function cannot return an array. (It can return a pointer to an array.)

C2091 function returns function

A function cannot return a function. (It can return a pointer to a function.)

C2092 array element type cannot be function

Arrays of functions are not allowed; however, arrays of pointers to functions are allowed.

C2093 cannot use address of automatic variable as static initializer

You cannot use the address of an automatic variable in the initializer of a static item.

- C2094 label *identifier* was undefined
The function did not contain a statement labeled with the given *identifier*.
- C2095 *function*: actual has type void: parameter *number*
An attempt was made to pass a void argument to a function. Formal parameters and arguments to functions cannot have type void; they can, however, have type void * (pointer to void).
- C2096 struct/union comparison illegal
You cannot compare two structures or unions. (You can, however, compare individual members within structures and unions.)
- C2097 illegal initialization
An attempt was made to initialize a variable using a nonconstant value.
- C2098 nonaddress expression
An attempt was made to initialize an item that was not an lvalue.
- C2099 nonconstant offset
An initializer used a nonconstant offset.
- C2100 illegal indirection
The indirection operator (***) was applied to a nonpointer value.
- C2101 & on constant
The address-of operator (*&*) did not have an lvalue as its operand.
- C2102 & requires lvalue
The address-of operator must be applied to an lvalue expression.
- C2103 & on register variable
An attempt was made to take the address of a register variable.
- C2104 & on bit field ignored
An attempt was made to take the address of a bit field.
- C2105 *operator* needs lvalue
The given *operator* did not have an lvalue operand.
- C2106 *operator* : left operand must be lvalue
The left operand of the given *operator* was not an lvalue.
- C2107 illegal index, indirection not allowed
A subscript was applied to an expression that did not evaluate to a pointer.
- C2108 nonintegral index
A nonintegral expression was used in an array subscript.
- C2109 subscript on nonarray
A subscript was used on a variable that was not an array.

- C2110 pointer + pointer
An attempt was made to add one pointer to another.
- C2111 pointer + nonintegral value
An attempt was made to add a nonintegral value to a pointer.
- C2112 illegal pointer subtraction
An attempt was made to subtract pointers that did not point to the same type.
- C2113 pointer subtracted from nonpointer
Cannot subtract pointer from int.
- C2114 *operator* : pointer on left; needs integral right
The left operand of the given *operator* was a pointer; the right operand must be an integral value.
- C2115 *identifier* : incompatible types
An expression contained incompatible types.
- C2117 *operator* : illegal for struct/union
Structure and union type values are not allowed with the given *operator*.
- C2118 negative subscript
A value defining an array size was negative.
- C2119 typedefs both define indirection
Two **typedef** types were used to declare an item and both **typedef** types had indirection. For example, the declaration of **p** in the following example is illegal:
- ```
typedef int *P_INT;
typedef short *P_SHORT;
/* this declaration is illegal */
P_SHORT P_INT p;
```
- C2120 void illegal with all types  
The **void** type was used in a declaration with another type.
- C2121 *op* : bad *left/right* operand  
Bad operand.
- C2124 divide or mod by zero  
Attempting to divide or mod by zero.
- C2125 *identifier* : allocation exceeds 64K  
The given item exceeds the size limit of 64K. The only items that are allowed to exceed 64K are huge arrays.
- C2127 parameter allocation exceeds 32K  
The storage space required for the parameters to a function exceeded the limit of 32K.

- C2128 *identifier* : huge array cannot be aligned to segment boundary  
**The given array violated one of the restrictions imposed on huge arrays; see the “Memory models” section of the manual page for more information on these restrictions.**
- C2129 static function *identifier* not found  
**A forward reference was made to a static function that was never defined.**
- C2130 #line expected a string containing the file name, found *token*  
**A file name was missing from a #line directive.**
- C2131 more than one memory attribute  
**More than one near, far, or huge attribute was applied to an item, as in the following example:**
- ```
typedef int near NINT;
NINT far a;          /* Illegal */
```
- C2132 syntax error : unexpected identifier
An identifier appeared in a syntactically illegal context.
- C2133 *identifier* : unknown size
An attempt was made to declare an unsized array as local variable, as in the following example:
- ```
int mat_add(array1)
int array1[]; /* Legal */
{
int array2[]; /* Illegal */
.
.
.
}
```
- C2134 *identifier* : struct/union too large  
**The size of a structure or union exceeded the compiler limit (2<sup>32</sup> bytes). This limit is 64K on 80286 systems.**
- C2136 *symbol* : prototype must have parameter types  
**Wrong kind of formals for prototype.**
- C2137 empty character constant  
**An illegal character constant was used.**
- C2139 type following *type* is illegal  
**An illegal *type* combination such as the following was used:**
- ```
long char a;
```

C2140 argument cannot be function type

A function was declared as a formal parameter of another function, as in the following example:

```
int func1(a)
    int a ( );    /* Illegal */
```

C2141 value out of range for enum constant

An enumeration constant had a value outside the range of values allowed for type `int`.

C2143 syntax error : missing *token1* before *token2*

The compiler expected *token1* to appear before *token2*. This message may appear if a required closing curly brace `}`, right parenthesis `)`, or semicolon `;` is missing.

C2144 syntax error : missing *token* before type *type*

The compiler expected the given token to appear before the given *type* name. This message may appear if a required closing curly brace `}`, right parenthesis `)`, or semicolon `;` is missing.

C2145 syntax error : missing *token* before identifier

The compiler expected the given token to appear before an identifier. This message may appear if a semicolon `;` does not appear after the last declaration of a block.

C2146 syntax error : missing *token* before identifier *identifier*

The compiler expected the given *token* to appear before the given *identifier*.

C2147 unknown size

An attempt was made to increment an index or pointer to an array whose base type has not yet been declared.

C2148 array too large

An array exceeded the maximum legal size (2^{32} bytes).

C2149 *identifier* : named bit field cannot have 0 width

The given named bit field had a zero width. Only unnamed bit fields are allowed to have zero width.

C2150 *identifier* : bit field must have type `int`, signed `int`, or unsigned `int`

The ANSI C standard requires bit fields to have types of `int`, **signed `int`**, or **unsigned `int`**. This message appears only if you compiled your program with the `-Za` option.

C2151 more than one language attribute specified

More than one keyword specifying a function-calling convention was given.

- C2152 *identifier* : pointers to functions with different attributes
An attempt was made to assign a pointer to a function declared with one calling convention (**cdecl**, **fortran**, or **pascal**) to a pointer to a function declared with a different calling convention.
- C2153 hex constants must have at least 1 hex digit
At least one hexadecimal digit must follow the "x". The hexadecimal constants 0x and 0X are illegal.
- C2154 *name* : does not refer to a segment name
The *name* was the first identifier given in an **alloc_text** pragma argument list and it is already defined as something other than a segment name.
- C2156 pragma must be outside function
Certain pragmas must be specified at a global level, outside a function body, and there is an occurrence of one of these pragmas within a function.
- C2157 *name* : must be declared before use in pragma list
The function *name* in the list of functions for an **alloc_text** pragma has not been declared prior to being referenced in the list.
- C2158 *name* : is a function
name was specified in the list of variables in a **same_segment** pragma, but was previously declared as a function.
- C2159 more than one storage class specified
Illegal declaration - only one storage class is allowed.
- C2160 **##** cannot occur at the beginning of a macro definition
A macro definition cannot begin with a token-pasting (**##**) operator.
- C2161 **##** cannot occur at the end of a macro definition
A macro definition cannot end with a token-pasting (**##**) operator.
- 2162 expected macro formal parameter
The token following a stringizing operator (**#**) must be a formal parameter name.
- 2163 *string* : not available as an intrinsic function
A function specified in the list of functions for an intrinsic or function pragma is not one of the functions available in intrinsic form.
- C2164 *identifier* : intrinsic function not declared
The intrinsic function was not declared.
- C2165 *keyword* : cannot modify pointers to data
Bad use of **fortran**, **pascal** or **cdecl** keyword to modify pointer to data.
- C2167 *name* : too many actual parameters for intrinsic
A reference to the intrinsic function *name* contains too many actual parameters.

- C2168 *name* : too few actual parameters for intrinsic
A reference to the intrinsic function *name* contains too few actual parameters.
- C2169 *name* : intrinsic function, cannot be defined
An attempt was made to provide a function definition for a function already declared as an intrinsic.
- C2170 *identifier* : not declared as a function, cannot be intrinsic
You tried to use the **intrinsic** pragma for an item other than a function, or for a function that does not have an intrinsic form.
- C2171 *operator* : illegal operand
Illegal operand type for the specified unary *operator*.
- C2172 *actual* : actual is not a pointer : parameter *number*
The expression result type is checked against the given argument.
- C2173 *actual* : actual is not a pointer : parameter *number*, parameter list *number*
The expression result type is checked against the given argument.
- C2174 *actual* : actual has type void : parameter *number*, parameter list *number*
The expression result type is checked against the given argument.
- C2177 constant too big
Information was lost because a constant value was too large to be represented in the type to which it was assigned. (1)
- C2180 controlling expression has type void
Result type of the expression is void.
- C2182 *identifier* : has type void
Illegal non-indirected void.
- C2187 : cast of near function pointer to far function pointer
You attempted to cast a near function pointer as a far function pointer.
- C2189 #error : *string*
Emit pragma #error *string*.
- C2193 *identifier* : already in a segment
The symbol was already in a segment.
- C2194 *segment* : is a text segment
The named *segment* is a text *segment*.
- C2195 *segment* : is a data segment
The named *segment* is a data *segment*.
- C2200 *name* : function has already been defined
The function *name* has already been defined.
- C2201 *identifier* : storage class must be extern
The *identifier* is an auto or a register, it should be extern.

- C2205 *identifier*: cannot initialize extern block-scoped variables
Cannot initialize extern block-scoped variables.
- C2206 *name*: typedef cannot be used for function definition
A declarator was declared to be a function by using a typedef.
- C2208 no members defined using this type
Tried to define a member of a struct/union of inappropriate type.
- C2219 syntax error : type qualifier must be after '*'
Type qualifier must be after '*'.
- C2223 left of *->identifier* must point to struct/union
Left operand must point to struct/union.
- C2224 left of *.identifier* must have struct/union type
Left operand must have struct/union type.
- C2226 syntax error : unexpected type *typename*
Unexpected type.
- C2229 *member* in *keyword nametag* has a zero-sized array
A zero sized array was detected.
- C2231 *'.'* : left operand points to *keyword*, use *'->'*
The indirection is an address. Use *'->'* instead.
- C2232 *'->'* : left operand has *keyword* type, use *'.'*
No indirection. Use *'.'* instead.
- C2400 in-line syntax error in *context*, found *token*
The asm inline has a syntax error. The *context* is either *operand n* or *opcode*. The *token* is the offending input token.
- C2401 *register*: register must be base in *operand n*
The asm inline statement is attempting to use *register* as a base/register. A valid base register is required.
- C2402 *register*: register must be index in *operand n*
The asm inline statement is attempting to use *register* as an index register. A valid index register is required.
- C2403 *register*: register must be base/index in *operand n*
The asm inline statement is attemptint to use *register* as a base/index register. A valid base/index register is required.
- C2404 *register*: illegal register in *operand n*
The asm inline statement is attempting to use *register* in an invalid context.
- C2405 illegal short forward reference with offset
The asm inline statement is attempting a short jump or call to a forward location. This forward location cannot ve range checked. The asm code must be written to avoid the short forward reference.

- C2406 *symbol*: name undefined in *operand n*
The asm inline statement is using *symbol* which is undefined.
- C2407 illegal float register in *operand n*
The asm inline statement has an operand which is not a valid float register.
- C2408 illegal type on PTR operator in *operand n*
The asm inline statement has a bad *operand*. The PTR operator specifies a bad type.
- C2409 illegal type used as operator in *operand n*
The asm inline statement has a bad *operand*. A type specifier is being used as an operator.
- C2410 *member*: ambiguous member name in *operand n*
The asm inline statement has a bad operand. The *member* name is ambiguous. The structure/union has members which the asm inline mechanism cannot distinguish.
- C2411 *member*: illegal struct/union member in *operand n*
The asm inline statement has a bad operand. The *member* is not a member of the struct/union specified.
- C2412 *labelname*: case insensitive label redefined
The asm inline statement is trying to define a *label* that has already been used. Use a different *label name*. Note that these *label names* are case insensitive.
- C2413 *value*: illegal align size
The asm inline statement has specified an illegal *value* to the align directive. The allowed range is 0 through 4, inclusive.
- C2414 illegal number of operands
The asm inline statement has the wrong number of operands for the specified opcode.
- C2415 improper operand type
The asm inline statement has the wrong type of operands for the specified opcode. Or else, a SEG or OFFSET operator has an improper operand.
- C2416 *opcode*: illegal opcode for processor
The asm inline statement uses *opcode* which is not a valid opcode for the processor.
- C2417 divide by zero in *operand n*
The asm inline statement has an operand which results in a divide by zero. Constant folding is applied to the operand, so the bad *operand* value is detected at compile time, instead of runtime.

- C2419 `mod by zero in operand n`
 The asm inline statement has an operand which results in a modulus by zero. Contant folding is applied to the operand, so the bad operand value is detected at compile time, instead of runtime.
- C2420 `identifier:illegal symbol in operand n`
 Error in asm inline statement. The *n*-th operand uses an illegal symbol. The allowable symbols are symbols which have already been seen, or a label, function, or data symbol that has not yet been seen.
- C2421 `PTR operator used with register in operand n`
 Error in asm inline statement. The *n*-th operand is a register, and an illegal attempt to apply the PTR operator to the register is made.
- C2422 `illegal segment override in operand n`
 Error in asm inline statement. The *n*-th operand has an illegal segment-override. A valid segment register is required, and the operand to which the segment override applies cannot be in a register.
- C2423 `value:illegal scale`
 Error in asm inline statement. The scale *value* for an index instruction must be one of 1, 2, 4, or 8.
- C2424 `op:improper expression in '%s'`
 Error in asm inline statement. The operands specified for the *op* operator are invalid.
- C2425 `op:nonconstant expression in operand n`
 Error in asm inline statement. A constant was expected for one of the operands of *op*.
- C2426 `op:illegal operator in operand n`
 Error in asm inline statement. The *n*-th operand has more than one level of brackets (ie. indexing).
- C2427 `label:jump referencing label is out of range`
 A short jump is out of range. The allowed range is +/-127.
- C2429 `label:illegal far label reference`
 Error in asm inline statement. A jump or call to a far *label* is not allowed.
- C2430 `more than one index register in operand n`
 Error in asm inline statement. Only one of the operands can use an index register in a single instruction.
- C2431 `illegal index register in operand n`
 Error in asm inline statement. The *n*-th operand uses an illegal register for indexing. Use an allowable index register.

- C2432 illegal reference to 16-bit data in *operand n*
 Error in asm inline statement. The *n*-th operand tries to access memory in a 16-bit mode which is not supported in the 386 compiler.

Warning messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in parentheses at the end of each warning-message description (if any) gives the minimum warning level that must be set for the message to appear.

- C4000 UNKNOWN WARNING Contact SCO Technical Support
 The compiler detected an unknown error condition.
- C4001 nonstandard extension used - *identifier*
 A non-standard extension was used - see -Ze flag.
- C4002 too many actual parameters for macro *identifier*
 The number of actual arguments specified with the given *identifier* was greater than the number of formal parameters given in the macro definition of the *identifier*. (1)
- C4003 not enough actual parameters for macro *identifier*
 The number of actual arguments specified with the given *identifier* was less than the number of formal parameters given in the macro definition of the *identifier*. (1)
- C4004 missing ')' after defined
 The closing parenthesis was missing from an #if defined phrase. (1)
- C4005 *identifier* : macro redefinition
 The given *identifier* was redefined. (1)
- C4006 #undef expected an identifier
 The name of the identifier whose definition was to be removed was not given with the #undef directive. (1)
- C4007 *identifier* : must be *calltype*
 The *identifier* is the entry point from the program start-up code. It will use the regular C calling convention, whether or not some other calling convention is specified.
- C4008 *identifier* : *calltype* attribute on data ignored
 The *identifier* is not a function, so the *calltype* attribute is ignored.
- C4009 string too big, trailing chars truncated
 A string exceeded the compiler limit on string size. To correct this problem, break the string into two or more strings. (1)
- C4011 identifier truncated to *identifier*
 Only the *identifiers* first 31 characters are significant. (1)

- C4012 float constant in a cross compilation
The code is being cross-compiled. Value of constant could be different if compiled on target machine.
- C4015 *identifier* : type of bit field must be integral
The given bit field was not declared as an integral type.
Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)
- C4016 *identifier* : no function return type, using int as default
The given function had not yet been declared or defined, so the return type was unknown.
The default return type (**int**) is assumed. (2)
- C4017 cast of int expression to far pointer
A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer may produce an address with a meaningless segment value. (1)
- C4018 *operator* : signed/unsigned mismatch
Signed/unsigned mismatch between operands.
- C4020 *funcname* : too many actual parameters
The number of arguments specified in a function call was greater than the number of parameters specified in the argument-type list or function definition. (1)
- C4021 *funcname* : too few actual parameters
The number of arguments specified in a function call was less than the number of parameters specified in the argument-type list or function definition. (1)
- C4022 pointer mismatch : parameter *n*
The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition. (1)
- C4024 *funcname* : different types : parameter *n*
The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition. (1)
- C4026 function was declared with formal argument list
The function was declared to take arguments, but the function definition did not declare formal parameters. (1)
- C4027 function was declared without formal argument list
The function was declared to take no arguments (the argument-type list consisted of the word **void**), but formal parameters were declared in the function definition or arguments were given in a call to the function. (1)

- C4028 parameter *n* declaration different
The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)
- C4029 declared parameter list different from definition
The argument-type list given in a function declaration did not agree with the types of the formal parameters given in the function definition. (1)
- C4030 first parameter list is longer than the second
A function was declared more than once with different argument-type lists in the declarations. (1)
- C4031 second parameter list is longer than the first
A function was declared more than once with different argument-type lists. (1)
- C4032 unnamed struct/union as parameter
The structure or union type being passed as an argument was not named, so the declaration of the formal parameter cannot use the name and must declare the type. (1)
- C4033 function must return a value
A function is expected to return a value unless it is declared as **void**. (2)
- C4034 sizeof returns 0
The **sizeof** operator was applied to an operand that yielded a size of zero. (1)
- C4035 *identifier* : no return value
A function declared to return a value did not do so. (2)
- C4036 unnamed *struct/union* as parameter
Unnamed *struct/union* as parameter: see -Zg flag.
- C4040 memory attribute on *identifier* ignored
The **near** or **far** keyword has no effect in the declaration of the given *identifier* and is ignored. (1)
- C4042 *identifier* : has bad storage class
The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context was used in place of the illegal class. (1)
- C4044 **_huge** on *identifier* ignored, must be an array
The **huge** keyword was used to declare the given nonarray item. (1)
- C4045 *identifier* : array bounds overflow
Too many initializers were present for the given array. The excess initializers are ignored. (1)

C4047 *operator* : different levels of indirection

An expression involving the specified *operator* had inconsistent levels of indirection. (1)

The following example illustrates this condition:

```
char **p;  
char *q;  
.  
.  
.  
p = q;
```

C4048 arrays declared subscripts different

An array was declared twice with different sizes. The larger size is used. (1)

C4049 *operator* : indirection to different types

The indirection *operator* (*) was used in an expression to access values of different types. (1)

C4050 *operator* : different code attributes

The operands to the *operator* are pointers to functions. However, the **interrupt**, **saveregs**, **export**, or **loads** modifiers of the operands do not match.

C4051 type conversion - possible loss of data

Two data items in an expression had different types, causing the type of one item to be converted. (2)

C4053 at least one void operand

An expression with type **void** was used as an operand. (1)

C4056 overflow in constant arithmetic

The result of an operation exceeded 0x7FFFFFFF. (1)

C4057 overflow in constant multiplication

The result of an operation exceeded 0x7FFFFFFF. (1)

C4058 address of automatic (local) variable taken, DS != SS

The program was compiled with the default data segment (**DS**) not equal to the stack segment (**SS**), and the program tried to point to a frame variable with a near pointer. (1)

C4059 segment lost in conversion

The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) resulted in the loss of the segment address. (1)

C4061 long/short mismatch in argument: conversion supplied

The base types of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

- C4062 near/far mismatch in argument: conversion supplied
The pointer sizes of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)
- C4063 *identifier*: function too large for post-optimizer
The given function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)
- C4065 recoverable heap overflow in post-optimizer - some optimizations may be missed
Some optimizations were skipped because not enough space was available for optimization. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)
- C4066 local symbol table overflow - some local symbols may be missing in listings
The listing generator ran out of heap space for local variables, so the source listing may not contain symbol-table information for all local variables.
- C4067 unexpected characters following *directive*
directive - newline expected
Extra characters followed a preprocessor *directive*, as in the following example (1):
- ```
#endif NO_EXT_KEYS
```
- This is accepted in Version 3.0, but not in Versions 4.0 and 5.0. Versions 4.0 and 5.0 require comment delimiters, such as the following:
- ```
#endif    /* NO_EXT_KEYS */
```
- C4068 unknown pragma
The compiler did not recognize a pragma and ignored it. (1)
- C4069 conversion of near pointer to long integer
A near pointer was converted to a long integer, which involves first extending the high-order word with the current data-segment value, not 0 as in Version 3.0. (1)
- C4071 *identifier*: no function prototype given
The given function was called before the compiler found the corresponding function prototype. (3)
- C4072 *funcname* no function prototype on _fastcall function
No function prototype.
- C4073 scoping too deep, deepest scoping merged when debugging
Declarations appeared at a static nesting level greater than 13. As a result, all declarations will seem to appear at the same level. (1)

- C4076 *type* : may be used on integral types only
 The *type* modifiers **signed** and **unsigned** can be combined only with other integral *types*.
- C4077 unknown *check_stack* option
 Unknown option given when using the old form of the *check_stack* pragma. The option must be empty, **+**, or **-**.
- C4078 case constant *number* too big for the type of switch expression
 Switch expression limits the size that can be used as a case constant.
- C4079 unexpected token *token*
 Unexpected separator *token* found in argument list of a pragma.
- C4080 expected identifier for segment name, found *string*
 Expecting an identifier for segment name in the pragma.
- C4081 expected a comma, found *string*
 There is a missing comma (**,**) between two arguments of a pragma.
- C4082 expected an identifier, found *string*
 There is a missing identifier in list of arguments to a pragma.
- C4083 expected '(', found *string*
 There is a missing opening parenthesis (**(**) in the argument list for a pragma.
- C4084 expected a pragma directive, found *string*
 The token following the **pragma** keyword is not an identifier.
- C4085 expected [on|off]
 Bad argument given for new form of *check_stack* pragma.
- C4086 expected [1 | 2 | 4]
 Bad argument given for **pack** pragma.
- C4087 *name* : declared with void parameter list
 The function *name* was declared as taking no parameters, but a call to the function specifies actual parameters.
- C4088 *identifier*:pointer mismatch : parameter *number*, parameter list *number*
 Inline assembler warning.
- C4089 *identifier* different types : parameter *number* parameter list *number*
 Inline assembler warning.
- C4090 different const/volatile qualifiers
 The program passed a pointer to a **const** item to a function where the corresponding formal parameter is a pointer to a non-const item, which means the item could be modified by the function undetected.
- C4091 no symbols were declared
 An empty declaration was detected. (2)

- C4092 untagged enum/struct/union declared no symbols
An empty declaration was detected that used an untagged enum/struct/union. (2)
- C4093 unescaped newline in character constant in inactive code
The constant expression of an `#if`, `#elif`, `#ifdef`, or `#ifndef` preprocessor directive evaluated to 0, making the following code inactive, and a new-line character appeared between a single or double quotation mark and the matching single or double quotation mark in that inactive code.
- C4094 untagged *member* declared no symbols
An unnamed *member* is allowed to be an enum/struct/union, only if the enhancements are enabled. The ANSI C Standard only allows bitfield members to be unnamed.
- C4095 expected ')', found *string*
Pragma requires a right parenthesis ')' but did not find it.
- C4096 *modifier* must be used with *modifier*
Certain function modifiers must appear in combination. A warning is given when the compiler adds the necessary constraint.
- C4098 void function returning a value
A function was declared as not returning a value (ie. void), but a return value was specified in the function body.
- C4100 *formal*:unreferenced formal parameter
The parameter *formal* was not used in the function body.
- C4101 *name*:unreferenced local variable
The local variable *name* was not used in the function body.
- C4102 *label*:unreferenced label
The label *label* was not used in the function body.
- C4104 *identifier*:near data in same_seg pragma, ignored
Near data in same_seg pragma, ignored.
- C4105 *identifier*:code modifiers only on function or pointer to function
Code modifications are only allowed on function or pointer to function,
- C4106 : pragma requires integer between 1 and 127
You must supply an integer constant in the range 1-127, inclusive, for the given pragma.
- C4107 : pragma requires integer between 15 and 255
You must supply an integer constant in the range 15-255, inclusive, for the given pragma.

- C4108 : pragma requires integer between 79 and 132
You must supply an integer constant in the range 79-132, inclusive, for the given pragma.
- C4109 : unexpected identifier *token*
The designated line contains an unexpected *token*.
- C4110 : unexpected token 'int constant'
The designated line contains an unexpected integer constant.
- C4111 : unexpected token string
The designated line contains an unexpected string.
- C4112 : macro name *name* is reserved, *command* ignored
You attempted to define a predefined macro *name* or the preprocessor operator **defined**. This warning error also occurs if you attempt to undefine a predefined macro *name*. If you attempt to define or undefine a predefined macro *name* using command-line options, *command* will still be either **#define** or **#undef**.
- C4113 function parameter lists differed
Function parameter lists differed.
- C4114 same type qualifier used more than once
Same type of qualifiers used more than once, example *unsigned unsigned*.
- C4115 *name*:type definition in formal parameter list
Tag type definition in formal parameter list.
- C4116 (no tag) : type definition in formal parameter list
Unnamed tag type definition in formal parameter list.
- C4118 pragma not supported
This pragma is not supported.
- C4124 *_fastcall* with stack checking is inefficient
_fastcall and stackchecking are two contradictory goals.
- C4125 decimal digit terminates octal escape sequence
A non-octal digit was found and terminated the octal sequence.
- C4126 *<char>*:unknown memory model flag
An unknown memory model flag was specified in pass 1.
- C4127 conditional expression is constant
The expression has a constant value, it is always true or false.
- C4128 storage-class specifier after type
A storage-class specifier was found after the type.
- C4129 *<char>*:unrecognized character escape sequence
While getting an escaped character (\ddd) an unrecognized character was found.

- C4130 *string*: logical operation on address of string constant
 Attempting to do a logical operation on address of *string* constant.
- C4131 *name*: uses old-style declarator
 Using old-style declarator.
- C4132 *identifier*: const object should be initialized
 A non-extern const item has not been initialized.
- C4135 conversion between different integral types
 Converting an integral type to a smaller integral type.
- C4136 conversion between different floating types
 Converting a real type to a smaller real type.
- C4137 *funcname*: no return value from floating-point function
 Expected a return value from floating-point function.
- C4138 *'*/'* found outside of comment
 Improper comment.
- C4139 *hexvalue*: hex escape sequence is out of range
 Hex escape number is out of range, it must fit into a byte.
- C4186 string too long - truncated to *num* characters
 The string was too large, it has been truncated to *num* characters.
- C4200 local variable *identifier* used without having been initialized
 The local variable will have random contents.
- C4201 local variable *identifier* may be used without having been initialized
 The local variable may not have been initialized before being used.
- C4202 unreachable code
 This code will never be run.
- C4203 *funcname*: function too large for global optimizations
 Ran out of internal buffers necessary to do global optimizations.
- C4204 *name*: in-line assembler precludes global optimizations
 In-line assembler prevents global optimizations.
- C4205 statement has no effect
 The statement does nothing.
- C4206 assignment within conditional expression
 An assignment occurred inside a conditional expression.
- C4209 comma operator within array index expression
 Comma operator within array index expression.
- C4300 insufficient memory to process debugging information
 Ran out of memory while processing debug information.

- C4301 loss of debugging information caused by optimization
Optimization may have scrambled debugging information. Debugging information will be lost.
- C4302 loss of optimization with symbolic debug selected
Will not do certain optimizations if debug is selected.
- C4323 potential divide by 0
Potential divide by 0.
- C4324 potential mod by 0
Potential mod by 0.
- C4401 *identifier*:member is bit field
Inline assembly warning. The *identifier* is a bit field number.
- C4402 must use PTR operator
Inline assembly warning.
- C4403 illegal PTR operator
Inline assembly warning.
- C4404 period on directive ignored
Inline assembly warning. The ALIGN and EVEN directives do not require a period prefix.
- C4405 *identifier*:identifier is reserved word
Inline assembly warning. The *identifier* is a reserved word. The context in this case is able to determine that it is an identifier. It is not recommended that reserved words be used as identifiers.
- C4406 operand on directive ignored
Inline assembly warning. The assembler directive does not require the operand. The operand is ignored.
- C4407 operand size conflict
Inline assembly warning. The size of the operands are in conflict with each other. The size chosen is the size associated with the given opcode. If the opcode does not specify a size, then the size of whichever operand is a register is chosen.
- C4409 illegal instruction size
Inline assembly warning. The size specified for the opcode is not valid for the opcode. A default size appropriate for the opcode is chosen. It is much safer to specify the correct size than to depend on the default value.
- C4410 illegal size for operand
Inline assembly warning. The size of some operand conflicts with the requirements of the opcode. A default size appropriate for the opcode is chosen. It is much safer to specify the correct size than to depend on the default value.

- C4411 *identifier*: symbol resolves to displacement register
 Inline assembly warning. The *identifier* is a register argument. Make sure that the inline assembly is appropriate, given this fact.
- C4414 *identifier*: short jump to function converted to near
 Inline assembly warning. The size specified for the *call* instruction is byte sized. The size is converted to a near call.

Index

A

Abbreviations used in lex, 201

Action

- associated with yacc rules, 218, 220
- default in yacc specification, 221
- executed by parser, 215
- in the middle of yacc rules, 222

Address

- binding, ld(CP), 24
- physical, ld(CP), 24

admin command, SCCS, 118, 124-125, 139

Allocation

- algorithm in ld(CP), 43
- virtual address space, ld(CP), 23

Ambiguities in lex. *See* Disambiguating rules

Ambiguous grammar rules for yacc, 237

ANSI, predefined macro names, cc(CP), 19

a.out file

- building, 153
- compatibility, 181
- differences with shared and non-shared libraries, 155
- identification, 158
- importing symbols, 173
- incompatible libraries, 184
- l option, 153
- non-shared libraries, 155
- referencing symbols, 153
- shared libraries, 151-152, 155, 159-160, 166, 169, 176, 190

Appending to yytext with yymore, 207

Arbitrary value types in yacc specifications, 225

Archive libraries

- compatibility, 179
- make(CP), 101, 104
- non-shared libraries, 152
- searching, ld(CP), 41
- shared library, 152

Arguments

- m4(CP), 262, 266
- macro, 266
- unused in lint, 53

Arithmetic expression parsing, 239

arithmetic macro, m4(CP), 267

Assembly language, shared and non-shared libraries, 157

Assignment statement

- ld(CP), 26, 37
- source code, ld(CP), 27

Associativity rules for yacc grammar, 239

Auditing, in SCCS, 149

B

Binding address, ld(CP), 24

Blanks in yacc specifications, 217

Branch

- libraries, 160
- table in shared libraries, 160, 167

#branch in shared libraries, 170

Branching, in SCCS, 122

browser. *See* cscope

Browser choice for cscope, 79

.bss, ld(CP), 24

Built-in macros in m4(CP), 261

C

C shared and non-shared libraries, 157

C code

- created from lex specifications, 212
- translating lex specifications into C, 213

C compiler. *See* Compiling and linking C programs

C declarations in yacc specifications, 225

C Programmer's Productivity Tools (CPPT), 61

C programs, compiling and linking, cc(CP), 9

cc(CP)

- checking syntax, 16
- compiling simple programs, 10
- disabling and enabling language extensions, 19
- disabling optimization, 17
- error messages, 14
- file processing, 11
- format, 11

cc(CP) (*continued*)

- linking libraries, 14
- linking object files, 13
- maximum optimization, 18
- naming executable files, 13
- naming object files, 12
- options, 12-13, 15-21
- redirecting compiler error messages, 15
- specifying file, 11-12
- specifying output file, 12
- syntax, 11
- warning level, 15
- warning messages, 14
- cc(CP) optimizing
 - code size, 17
 - execution time, 17
 - for consistent floating point results, 17
 - for speed, 17
- cc(CP) option
 - D, 175
 - I, 174
 - xenix, 21
- cdc command, SCCS, 125, 143
- cflow(CP), 180
- changequote macro, m4(CP), 265
- Character, non-portable use with lint, 57
- Checksum, in SCCS, 148
- chkshlib(CP), 151, 181
- chmod, in SCCS, 124
- .cnt files, 62
- Code size, optimizing, cc(CP), 17
- CodeView debugger, with cc(CP), 16
- comb command, SCCS, 125, 145
- Command language
 - expressions, ld(CP), 25
 - symbols, ld(CP), 36
- Command line, error messages, 305
- Commands
 - SCCS, 124
- Comments
 - in make(CP), 96
 - in yacc specifications, 217
- Compiler
 - Microsoft C Version 6.0, 9
 - error message, 309
 - error messages, cc(CP), 15
 - rcc, cc(CP), 9, 22
 - redirecting error messages, cc(CP), 15
 - writing with lex, 193

Compilers. *See* Compiling and linking C programs

Compiling

- C programs, cc(CP), 9
- and linking C programs, cc(CP), 9
- programs, cc command, 10
- programs for DOS and OS/2, cc(CP), 21
- programs for XENIX, cc(CP), 21

Concurrent edits

- of different SID in SCCS, 132
- of same SID in SCCS, 134

Conditional compilation directives in cscope, 88

Conditionals, m4 macro processor, 269

Configured memory, ld(CP), 24

Context sensitivity in lex, 203

Continuation lines, in make(CP), 96

COPY section in ld(CP), 45

Core files, shared libraries, 158

Correctable error messages, 309

Coverage analysis with lprof, 61

CPPT (C Programmer's Productivity Tools)

- cscope. *See* cscope

- introduced, 61

- lprof. *See* lprof

Creating

- SCCS File using admin, 118

- SCCS files, 139

- new macros, m4(CP), 262

Cross-reference symbol table for cscope, 82

cscope

- browser choice, 79

- cross-reference symbol table, 78, 82

- cscope.out cross-reference table, 82

- editing, 78-79, 88

- environment variables, 78-80

- escape to the shell, 91

- examples of use, 89

- introduced, 61

- locating error message source, 83

- setting terminal type, 78

- stacking editor calls, 88

- terminal information (terminfo)

- database, 78

- use, 80

D

.data
 ld(CP), 24
 shared libraries, 155
 dbxtra, 158
 Debugger
 CodeView with cc(CP), 16
 cc(CP), 16
 dbxtra with cc(CP), 16
 sdb with cc(CP), 16
 shared libraries, 158
 yacc, 230
 Declarations in yacc specification. *See* yacc declarations
 Default action in yacc specification, 221
 define
 arguments in m4(CP), 262
 expansion delayed with quoting in m4(CP), 264
 macro in m4(CP), 262
 Definitions
 lex, 195, 200
 yacc specification. *See* yacc declarations
 Delimiters
 in yacc, 216, 218, 225
 Delta
 comments, SCCS, 140
 numbering, SCCS, 122
 delta command, SCCS, 120, 125, 136-137
 delta file, SCCS, 118, 120
 Descriptive text, in SCCS, 148
 Disabling optimization, cc(CP), 17
 Disabling or enabling language extensions, cc(CP), 19
 Disambiguating rules
 lex, 203, 205
 yacc, 238, 242
 Disk storage space, 151
 divert macro, m4(CP), 268
 divnum macro, m4(CP), 269
 Dollar-sign symbol in yacc rules, 221
 dosld(CP), 21
 DSECT dummy section, ld(CP), 45
 Dummy section, DSECT option, ld(CP), 45
 dump(CP), 158
 dumpdef macro, m4(CP), 272
 dynamic analysis, 61
 Dynamic dependency parameters
 in make(CP), 109

E

ECHO macro in lex, 199
 Editor
 calls stacked with cscope, 88
 choice for cscope, 79
 link editor. *See* ld(CP)
 EDITOR environment variable for cscope, 79
 Empty rules in yacc specification, 219
 Enabling or disabling language extensions, cc(CP), 19
 End marker in yacc specifications, 229
 Entry point, changing, ld(CP), 40
 Enumeration, type checking with lint, 56
 Error
 handling in yacc, 242, 244
 recovery, interactive in yacc, 244
 recovery, YYERROR() function in yacc, 234
 reporting in yacc, 230
 token in yacc, 242
 Error message
 SCCS, 126, 128-129, 132
 cc(CP), 14-15
 compiler, 305, 309-310, 331
 cscope, 78
 fatal, cc(CP), 15
 format. *See* Error message, compiler help, SCCS, 125-126
 ld(CP), 32
 lint, 54
 locating source with cscope, 83
 m4(CP), 272
 profiled program, lprof, 71
 redirecting, cc(CP), 15
 syntax, cc(CP), 16
 warning as fatal, cc(CP), 16
 yacc, 227, 230
 Errorlevel codes *See* Exit codes
 errprint macro, m4(CP), 272
 Escape sequences in yacc specifications, 218
 eval macro, m4(CP), 266-267
 Examples of yacc specifications, 248
 Executable files, naming, cc(CP), 13
 Exit code, 305
 #export, 166, 168
 Extensions, object files, cc(CP), 12

F

Fatal-error messages, 309

File

- SCCS, 147
- SCCS delta file, 118
- a.out file. *See* a.out file
- arguments in SCCS, 124
- creating, in SCCS, 118
- creating in SCCS, 139
- delta file in SCCS, 120
- end-of-file processing in lex, 209
- extensions, cc(CP), 11
- g.file in SCCS, 119
- interdependencies. *See* make(CP)
- linking object files, cc(CP), 13
- naming executable files, cc command, 13
- naming files, cc(CP), 12
- nonrelocatable input file, ld(CP), 46
- object file, cc(CP), 11-12
- object, ld(CP), 24
- output blocking, ld(CP), 46
- parameters in SCCS, 140
- processing, cc(CP), 11
- protection in SCCS, 147
- retrieving SCCS file with get, 119
- source file, cc(CP), 11
- source file, shared libraries, 185
- specifying file, cc(CP), 12
- find command, SCCS, 124
- .fini, ld(CP), 25
- finite-state machine, 231
- Flags, in SCCS, 148
- Floating point, optimizing for consistent results, cc(CP), 17
- fork(S), 65
- Format, yacc specifications, 216
- Formatting, in SCCS, 148
- Frequency profiling with lprof, 61
- Function
 - unused in lint, 53, 55

G

- get command options, SCCS, 119, 128-131, 135
- get command, SCCS, 119-120, 125-126
- getgrent(S) and shared library, 167
- g.file, in SCCS, 119
- Global data
 - shared libraries, 168, 170, 189

Grammar rules

- in yacc, 215, 218-219
- left associativity, yacc, 237
- left associativity, yacc, 237
- right associativity, yacc, 237
- GROUP directive, ld(CP), 34
- Grouping, output sections, ld(CP), 33

H

- help command, SCCS, 125-126
- #hide, 166, 168, 170
- Holes
 - creating within a section, ld(CP), 38
 - in physical memory, ld(CP), 39
- Host shared library, 152, 159, 165, 176, 182-183

I

- ID keywords, in SCCS, 127
- ifdef macro, m4(CP), 265
- ifndef macro, m4(CP), 269
- ifile, ld(CP), 25
- Implicit rules, in make(CP), 102
- Imported symbol, 177
- include files, in make(CP), 109
- include macro, m4(CP), 268
- incr macro, m4(CP), 267
- Incremental link editing, ld(CP), 43
- INFO section in ld(CP), 45
- #init, 170
- .init, initialized variable, ld(CP), 25
- Initialization and modification of file
 - parameters
 - in SCCS, 140
- Initialized variable, ld(CP), 25
- Input
 - section, ld(CP), 23
 - style for yacc specifications, 245
- Inserting commentary for the initial delta
 - in SCCS, 140
- Interactive error recovery in yacc, 244
- Internal rules of make(CP), 111
- Interpreting profiling output, 66
- I/O routines for lex. *See* lex I/O routines

J

j flag, in SCCS, 129

K

Kernighan and Ritchie code, compiling with
rcc, cc (CP), 9

Key letters that affect output
in SCCS, 135

Keywords in yacc specifications, 224, 240-241

L

Language extensions, enabling or disabling,
cc(CP), 19

ld(CP)

ADDR pseudo-function, 28, 32

ALIGN pseudo-function, 28, 32

C language operators in command lan-
guage, 25

COPY, 45

DSECT dummy section, 45

GROUP directive, 34

INFO, 45

MEMORY directives, 28

NEXT pseudo-function, 32

NOLOAD, 46

OVERLAY, 46

SECTIONS directive, 30, 32

SECTIONS output alignment, 32

SIZEOF pseudo-function, 28, 32

address, 24

allocation, 23, 43

archive libraries, 41

assignment statement, 26-27, 37

binding address, 24

.bss, 24

changing entry point, 40

command language expressions, 25

configured memory, 24

constants in command language, 25

.data, 24

dummy section, 45

expressions, 25

.fini, 25

global symbols in command language, 25

grouping output sections, 33

holes, 35, 39

ld(CP) (*continued*)

ifile, 25

incremental link editing, 43

.init, ld(CP), 25

initialized section holes, 38

input section, 23

introduced, 23

-l option, 41

linking, cc(CP), 13

location counter, 27

memory configuration, 24-25, 28

nonrelocatable input file, 46

output, 23, 46

-r option, 43

section, 23-24, 37

shared libraries, 160

source code and assignment statement
interaction, 27

symbol creating and defining, 36

syntax for input directives, 47

.text, 24

-u option, 42

unconfigured memory, 24

Left associativity rules for yacc
grammar, 239

Left recursion rules for yacc, 246

len macro, m4(CP), 270

Level numbering, delta numbering in SCCS,
122

lex

I/O routines, 206

REJECT, 208

UNIX systems, 212

actions, 198-199

compiling and linking, 213

context sensitivity, 203-204

definitions, 195, 200-201

disambiguating rules, 203, 205

end-of-file processing with yywrap, 209

introduced, 193

library default main() program, 213

options, 213

output file, 213

programs checking with lint, 55

regular expressions, 195

reprocessing input with yymore, 207

rules, 195

running lexical analyzer program, 213

specification, 195

specifications, 193

subroutines, 201

lex (continued)

- symbol table, 211
 - tokens, 195, 209
 - used, 214
 - used with yacc, 217, 227
 - user subroutines, 195
 - y.tab.h file, 211
 - yylen variable, 199
 - yyless, 208
 - yylex() lexical analyzer, 209
 - yytext character array, 199
 - yyval variable, 211
- Lexical analysis,
- lex, 193
 - yacc, 215, 227
- Lexical analyzer yylex() generated by
- lex, 209
- Lexical tie-ins, yacc, 246
- lex.yy.c file, 213
- .lib, 151
- libc, 173
- libmalloc, 173

Library

- archive. *See* Archive library
 - cc(CP) -M options, 13
 - host, 165, 176
 - lex, 213
 - libc, 173
 - libmalloc, 173
 - lint, 52
 - math, 52
 - networking, 152-153
 - non-shared, 152
 - search path, cc(CP), 14
 - shared. *See* Shared libraries
 - target, 165
 - yacc access, 230
- Line count data, 62
- Lines, newlines in yacc specifications, 217
- Link edit, shared libraries, 151
- Link editor. *See* ld(CP)
- Linking
- shared libraries, 160
 - standard C library, 13
 - with additional libraries, cc(CP), 14
 - with cc(CP), 13
- Linking and compiling C programs, cc(CP), 9

lint

- flow of control, 54
- introduction, 51

lint (continued)

- libraries, 52
 - longs to ints, 57
 - math library, 52
 - messages, 53
 - non-portable character use, 57
 - options, 52
 - portable library, 52
 - set, 54
 - subexpression evaluation order, 58
 - syntax, 51
 - type casts, 56
 - type checking, 56
 - unused arguments, 53
 - unused function values, 55
 - unused variables and functions, 53
 - usage, 51
 - used with lex or yacc, 55
 - variable used before set, 54
- Literals in yacc specifications, 218
- Location counter, ld(CP), 27
- look-ahead token, 231, 244
- lprof
- PROFOPTS environment variable, 62
 - behavior differences with rcc, 76
 - .cnt file, 62
 - coverage analysis, 61
 - creating profiled program, 62
 - data files in separate directory, 65
 - dynamic analysis, 61
 - frequency profiling, 61-62
 - incompatibility with optimization, 72
 - interpreting output, 66
 - introduced, 62
 - line count data, 62
 - merging data files, 64
 - non-terminating programs, 72
 - profiled source listing, 66
 - programs that fork(S), 65
 - run-time data, 62
 - summary option, 71
 - test coverage improvement, 74
 - timestamp override, 71
 - unexecuted lines, 68
 - used with program generators, 73
 - used with rcc compiler, 76
 - used with shared libraries, 73
- ls command, SCCS, 124

M

m4(CP)

See also Macro processor

arguments, 262
 basic operation, 261
 introduced, 261
 invoking, 262
 macro, 262
 macros, 261-262
 printing, 272
 quoting to delay define expansion, 264
 redirection, 262
 removing macro definitions, 265
 standard input, 262
 standard output, 262
 undefine function, 265
 using arguments, 266

Macro

ECHO in lex, 199
 arithmetic, m4(CP), 267
 built-in, m4(CP), 261
 changequote, 265
 defining in m4(CP), 262
 defining, m4(CP), 261
 definition removed with undefine, 265
 definitions, make(CP), 97
 divert, m4(CP), 268
 divnum m4(CP), 269
 dumpdef in m4(CP), 272
 errprint, m4(CP), 272
 eval, m4(CP), 266
 ifdef, m4(CP), 265
 ifelse, m4(CP), 269
 include, m4(CP), 268
 len, m4(CP), 270
 printing, 272
 sinclude, m4(CP), 268
 substr, m4(CP), 271
 syscmd, m4(CP), 269
 translations, make(CP), 101
 translit, m4(CP), 271
 undefine in m4(CP), 265
 undivert, m4(CP), 268
 user-defined in m4(CP), 261

Macro processor

See m4(CP)
 arguments, 266
 conditionals in m4(CP), 269
 m4(CP), 261
 printing, 272

Macro processor (*continued*)

system commands in m4(CP), 269
 main() routine in yacc, 227
 make(CP)
 Include files, 109
 MAKEFLAGS, 110
 Tilde (~) in SCCS filenames, 106
 archive libraries, 101, 104
 comments, 96
 continuation lines, 96
 creating new suffix rules, 108
 dependency, 96
 dynamic dependency parameters, 109
 environment variables, 110
 example makefile, 94, 99
 file interdependencies, 93-94, 109
 implicit rules, 102
 internal rules, 111
 introduction, 93
 macro definitions, 97
 macro translations, 101
 maintaining parser code, 231
 makefile, 93-94, 96
 null suffix, 107
 recursive makefiles, 102
 suffix rules, 94, 101-102, 107-108
 timestamp information, 94
 used, 214
 used with yacc, 231
 using shell commands, 100
 Makefile. *See* make(CP)
 makefile. *See* make(CP)
 MAKEFLAGS. *See* make(CP)
 malloc(S), 173
 Manipulating files
 include function, 268
 macro processors, 268
 Manipulating strings, m4 macro processor, 270
 Math library, lint, 52
 Memory configuration, ld(CP), 24-25, 28
 MEMORY directives, ld(CP), 28
 Merging, cscope, 64, 71
 Messages
 lint, 53
 warning and error messages, cc(CP), 14
 Microsoft C compiler, 9
 mkshlib(CP), 151, 161, 165, 170, 178, 190-191
 Multi-line action in lex, 199
 Multi-statement action in lex, 199
 Multiple function definitions and cscope, 88

N

Name

- executable files, cc(CP), 13
- object file naming with cc command, 12
- yacc specifications, 218

Newlines in yacc specifications, 217

noload, 178

NOLOAD section in ld(CP), 46

Non-portable character use
with lint, 57

Non-shared libraries, 152

Non-terminating profiled programs, 72

Null suffix, in make(CP), 107

Numbering, delta in SCCS, 122

O

#object, 170

Object file

- cc(CP), 11
- default extension, cc(CP), 11
- extensions, cc(CP), 12
- ld(CP), 24
- linking, cc(CP), 13
- naming with cc command, 12
- section, ld(CP), 23

Operand types, type checking with lint, 56

Optimizing

- consistent floating point results, cc(CP), 17
- default, cc(CP), 17
- disabling, cc(CP), 16-17
- for code size, cc(CP), 17
- for execution time, cc(CP), 17
- for speed, cc(CP), 17
- incompatibility with profiling, 72
- maximum, cc(CP), 18
- unsafe procedures, cc(CP), 18

Output file

- blocking, ld(CP), 46
- specifying, cc(CP), 12

Output section

- creating holes, ld(CP), 35
- ld(CP), 23

Output translations, in make(CP), 101

OVERLAY section, ld(CP), 46

P

Parse trees in yacc rules, 224

Parser

- compiling and running, 231
- error recovery, 234
- generator. *See also* yacc
- operation, 232
- reset with yyerror, 244

Parsing

- See also* yacc
- arithmetic expressions, 239

Pattern in lex specification. *See also* Regular expressions

Performance improvement with prof and lprof, 73

p.file, in SCCS, 120

Physical address, ld(CP), 24

Portability restrictions, checking with lint, 51

Portable lint library, 52

Portable operating system for UNIX. *See* POSIX

POSIX conforming code, cc(CP), 20

Precedence rules for yacc grammar, 239

Preparing specifications for yacc, 245

printf(S), 158, 160, 166-167

Printing, m4(CP), 272

prof, time-profiling, 62

Profile options. *See* PROFOPTS

Profiled program

- creating, 62
- running, 62

profiler. *See* lprof

Profiling

- error messages, cscope, 63
- programs that fork(S), 65
- within a shell script, 65

PROFOPTS, lprof

- environment variable, 62
- examples, 63
- options list, 63
- profiling within a shell script, 65
- turning off, 64

Program

- checker. *See* lint
- creating profiled version, 62
- generator. *See* lex
- generator and lprof, 73

Protection, in SCCS, 147
 prs command, SCCS, 125, 141
 Pseudo-variable in yacc rules, 221, 223

Q

Quoting
 m4(CP), 264

R

rcc
 compiler, cc(CP), 9
 used with lprof, 76
 Recording changes
 SCCS, 120
 Recursion rules for yacc, 246
 Reduce-reduce conflict in yacc, 238, 242
 Region addresses, shared libraries, 162
 Regular expressions in lex. *See also* Pattern
 REJECT in lex, 208
 Release numbering, delta numbering in SCCS,
 122
 Removing macro definitions with
 undefine, 265
 Reserved words in yacc, 247
 Resetting parser with yyerror, 244
 Retrieving
 different versions, SCCS, 128
 get command, SCCS, 119
 to make a delta, SCCS, 129
 Return codes *See* Exit Codes
 Return value, type checking with lint, 56
 Right
 associativity rules for yacc grammar, 239
 recursion rules for yacc, 246
 rlint. *See* lint
 rmdel command, SCCS, 125, 143
 Rules, yacc specification, 216, 218
 Runtime analysis. *See* lprof
 Run-time data file, 62

S

sact command, SCCS, 125, 142
 SCCS
 ID keywords, 127
 SCCS identification string(SID) *See* SID
 SID, 122

SCCS (continued)

access permission, 125
 auditing, 149
 beginners, 118
 body, 148
 branching, 122
 checksum, 148
 chmod, 124
 commands, 118-120, 124-125, 128-129, 131,
 135-137, 139-146
 concurrent edits of different SID, 132
 concurrent edits of same SID, 134
 creating a file using admin, 118
 creating of SCCS files, 139
 delta comments, 140
 delta file, 118, 120
 delta numbering, 122
 delta table, 148
 descriptive text, 148
 error messages, 126
 file arguments, 124
 filenames in make(CP), 106
 files, 147
 flags, 148
 formatting, 148
 get command, 126, 135
 get options, 131
 g.file, 119
 help command, 126
 id keywords, 127
 initialization and modification of file
 parameters, 140
 inserting commentary for the initial
 delta, 140
 introduced, 117
 j flag, 129
 key letters that affect output, 135
 permission, 125
 p.file, 120
 protection, 147
 recording changes by using delta, 120
 retrieving a file, 119, 128-129
 terminology, 118
 tree structure, 122
 undoing a get -e, 130
 user names, 148
 x.files and z.files, 124
 SCCS IDentification string (SID) *See* SID
 sccsdiff command, SCCS, 124-125, 145
 sdb debugger, with cc(CP), 16
 Searching for patterns with lex, 195

Section

- allocation into named memory, ld(CP), 37
- holes in physical memory, ld(CP), 39
- initialized holes, ld(CP), 38
- ld(CP), 23
- physical address, ld(CP), 24

Section definition, ld(CP)

- COPY, 45
- INFO, 45
- NOLOAD, 46
- OVERLAY, 46

Section type. *See* Section definition

SECTIONS

- assignment statement, ld(CP), 37
- directive, ld(CP), 30
- file specification directive, ld(CP), 30
- loading to a specific address, ld(CP), 32
- output, ld(CP), 32

Shared libraries

- C library, 152
- a.out file, 155, 158-160, 166, 169
- assembly language, 157
- benefits, 152
- #branch, 170
- branch table, 160, 167, 171
- building, 161, 191
- building a.out file, 153
- building host libraries, 165
- building target libraries, 165
- changing existing code, 167
- checking for different version, 181
- chkshlib(CP), 181
- code conventions, 157
- coding an application, 157
- compatibility, 170, 179, 181
- data symbol, 168
- debugging, 158
- description, 151
- example, 152, 185
- #export, 166, 168, 171, 191
- external symbol, 168
- getgrent(S), 167
- global data, 168, 170
- guidelines, 165
- #hide, 166, 168, 170-171
- #hide linker, 191
- host, 152
- host library, 159
- implementing, 159
- importing symbols, 172, 177
- incompatible libraries, 184

Shared libraries (*continued*)

- #init, 170
- lc_s option, 152
- ld (CP) link editor, 160
- .lib, 155
- libc, 172-173
- libmalloc, 173
- link edit, 151
- lnsl_s option, 152
- malloc(S), 173
- members, 166
- memory, 151
- memory management unit (MMU), 162
- mkshlib(CP), 190-191
- networking library, 152
- noload, 178
- #object, 170
- #objects noload, 191
- paging, 180
- printf(S), 160, 166-167
- prof(CP), 180
- profiling, 73
- region addresses, 162, 188
- rewriting existing code, 165, 189
- routines, 166
- _s suffix, 152
- selecting contents, 164, 189
- source code, 157
- source files, 185
- space considerations, 154, 157
- specification file, 165, 170, 190
- stack variables, 168
- static data, 168
- static library symbols, 169
- static linking, 160
- strcmp(S), 157
- target library, 152, 159
- target library pathname, 164
- target pathname, 188
- tuning code, 180
- using, 191
- when to use, 154
- Shell commands, in make(CP), 100
- Shift-reduce conflict in yacc, 238, 242
- SID
 - SCCS identification string, 118
 - concurrent edits of different SID, 132
 - concurrent edits of same SID, 134
 - retrieving different versions, 128
- sininclude macro, m4(CP), 268

Source code and assignment statement,
ld(CP), 27

Source code control system. *See* SCCS

Source file

- default extension, cc(CP), 11
- in a different directory, 70
- searching with cscope, 80
- shared libraries, 185, 187, 189
- specifying, cc(CP), 12

Source listing for a subset of files, 70

Specification file, 190

Specifying program and data file to lprof, 69

Stacking cscope and editor calls, 88

Standard C library, linking, cc(CP), 13

Standards conformance, cc(CP), 19

Start symbol in yacc specifications, 225

Starting state in lex, 204

Static data, 167, 169

Static link editor, shared libraries, 160

strcmp(S), 157-158

Structure reference, type checking with lint, 56

Subexpression, evaluation order with lint, 58

Subroutines

- lex, 201
- yacc. *See* yacc subroutines

substr macro, m4(CP), 271

Suffix rules, in make(CP), 94

Suffixes and transformation rules

- in make(CP), 102

SVID conforming code, cc(CP), 21

Symbol

- ($\$$) dollar-sign in yacc rules, 221
- importing into shared libraries, 172-173, 177
- ld(CP) command language, 36
- left context in yacc specification, 223
- table, cscope, 82
- table, lex, 211
- terminal and non-terminal in yacc specification, 218
- values in yacc rules, 221

Syntax

- checking, cc(CP), 16
- diagram for input directives, ld(CP), 47
- error messages, cc(CP), 16
- error messages in yacc, 230
- ld(CP), 26
- lint, 51

syscmd macro, m4(CP), 269

System commands, m4 macro processor, 269

T

Tabs in yacc specifications, 217

Target program, using make(CP) to produce, 94

Target shared libraries, 152, 159, 165, 182

TERM environment variable for cscope, 78

Terminal

- and non-terminal symbols in yacc specification, 218
- information (terminfo) database, 78
- type setting for cscope, 78

Terminfo database, 78

Terminology, in SCCS, 118

Test coverage improvement with lprof, 74

.text, executable instructions

- ld(CP), 24
- shared libraries, 155

Time-profiling with prof, 62

Timestamp

- information
- in make(CP), 94
- override with lprof, 71

Token

- changing the value, 211
- declared in yacc specifications, 224
- in yacc, 218
- lex specification, 195
- lexical analysis, 215
- look ahead clearing with yyclearin, 244
- look ahead use in yacc, 231
- names, reserved, 229
- numbers in yacc, 229
- returning, 209
- use with lexical analyzer and parser, 209
- yacc error token, 242
- yyclearin, 244

translit macro, m4(CP), 271

Tree structure, in SCCS, 122

Type

- casts with lint, 56
- checking with lint, 56

U

Unconfigured memory, `ld(CP)`, 24
 undefine macro in `m4(CP)`, 265
 undivert macro, `m4(CP)`, 268
 Undoing a `get -e`, in SCCS, 130
`unget` command, SCCS, 125
 Uninitialized variable, `ld(CP)`, 25
 Unused
 arguments, `lint`, 53
 function values, `lint`, 55
 functions, `lint`, 53
 variables and functions, `lint`, 53
 User
 defined macros in `m4(CP)`, 261
 defined subroutines. *See* `yacc` subroutines
 names, SCCS, 148
 subroutines, `lex`, 195

V

`val` command, SCCS, 124-125, 146
 Value
 left context in `yacc` specification, 223
 stack in `yacc` declarations, 225
 types in `yacc` specifications, 225
 Variable
 initialized, `ld(CP)`, 25
 uninitialized, `ld(CP)`, 25
 unused in `lint`, 53
`vc` command, SCCS, 125, 146
 VIEWER environment variable for
 `cscope`, 79
 VPATH environment variable for `cscope`, 80

W

Warning error messages, 310, 331
 Warning message
 SCCS, 128
 `cc(CP)`, 14
 setting level, `cc(CP)`, 15
`what` command, SCCS, 124-125, 144
 Wide return values, `cc(CP)`, 18

X

XENIX, compiling, `cc(CP)`, 21
`x.files` and `z.files`, in SCCS, 124
 XPG3 conforming code, `cc(CP)`, 20

Y

`yacc`
 YYERROR() parser error recovery, 234
 compiling and running the parser, 231
 declarations, 216, 224-226, 240-241
 delimiters, 218
 error. *See* `yacc` error
 forced error recovery, 244
 grammar rules. *See* `yacc` grammar rules
 grammar use, 215
 interactive error recovery, 244
 introduced, 215
 lexical analysis, 227
 library, accessing, 230
 maintenance using `make(CP)`, 231
 output file `y.tab.c`, 230
 parser operation, 231
 program checking with `lint`, 55
 reserved words, 247
 rules. *See* `yacc` rules
 source files, 231
 specifications. *See* `yacc` specifications
 subroutines. *See* `yacc` subroutines
 token names reserved, 229
 used with `lex`, 209, 217, 227
 `yychar`, 230
 `yydebug`, 230
 `yyparse( )` function, 230
`yacc` error
 forced recovery, 244
 handling, 242
 interactive recovery, 244
 recovery, reset with `yyerrok`, 244
 reporting, 230
 token, 242
`yacc` grammar rules
 ambiguous, 237
 associativity, 239
 disambiguating rules, 238, 242
 left associativity, `yacc`, 237
 precedence, 239
 reduce-reduce conflict, 238, 242
 right associativity, `yacc`, 237
 shift-reduce conflict, 238, 242

yacc rules

- actions, 218, 220, 222
- default action, 221
- dollar-sign symbol, 221
- empty rules, 219
- escape sequences, 218
- grammar, 218-219
- left recursion, 246
- literals, 218
- names, 218
- parse trees, 224
- pseudo-variable, 221, 223-224
- right recursion, 246
- specification, 218
- symbols, 218, 221, 223
- terminal and non-terminal symbols, 218
- tokens, 218
- values, left context, 223

yacc specifications

- blanks, 217
- comments, 217
- delimiter %, 216
- examples, 248
- format, 216
- input style, 245
- newlines, 217
- tabs, 217

yacc subroutines

- end marker, 229
- external variable yyval, 227
- lexical analyzer, 227
- main() routine, 227
- token numbers, 229
- user-defined, 227
- yyerror routine, 227
- yylex routine, 227

y.tab.c output file for yacc, 230

y.tab.h, used with yacc, 229

y.tab.h used with lex, 211

y.tab.h used with yacc, 211

yychar variable used in yacc, 230

yyclearin token in yacc, 244

yydebug used in yacc, 230

yyerrok, parser reset in yacc, 244

YYERROR() parser error recovery, 234

yyerror routine in yacc, 227

yylen variable in lex, 199

yyless function in lex, 208

yylex, routine in yacc, 227

yylex() lexical analyzer generated by lex, 209

yylval

assignment, 227

variable used in yacc subroutines, 227

yylval variable used in yacc declaration, 226

yymore used to append to yytext, 207

yyvsparse() function in yacc, 230

YYSTYPE union declaration in yacc, 226

yytext character array in lex, 199, 207

yyval, variable used in yacc declaration, 226

yyval variable, 211

yywrap end-of-file processing in lex, 209

Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____

(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____

(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree		Disagree	
The software was fully and accurately described	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation? If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

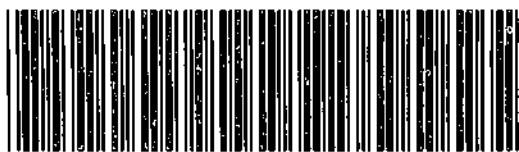
City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____



10 December 1991



BH01 407P000



56703

SCO UNIX Development System

Programming Tools Guide



AU01411P000