

Miscellaneous Tools

In addition to the standard development tools used in software development projects, many other utilities are also helpful. Most of the time these utilities are not directly related to the software development process but are used as helping aids. This chapter provides an introduction to some of these utilities that are extremely helpful in the software development process. Some of the more common uses of these utilities are presented and readers are encouraged to experiment with them. If properly utilized, they may save time and increase productivity.

The `indent` utility is helpful in implementing a common programming style scheme to all source code files in a project. The `sed` utility is useful for searching and replacing text across multiple files. The `diff` command is used to display difference between two or more files. The `cscope` and `cbrowser` are used to locate symbols in a software source code tree. The `strace` and `ltrace` utilities are useful to find out system and library calls when a program is executed. GNU binary utilities are a set of utilities that is often used for different needs. Information about GNU utilities is presented in this chapter as well.

7.1 Using indent Utility

One major objective of every software development project is that the code be well organized, structured and easy to understand for other members of the team. Proper indentation plays an important role in achieving these objectives. Most of the editors used in software development can do automatic indentation if they are configured to do so. The Emacs editor is the best example as it understands different programming languages and does indentation according to language rules and style. However you can come across code written by others that is poorly indented and you may like to re-indent it according to your own requirements or preferences. In large software development projects, you may try to enforce a particular coding style, but be assured that not everyone will follow it. The `indent` program is your tool to handle the situation and reformat all files in the project according to the style adopted for the project. You can do it by creating a script that goes into each directory of the project and reformats all files. In this section we shall explore some of its features and how to utilize it.

By default, the `indent` program uses the GNU style of coding and applies these rules to input files. It creates a backup file with original contents of the file. The reformatted file is created with the same name. The backup file ends with a tilde character `~`. Note that the same scheme of creating backup files is used by the Emacs editor.

Let us see how `indent` works. We will take a poorly indented file and apply the `indent` program to it. Following is a listing of the poorly indented file `hello.c`.

```
1  /*****
2  * hello.c
3  *
4  * This file is used to demonstrate use
5  * of indent utility
6  *****/
7  #include <stdio.h>
8
9  main () {
10     char string[25] ;
11     printf ("Enter a string of characters : ") ;
12     scanf ("%s", string);
13     printf ("The entered string is : %s\n ", string);
14 }
```

Line numbers are added to the listing using the “`cat -n hello.c`” command to explain changes. They are not part of the `hello.c` file. To properly indent the file `hello.c`, the following command is executed:

```
indent hello.c
```

The resulting `hello.c` file is shown below. Note how the `indent` program has modified the file. Lines 10 to 14 are indented to two characters (depending upon default settings).

```
1  /*****
2  * hello.c
3  *
4  * This file is used to demonstrate use
5  * of indent utility
6  *****/
7  #include <stdio.h>
8
9  main ()
10 {
11     char string[25];
12     printf ("Enter a string of characters : ");
13     scanf ("%s", string);
14     printf ("The entered string is : %s\n ", string);
15 }
```

However line 6 is still not properly indented. By default the indent program does not modify comment lines.

To find out which version of the indent program you are using, use the following command.

```
[root@boota ftp-dir]# indent --version
GNU indent 2.2.6
[root@boota ftp-dir]#
```

7.1.1 Getting Started with Indent

The indent program may be used as a command or as a pipe. In the command mode, the program has a general format as shown below:

```
indent [options] [input filename] [-o output filename]
```

The following two lines show the simplest use of the indent program. It takes hello.c as input file and creates hello.c as output file and hello.c~ as backup file.

```
[root@boota indent]# indent hello.c
[root@boota indent]#
```

If you want to create an output file with a different name without modifying the original file, you can use the -o option with indent. When multiple input files are specified, indent reformats each file and creates a backup file for each input file. Wild card characters can also be used with indent so that you can indent all C source code files with the "indent *.c" command.

The output of another command can also be piped into the indent command. The following command does the same thing as indent hello.c but writes the output on STDOUT instead of writing it to a file.

```
cat hello.c | indent
```

This is sometimes useful if you just want to test the behavior of `indent` without modifying a file.

The most efficient use of `indent` is through a shell script that goes into each directory of the source code tree and indents each and every file in the project. You may also create a rule in the makefile of a project to indent all files in one or all directories. A typical makefile rule may look like the following:

```
indent:
  indent *.c
  indent *.cpp
  indent *.h
```

In the case of a project with multiple subdirectories, you can have a more sophisticated rule. Taking an example from Chapter 4, where we have four subdirectories, you can use the following rule to go into each directory and indent all C source code files.

```
SUBDIRS = $(COMDIR) $(FTPDIR) $(TFTPDIR) $(DNSDIR)
indent:
  for i in $(SUBDIRS) ; do \
    ( cd $$i ; indent *.c ) ; \
  done
```

Keep in mind that after indenting all of the files, you may have to build the whole project because source code files have been modified. In big projects this process may take a while.

Indent uses a configuration file called `.indent.pro`, which should be present in the current directory or in your home directory. You can use this file to set options that you always use. If the file is present in the current directory, it is used instead of the file in the home directory of the user. By creating a different `.indent.pro` file in different directories of a project, you can apply different indentation rules in different directories. This is especially useful when a project uses multiple languages. For example, indentation rules may be different for C, C++ and assembly language files.

7.1.2 Selecting Coding Styles

Different coding styles are in use by programmers. Some common styles are defined in `indent` program and these may be invoked by a single option at the command line. The most common style of coding in the open source is the GNU style. This style can be applied using `-gnu` option at the command line. Options, which are used with the GNU style, are listed below:

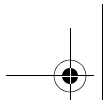
```
-nbad
-bap
-nbc
-bbo
-bl
-bli2
```

```
-bls  
-ncdb  
-nce  
-cp1  
-cs  
-di2  
-ndj  
-nfc1  
-nfca  
-hn1  
-i2  
-ip5  
-lp  
-pcs  
-nprs  
-psl  
-saf  
-sai  
-saw  
-nsc  
-nsob
```

A list of all the options and their meanings are presented later in this chapter. You can override a particular option in a style by explicitly typing it on the command line.

The other commonly used style is Kernighan and Ritchie style, also known as K&R style. This style is applied using the `-kr` option at the command line and it sets the following options.

```
-nbad  
-bap  
-bbo  
-nbc  
-br  
-brs  
-c33  
-cd33
```

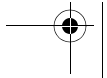
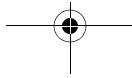


```
-ncdb
-ce
-ci4
-clip0
-cp33
-cs
-d0
-di1
-nfc1
-nfca
-hnl
-i4
-ip0
-l75
-lp
-npcs
-nprs
-npsl
-saf
-sai
-saw
-nsc
-nsob
-nss
```

Again, you can override some of these options by specifying them on the command line.

The Berkley style is used by `-orig` option at the command line and it sets the following options.

```
-nbad
-nbap
-bbo
-bc
-br
-brs
```



```
-c33
-cd33
-cdb
-ce
-ci4
-cli0
-cp33
-di16
-fc1
-fca
-hnl
-i4
-ip4
-l75
-lp
-npcs
-nprs
-psl
-saf
-sai
-saw
-sc
-nsob
-nss
-ts8
```

7.1.3 Blank Lines and Comments

Blank lines can be used in the source code files to add clarity. The `indent` program allows you to add or remove blank lines from the source code files. Most commonly used options to handle blank lines are as follows:

- bad This option adds a blank line after declarations.
- bap This option is used to add a blank line after a procedure body. Using this option will create a blank line after each function.

- bbb This option is used to add a blank line before a boxed comment. An example of a boxed comment is shown below.

```

/*****
 * hello.c
 *
 * This file is used to demonstrate use
 * of indent utility
 *****/

```

- sob This option is used to remove unnecessary blank lines from the source code.

By adding `n` in the start of these options, their effect may be reversed. For example, the `-nbad` option will not add any blank line after declaration.

7.1.4 Formatting Braces

People have different tastes about how braces should be formatted in C language. Using `indent` you can specify different ways of formatting braces. Let us consider a poorly formatted piece of C source code and see different ways of formatting it. The input segment of source code is shown below:

```

if (counter > 0)
{
counter-- ;
printf ("Counter value is: %d \n");
}
else
{
printf("Counter reached zero. Resetting counter\n");
counter = 100;
}

```

The default indentation of this segment of code is shown below. This is the default GNU indentation when you use no command line switches with `indent`.

```

if (counter > 0)
{
counter--;
printf ("Counter value is: %d \n");
}
else
{
printf ("Counter reached zero. Resetting counter\n");
counter = 100;
}

```


Among other things, note that `indent` has placed a space after the second `printf` function call. This is the default style of putting braces in C code. Since the GNU formatting style is used with GNU `indent` by default, and it uses the `-bl` option, the result is as shown above. To put the starting brace on the same line, you can use the `-br` command line switch and the result is as follows:

```

if (counter > 0) {
    counter--;
    printf ("Counter value is: %d \n");
}
else {
    printf ("Counter reached zero. Resetting counter\n");
    counter = 100;
}

```

Other forms of `indent` can be used with different statements. See the manual pages of `indent` for a detailed list of all options.

7.1.5 Formatting Declarations

You can tell `indent` to handle the variable declarations in other ways by using different options to separate the declaration lines from other code. The most common method is to place a blank line after the declaration's end. As an example, if you use the `-bad` option (blank line after declarations), `indent` will place a blank line wherever it finds end of the declaration part. In the previous example of `hello.c` program, the result of using this option will be as follows:

```

1  /*****
2  * hello.c
3  *
4  * This file is used to demonstrate use
5  * of indent utility
6  *****/
7  #include <stdio.h>
8
9  main ()
10 {
11     char string[25];
12
13     printf ("Enter a string of characters : ");
14     scanf ("%s", string);
15     printf ("The entered string is : %s\n ", string);
16 }

```

Note that line number 12 is inserted into the file. You can also use the `-di` option with `indent` to align an identifier to a particular column. For example, using `-di8` will align all identifiers to column number 8. Consider the following two lines in a C source code file.

```
int i;  
char c;  
long boota ;
```

After using `indent` with `-di8` option, the output will be as follows:

```
int     i;  
char    c;  
long    boota;
```

You can also force creation of a new line for each identifier, using the `-bc` option. Consider the following code segment:

```
int i, j, k;  
char c;  
long boota ;
```

After using the “`indent -di8 -bc`” command on these three lines, the result will be as follows:

```
int     i,  
        j,  
        k;  
char    c;  
long    boota;
```

In addition to these common methods, there are other ways to arrange identifiers in declaration sections.

7.1.6 Breaking Long Lines

Long lines can be broken using `indent` at a defined length. The `-l` option controls this behavior. Consider the following line of code:

```
printf ("This is example of a long line.");
```

This line is 43 characters long. If we set the line limit to 40 characters using `-l40` command line option, the line will be broken as follows:

```
printf  
("This is example of a long line.");
```

Lines with conditional operators may be broken more intelligently. Consider the following `if` statement:

```
if(((counter == 100) && (color == RED)) || (string[0] == 'S'))
```

Using options “`-l30 -bbo`” (break before Boolean operator) will be as follows:

```
if (((counter == 100)  
    && (color == RED))  
    || (string[0] == 'S'))
```

Using the `-nbbo` option results in the following output.

```
if (((counter == 100) &&
    (color == RED)) ||
    (string[0] == 'S'))
```

The `indent` utility may also be used to handle new line characters in a special way using the `-hnl` option. You are encouraged to experiment with this option.

7.1.7 Summary of Options

Options with the `indent` program can be used in two ways: the long way that starts with two hyphen characters and the short way that starts with a single hyphen character. Options used with the `indent` command are listed below. These are taken from the manual page of the `indent` command. The long method also describes the meaning of an option.

<code>-bc</code>	<code>--blank-lines-after-commas</code>
<code>-bad</code>	<code>--blank-lines-after-declarations</code>
<code>-bap</code>	<code>--blank-lines-after-procedures</code>
<code>-bbb</code>	<code>--blank-lines-before-block-comments</code>
<code>-bl</code>	<code>--braces-after-if-line</code>
<code>-bli</code>	<code>--brace-indent</code>
<code>-bls</code>	<code>--braces-after-struct-decl-line</code>
<code>-br</code>	<code>--braces-on-if-line</code>
<code>-brs</code>	<code>--braces-on-struct-decl-line</code>
<code>-nbbo</code>	<code>--break-after-boolean-operator</code>
<code>-bbo</code>	<code>--break-before-boolean-operator</code>
<code>-bfda</code>	<code>--break-function-decl-args</code>
<code>-clin</code>	<code>--case-indentation</code>
<code>-cbin</code>	<code>--case-brace-indentation</code>
<code>-cdb</code>	<code>--comment-delimiters-on-blank-lines</code>
<code>-cn</code>	<code>--comment-indentation</code>
<code>-cin</code>	<code>--continuation-indentation</code>
<code>-lp</code>	<code>--continue-at-parentheses</code>
<code>-cdw</code>	<code>--cuddle-do-while</code>
<code>-ce</code>	<code>--cuddle-else</code>
<code>-cdn</code>	<code>--declaration-comment-column</code>
<code>-din</code>	<code>--declaration-indentation</code>
<code>-nbfda</code>	<code>--dont-break-function-decl-args</code>
<code>-npsl</code>	<code>--dont-break-procedure-type</code>
<code>-ncdw</code>	<code>--dont-cuddle-do-while</code>
<code>-nce</code>	<code>--dont-cuddle-else</code>
<code>-nfca</code>	<code>--dont-format-comments</code>
<code>-nfc1</code>	<code>--dont-format-first-column-comments</code>
<code>-nlp</code>	<code>--dont-line-up-parentheses</code>
<code>-nss</code>	<code>--dont-space-special-semicolon</code>
<code>-nsc</code>	<code>--dont-star-comments</code>
<code>-cpn</code>	<code>--else-endif-column</code>

-fca	--format-all-comments
-fcl	--format-first-column-comments
-gnu	--gnu-style
-hnl	--honour-newlines
-nhnl	--ignore-newlines
-npro	--ignore-profile
-in	--indent-level
-kr	--k-and-r-style
-nsob	--leave-optional-blank-lines
-lps	--leave-preprocessor-space
-dn	--line-comments-indentation
-ln	--line-length
-nbc	--no-blank-lines-after-commas
-nbad	--no-blank-lines-after-declarations
-nbap	--no-blank-lines-after-procedures
-nbbb	--no-blank-lines-before-block-comments
-ncdb	--no-comment-delimiters-on-blank-lines
-ncs	--no-space-after-casts
-nip	--no-parameter-indentation
-nsaf	--no-space-after-for
-npcs	--no-space-after-function-call-names
-nsai	--no-space-after-if
-nprs	--no-space-after-parentheses
-nsaw	--no-space-after-while
-nut	--no-tabs
-nv	--no-verbosity
-orig	--original
-ipn	--parameter-indentation
-pin	--paren-indentation
-pmt	--preserve-mtime
-psl	--procnames-start-lines
-cs	--space-after-cast
-saf	--space-after-for
-sai	--space-after-if
-prs	--space-after-parentheses
-pcs	--space-after-procedure-calls
-saw	--space-after-while
-ss	--space-special-semicolon
-st	--standard-output
-sc	--start-left-side-of-comments
-sbin	--struct-brace-indentation
-sob	--swallow-optional-blank-lines
-tsn	--tab-size
-ut	--use-tabs
-v	--verbose

7.2 Using sed Utility

The `sed` utility is a stream editor that can be used for different file editing purposes when used as a filter. The most common task for software development purposes is the use of `sed` to search and replace text in source code files. Let us take the example of the following C source code file `hello.c`.

```
#include <stdio.h>

main ()
{
    char string[25];

    printf ("Enter a line of characters : ");
    scanf ("%s", string);
    printf ("The entered string is : %s\n ", string);
}
```

In order to replace every occurrence of word *string* with the word *STRING*, you can use `sed`. The `sed` filter command and its result on this file are shown below.

```
[root@boota]# cat hello.c | sed s/string/STRING/
#include <stdio.h>

main ()
{
    char STRING[25];

    printf ("Enter a line of characters : ");
    scanf ("%s", STRING);
    printf ("The entered STRING is : %s\n ", string);
}
[root@boota indent]#
```

The `sed` command understands UNIX regular expressions. Regular expressions can be used for a higher level of stream editing. You can also use `sed` in shell scripts as well as makefiles to carry out tasks in the entire source code directory tree. You can also use `-f` options followed by a filename. The filename contains `sed` commands. Please refer to the `sed` man pages for a complete set of options.

7.3 Using diff Utility

The `diff` utility is another useful tool that developers may need. It is used to find out the differences between two files. If you are using CVS, differences between different versions of a file in the CVS repository can be found using the `cv`s (`cv`s `diff`) command as well. However, if you want to find out the difference between two files that are not in the CVS repository, the `diff` utility may be quite handy. One common use may be to find out the difference between the working copy and the backup copy of a source code file. This will enable you to find out

what changes have been made in the working copy of the file. The output of the `diff` utility follows similar rules to those used in the CVS `diff` command. The following command shows that files `hello.c` and `hello.c~` are different at line number 11. The line starting with the less-than symbol is taken from the first file (`hello.c`) and the line starting with the greater-than symbol is taken from the file `hello.c~`.

```
[root@boota]# diff hello.c hello.c~
11c11
< char string[30];
---
> char string[25];
[root@boota]#
```

The first line of the output contains the character `c` (changed) that shows that line 11 in the first file is changed to line 11 in the second file.

You can also use “unified `diff`” that tells you additional information about the file and displays a few lines before and after the lines that are different. See the following output of the unified `diff` command:

```
[root@boota]# diff hello.c hello.c~ -u
--- hello.cTue Jun 25 14:43:30 2002
+++ hello.c~Tue Jun 25 14:43:38 2002
@@ -8,7 +8,7 @@

main ()
{
- char string[30];
+ char string[25];

printf ("Enter a line of characters : ");
scanf ("%s", string);
[root@boota]#
```

You can also use the `-p` option with the command to display the name of the function in which the modified line(s) exist.

If you add a line after line 15 in `hello.c` file and run the `diff` command once again, the result will be as follows:

```
[root@boota]# diff hello.c hello.c~
11c11
< char string[30];
---
> char string[25];
16d15
< printf ("End of program\n");
[root@boota]#
```

The first part of the output is familiar. The second part shows that line 16 in the first file, which is printed next, is not present in the second file. Sometimes it is useful to display two files being compared in side-by-side format. This can be done by using the `-y` option. The following command shows how to use side-by-side output. The CVS rules of displaying output are used here to show modified, added or removed lines.

```
[root@boota]# diff -y --width 60 hello.c hello.c~
/*****                               /*****
 * hello.c                             * hello.c
 *                                     *
 * This file is used to demo           * This file is used to demo
 * of indent utility                   * of indent utility
 *****/                               *****/
#include <stdio.h>                     #include <stdio.h>

main ()                               main ()
{                                       {
    char string[30];                   |   char string[25];

    printf ("Enter a string of         printf ("Enter a string of
scanf ("%s", string);                 scanf ("%s", string);
printf ("The entered strin           printf ("The entered strin
printf ("End of program\n" <
}                                       }
[root@boota]#
```

The `--width` option is used to specify width of columns in the output. As you can see the `|` symbol is used to show a modified line and `<` or `>` symbols are used to show added or deleted lines.

The `diff` utility can also be used on directories. When comparing directories, `diff` compares files of the same name in the two directories. Directories can also be compared recursively.

Many options are used with the `diff` command, the most common are listed in Table 7-1. All these options start with a hyphen character.

Table 7-1 Common options used with the `diff` command

Option	Description
<code>-b</code>	Ignore changes in white spaces
<code>-B</code>	Ignore changes that are related to insertion or deletion of blank lines
<code>-i</code>	Ignore changes in case
<code>-u</code>	Unified output
<code>-p</code>	Used with <code>-u</code> option. Shows the function name also.

Table 7-1 Common options used with the `diff` command (Continued)

Option	Description
<code>-c</code>	Context output
<code>-n</code>	RCS style of output
<code>-r</code>	Compare directories recursively
<code>-y</code>	Use side-by-side format

7.3.1 Other Forms of `diff` Utility

There are two other important forms of the `diff` utility. These are `diff3` and `sdiff`. The `diff3` utility is used to compare three files and its general format is shown below.

```
diff3 [options] mine older yours
```

Suppose you and your colleague start working on a file simultaneously. The original file is the *older* file. Now you have your copy of the file (the *mine* file) and your colleague has his own copy of the file (the *yours* file). If you want to compare both of these modified copies of the file with the original *older* file, `diff3` is a useful tool. See man pages of the `diff3` command for more details.

Another important utility is `sdiff` that finds difference between two files and merges these two files into a third file. The general format of `sdiff` is as follows.

```
sdiff -o outfile [options] file1 file2
```

The `sdiff` is useful when you want to interactively merge two files. This is the case when two people have made changes to a source file and at some point you want to merge these changes into a single file. The `sdiff` utility is interactive and it displays two files being compared in side-by-side fashion. It stops on each difference with a `%` sign prompt. On this sign you can press different characters to make a decision. Common responses on the `%` prompt are shown in Table 7-2.

Table 7-2 Commands used on `%` prompt of `sdiff`

Command	Description
<code>L</code>	Use the left side of the version
<code>R</code>	Use the right side of the version
<code>e l</code>	Edit and then use the left side
<code>e r</code>	Edit and use the right side
<code>Q</code>	Quit

7.4 Using cscope and cbrowser

The `cscope` is a very useful utility to browse through the source code tree of a large project. It was originally written by Santa Cruz Operations and made public later on. You can download it from <http://cscope.sourceforge.net>. For Linux, it is available in RPM format as well.

It is a text-mode screen-oriented utility. When you start it using the `cscope` command, the initial screen looks like the one shown in Figure 7-1.

When invoked, `cscope` first creates its symbol file `cscope.out` in the current directory that contains a reference to symbols in source code files. This reference is generated from files with the extensions `.c` and `.h`. You can also use command line options to create this symbol file from a particular type of source code files.

The bottom part of the screen shows options that can be used to look up a symbol in the source code. The cursor is blinking at the first option and can be moved to other options using arrow keys. For example, if you want to look up symbol “`msg`” in all source code files, just type it in at the first line and press the Enter key. You will see a listing of files containing this symbol as shown in Figure 7-2.

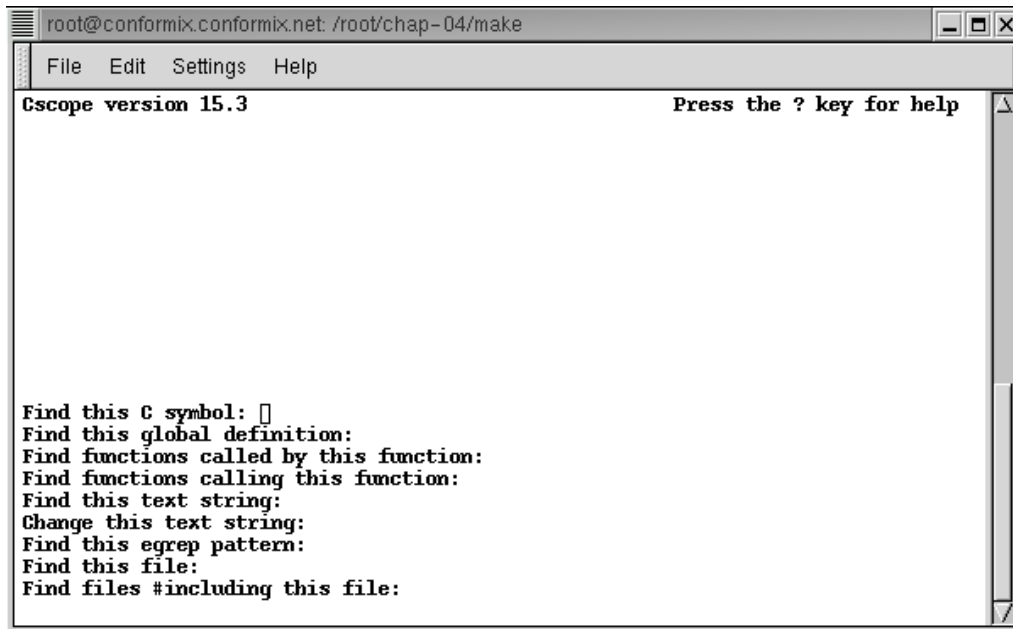
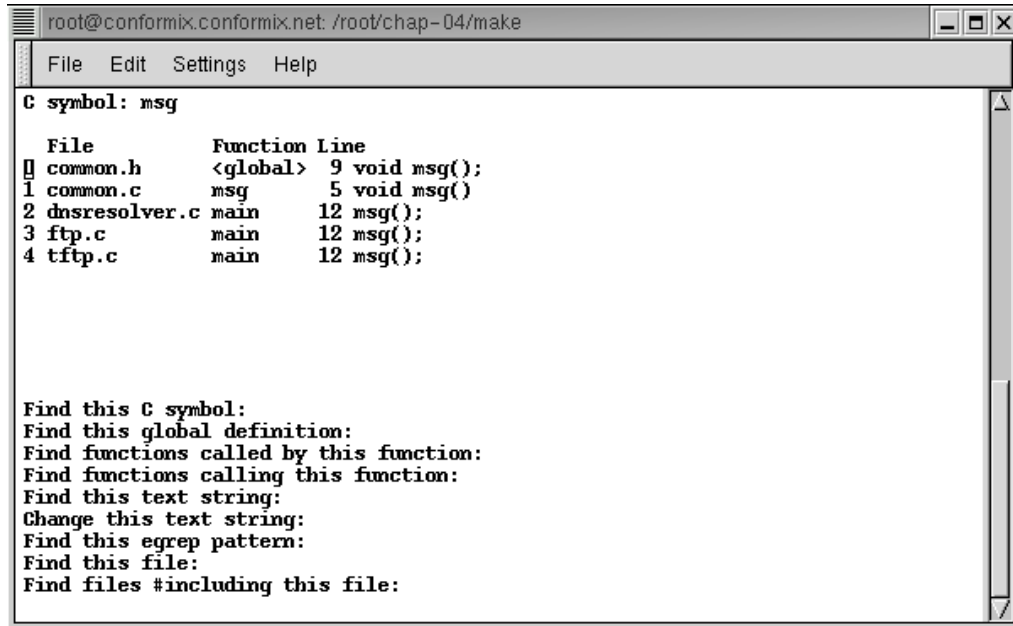


Figure 7-1 The `cscope` initial screen.



```

root@conformix.conformix.net: /root/chap-04/make
File Edit Settings Help
C symbol: msg

File      Function Line
0 common.h <global> 9 void msg();
1 common.c msg      5 void msg()
2 dnsresolver.c main   12 msg();
3 ftp.c    main   12 msg();
4 tftp.c   main   12 msg();

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:

```

Figure 7-2 List of files with symbol msg.

As you can see, there are five files listed in the top part of the screen that contain this symbol. This listing contains five columns as below:

1. Number of row, starting with zero.
2. File name that contains the symbol.
3. Function name where symbol is used. If the symbol is not inside a function, it is marked as global.
4. Line number where symbol is present in that file.
5. The line showing the definition or use of the symbol.

You can move the cursor to a particular line and press the Enter key to edit that file. By default the file is opened in vi editor. However, you can configure to use an editor of your choice. To move back to the bottom part of the screen to search for another symbol, you can use the Tab key. Use the Ctrl+D key combination to quit the program.

The utility is also very useful if you want to find non-utilized code in a software project. For example, if a function is present in the source code but never used anywhere in the project, it can be detected using `cscope`. It is also useful when you want to modify a particular symbol throughout the source code tree. Common options used with `cscope` are listed in Table 7-3.

Table 7-3 Options used with `cscope`

Option	Description
-b	Build only the cross-reference file
-C	Ignore case when searching
-f <i>reffile</i>	Use <i>reffile</i> as reference file instead of default <code>cscope.out</code> reference file. This is useful when you create a global reference file for the entire source code tree.
-R	Recursively search source code tree for input files

Use the manual pages of `cscope` to view a complete list of options. Its web site is also a good reference for updates and new features.

The `cbrowser` is a GUI interface to `cscope` and can be downloaded from its web site, <http://cbrowser.sourceforge.net>.

At the time of writing this book, version 0.8 of this utility is available. When you invoke `cbrowser`, it displays its initial window where you can select a `cscope` symbol reference file and use the same type of queries as are available on `cscope` text window. The `cbrowser` window is shown in Figure 7-3.

To use a particular `cscope` symbol file, use the “Selected Database” drop-down menu in the GUI. Click the “Symbols” button to pull down a menu of different options used with `cscope`. The box next to this button is used to type in the symbol for which you want to search. When you press the “Submit” button after typing in the symbol name, `cbrowser` displays a list of files where this symbol is used or defined as shown in Figure 7-3. An additional benefit of using `cbrowser` is that the bottom part of the window also shows the contents of the selected file. You can use cursor keys to select a different file. You can also edit these files by using options in the “File” menu. You can use syntax highlighting and other helping tools with `cbrowser` using menus.

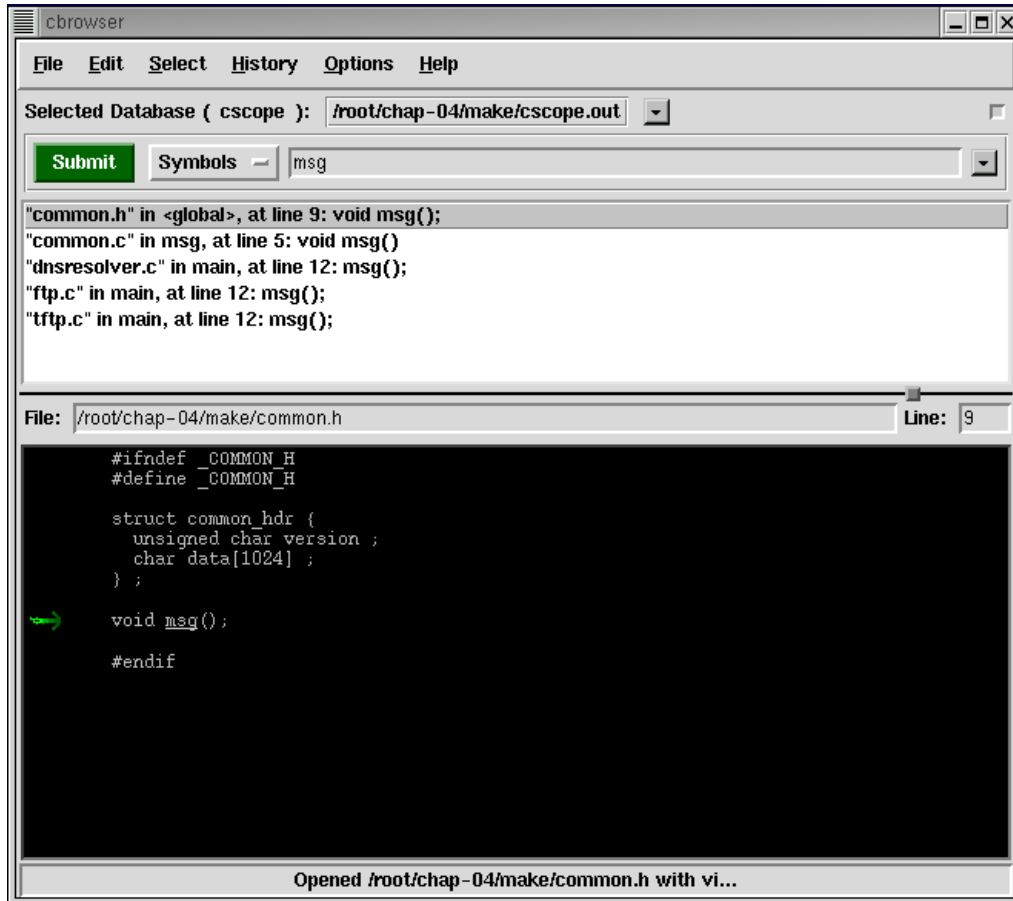


Figure 7-3 The cbrowser window.

7.5 Generating C Function Prototypes from C Source Code Using cproto

The `cproto` utility is used to create function prototypes from C source code files. It can also convert function definition style. The latest version can be downloaded from <http://sourceforge.net/projects/cproto/>.

At the time of writing this book, version 4.6 is available in source code form. You can download the file, untar it and compile it using the following sequence of commands.

```
tar zxvf cproto-4.6.tar.gz
cd cproto-4.6
./configure
make
make install
```

It can read existing C files or take its input from standard input. This utility is not extensively used in C software development but may be useful in some cases.

7.6 Using ltrace and strace Utilities

The `ltrace` program is a tracing utility for library function calls. It runs a program and logs all library function calls by that program. You can also use this utility to log system calls made by a program. The utility can also monitor child processes created by `fork()` or `clone()` system calls. This utility is very useful to quickly trace the failure point of an executable program. The utility may also print the time at which a particular function call or system call is executed with a resolution of microseconds.

Consider the simple single-line program that prints the string "Hello world" and then exits. Using `ltrace` with the executable of this program produces the following result.

```
[root@boota ltrace]# ltrace -S -tt ./a.out
22:21:48.325970 SYS_uname(0xbffff3b4) = 0
22:21:48.327037 SYS_brk(NULL) = 0x080495f8
22:21:48.327511 SYS_mmap(0xbffff104, 0xcccccccd, 0x400165f8,
4096, 640) = 0x40017000
22:21:48.328212 SYS_open("/etc/ld.so.preload", 0, 010) = -2
22:21:48.329000 SYS_open("/etc/ld.so.cache", 0, 00) = 3
22:21:48.329657 SYS_197(3, 0xbfffea64, 0, 0xbfffea64, 0) = 0
22:21:48.331719 SYS_mmap(0xbfffea34, 0, 0x400165f8, 1, 3) =
0x40018000
22:21:48.332460 SYS_close(3) = 0
22:21:48.332908 SYS_open("/lib/i686/libc.so.6", 0,
027777765514) = 3
22:21:48.333620 SYS_read(3, "\177ELF\001\001\001", 1024) =
1024
22:21:48.334256 SYS_197(3, 0xbfffeaa4, 3, 0xbfffeaa4, 0) = 0
22:21:48.334917 SYS_mmap(0xbfffe994, 0x0012f728, 0x400165f8,
0xbfffe9c0, 5) = 0x4002c000
22:21:48.335584 SYS_mprotect(0x40152000, 38696, 0, 0x4002c000,
0x00126000) = 0
22:21:48.336209 SYS_mmap(0xbfffe994, 24576, 0x400165f8,
0xbfffe9cc, 3) = 0x40152000
22:21:48.336953 SYS_mmap(0xbfffe994, 0xbfffe9cc, 0x400165f8,
0x40158000, 14120) = 0x40158000
22:21:48.337642 SYS_close(3) = 0
22:21:48.340431 SYS_munmap(0x40018000, 77871) = 0
22:21:48.341060 SYS_getpid() = 32540
22:21:48.341562 __libc_start_main(0x08048460, 1, 0xbffff88c,
0x080482e4, 0x080484c0 <unfinished ...>
22:21:48.342232 __register_frame_info(0x08049508, 0x080495e0,
0xbffff828, 0x0804838e, 0x080482e4) = 0x401575e0
22:21:48.343064 printf("Hello world\n" <unfinished ...>
```

```

22:21:48.343813 SYS_197(1, 0xbffefff0, 0x401569e4, 0x40154a60,
0x40154a60) = 0
22:21:48.344450 SYS_192(0, 4096, 3, 34, -1)          = 0x40018000
22:21:48.345154 SYS_ioctl(1, 21505, 0xbffef20, 0xbffef80,
1024) = 0
22:21:48.345890 SYS_write(1, "Hello world\n", 12Hello world
) = 12
22:21:48.346542 <... printf resumed> )              = 12
22:21:48.346878 __deregister_frame_info(0x08049508,
0x4000d816, 0x400171ec, 0x40017310, 7) = 0x080495e0
22:21:48.347746 SYS_munmap(0x40018000, 4096)        = 0
22:21:48.348235 SYS_exit(12)                        = <void>
22:21:48.348706 +++ exited (status 12) +++
[root@boota ltrace]#

```

The first column in each row shows the current time followed by a number that shows microseconds. The remaining part of the line shows the system call of the library call used. By comparing the time in two consecutive lines, you can find out the time taken by a particular system call. By looking at the output, you can also find out if a program fails during a particular system call. This may be especially helpful when testing device drivers and reading or writing to these drivers fails for some reason.

The most common options used with this utility are shown in Table 7-4.

Table 7-4 Common options used with the `ltrace` utility

Option	Description
-d	Debug. Displays extra information about trace.
-f	Trace child processes.
-S	Display system calls and library calls.
-r	Display time difference between successive lines.
-tt	Display time stamp with each line with a resolution to microseconds.
-o filename	Record output to a file.
-v	Display version.
-h	Display help.

The command can also read options either from its configuration file `/etc/ltrace.conf` or from `.ltrace.conf` in the home directory of the user.

The `strace` utility is a more comprehensive tool and it can be used to separate different system calls. For example, it can be used to display information about network related system

calls or for IPC related systems calls only. It displays arguments for all functions calls and their return values. A typical output from the execution of the same single line function follows:

```
[root@boota ltrace]# strace ./a.out
execve("./a.out", ["/a.out"], [/* 44 vars */]) = 0
uname({sys="Linux", node="boota.boota.net", ...}) = 0
brk(0) = 0x80495f8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40017000
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=77871, ...}) = 0
old_mmap(NULL, 77871, PROT_READ, MAP_PRIVATE, 3, 0) =
0x40018000
close(3) = 0
open("/lib/i686/libc.so.6", O_RDONLY) = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200\302"...
, 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=5634864, ...}) = 0
old_mmap(NULL, 1242920, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3,
0) = 0x4002c000
mprotect(0x40152000, 38696, PROT_NONE) = 0
old_mmap(0x40152000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x125000) = 0x40152000
old_mmap(0x40158000, 14120, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40158000
close(3) = 0
munmap(0x40018000, 77871) = 0
getpid() = 32610
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3),
...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40018000
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "Hello world\n", 12Hello world
) = 12
munmap(0x40018000, 4096) = 0
_exit(12) = ?
[root@boota ltrace]#
```

As you can see, it gives you a complete picture of what happens from the execution of a program to its end. See the manual pages of `strace` for a complete list of command line options. The `strace` utility can also be used for performance-gathering activities, and can log times spent in each function in a program. Refer to the man page for more information.

7.7 Using GNU Binary Utilities

GNU binutils is a package of many utilities that are related to manipulating library files or getting information about different types of binary files. These utilities range from finding strings in a binary file, creating library files, displaying information about binary files, assembling and disassembling files, and so on. This section contains an introduction to most commonly used binary utilities.

7.7.1 Using the ar Utility

Files can be stored into a single file using the `ar` utility. The file into which multiple files are stored is called an *archive*. The `ar` program retains file permissions, ownerships and other properties of the original files. These files can be extracted from the archive later on, if required. The program also creates an index within the archive file that is used to speed up locating a component (or *members*) of the archive.

From the development point of view, the `ar` program is used to create libraries of functions. These libraries are then used with other object files to link an executable file. The conventional name of these library files is `lib{name}.a` where *name* is any string used as a library name. For example, a library named *common* will be stored in a file called `libcommon.a`.

The `ar` program performs one of the following tasks depending upon options used on the command line.

- Delete a file from the archive (option `d`)
- Move files in the archive (option `m`)
- Print a list of files in the archive (option `p`)
- Append files to the archive (option `q`)
- Insert new files in the archive by replacing if a file already exists in the archive (option `r`)
- Displaying contents of an archive (option `t`)
- Extracting files from an archive (option `x`)

More options can be used in combination with these options to perform additional tasks. In this section we shall use some very common options to show you how the `ar` program is used in a software development environment.

We have used `ar` version 2.10.91 in this book. You can display a current version of the program on your system using `-V` option as shown below. The `ar` utility is usually used because of its compatibility with `ld`, the dynamic loader.

```
[root@boota ar]# ar -V
GNU ar 2.10.91
Copyright 1997, 98, 99, 2000, 2001 Free Software Foundation,
Inc.
This program is free software; you may redistribute it under
the terms of the GNU General Public License. This program has
absolutely no warranty.
[root@boota ar]#
```


Let us create an archive of simple text files. The following command creates an archive `test.a` from files `/etc/hosts` and `/etc/resolv.conf`.

```
ar -r test.a /etc/resolv.conf /etc/hosts
```

This command creates a file `test.a` that contains the contents of the other two files. To display the contents of the file, you can use the `-t` option on the command line.

```
[root@boota ar]# ar -t test.a
resolv.conf
hosts
[root@boota ar]#
```

Using `-x` option, you can extract one or more files from the archive. The following command extracts the file `hosts` from the archive.

```
ar -x test.a hosts
```

Now let us create a real library file from two object files `common.o` and `ftp.o`. The following command creates a file `libcommon.a` from these two object files.

```
ar -r libcommon.a common.o ftp.o
```

Functions or variables in these two files can be linked to any other object file using the `-l` command line option with `gcc` compiler. The following command line creates an executable file `project` from this library and `project.o` file.

```
gcc project.o -lcommon -o project
```

Note that `libcommon.a` is not used with the `-l` option when linking functions in this library to an executable file. Instead we used `common` as the library name by stripping the leading `lib` part and `.a` trailing part.

Files can be inserted into the archive in a particular order. Similarly you can also use a policy when replacing existing files using different command line options. Other options that can be used with the `ar` command can be listed by executing the command without an argument as shown below.

```
[root@boota /root]# ar
Usage: ar [-X32_64] [-]{dmpqrstx}[abcfilNoPsSuvV] [member-
name] [count] archive-file file...
       ar -M [<mri-script]
commands:
  d             - delete file(s) from the archive
  m[ab]        - move file(s) in the archive
  p            - print file(s) found in the archive
  q[f]         - quick append file(s) to the archive
  r[ab][f][u] - replace existing or insert new file(s) into
the archive
  t            - display contents of archive
  x[o]        - extract file(s) from the archive
```

```

command specific modifiers:
[a]          - put file(s) after [member-name]
[b]          - put file(s) before [member-name]
              (same as [i])

[N]          - use instance [count] of name
[f]          - truncate inserted file names
[P]          - use full path names when matching
[o]          - preserve original dates
[u]          - only replace files that are newer than
              current archive contents

generic modifiers:
[c]          - do not warn if the library had to be
              created
[s]          - create an archive index (cf. ranlib)
[S]          - do not build a symbol table
[v]          - be verbose
[V]          - display the version number
[-X32_64]    - (ignored)
ar: supported targets: elf32-i386 a.out-i386-linux efi-app-
ia32 elf32-little elf32-big srec symbolsrec tekhex binary ihex
trad-core
[root@boota /root]#

```

7.7.2 Using the ranlib Utility

The `ranlib` command is used to create index entries inside an archive file. This can also be done using `-s` command with the `ar` command while creating or updating an archive library file. The following command creates index inside `libcommon.a` file. Note that these index entries are stored inside the archive file and no other file is created.

```
ranlib libcommon.a
```

Most of the time you don't need to run `ranlib`, as the latest version of the `ar` command creates an index by itself. Index entries can be displayed using the `nm` command.

7.7.3 Using the nm Utility

The `nm` utility is used to list symbols used in an object file. If no object file is provided on the command line, the utility assumes the `a.out` file in the current directory as the default object file and lists its symbols. A common use of the utility is listing functions defined in a library. The following command displays object file names and functions defined in the library file created with `ar` command in the previous section.

```
[root@boota]# nm -s libcommon.a
```

```

Archive index:
msg in common.o
main in ftp.o

```

```
common.o:
00000000 T msg
          U printf

ftp.o:
00000000 T main
          U msg
          U printf
[root@boota make]#
```

For each symbol, nm displays three properties of the symbol.

1. Value
2. Type
3. Name

The *value* of the symbol is displayed in hexadecimal by default. *Type* is a character, either uppercase or lowercase. The uppercase character shows that the symbol is global and the lowercase character shows that the symbol is local. The following line shows that the symbol value is 00000000, its type is T which shows that the symbol is in code section and its name is msg.

```
00000000 T msg
```

Type U in the above output of nm command shows that the symbol is undefined and no value is displayed. Note that function msg is defined in common.o but undefined in the ftp.o member of the archive. This is because of the fact that the function was defined in common.c file and it is used in ftp.c file.

Usually, the list of symbols is very long in executable files. Consider the following simple file that displays the string “Hello world”.

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

Let us compile this file to create an executable a.out file. A list of symbols in the a.out executable file is shown below:

```
[root@boota]# nm -s a.out
08049540 ? __DYNAMIC
0804951c ? __GLOBAL_OFFSET_TABLE__
080484e4 R __IO_stdin_used
08049510 ? __CTOR_END__
0804950c ? __CTOR_LIST__
08049518 ? __DTOR_END__
08049514 ? __DTOR_LIST__
```

```

08049508 ? __EH_FRAME_BEGIN__
08049508 ? __FRAME_END__
080495e0 A __bss_start
      w __cxa_finalize@@GLIBC_2.1.3
080494f8 D __data_start
      w __deregister_frame_info@@GLIBC_2.0
08048480 t __do_global_ctors_aux
080483b0 t __do_global_dtors_aux
      w __gmon_start__
      U __libc_start_main@@GLIBC_2.0
      w __register_frame_info@@GLIBC_2.0
080495e0 A _edata
080495f8 A _end
080484c0 ? _fini
080484e0 R _fp_hw
080482e4 ? _init
08048360 T _start
08048384 t call_gmon_start
08049504 d completed.1
080494f8 W data_start
08048410 t fini_dummy
08049508 d force_to_data
08049508 d force_to_data
08048420 t frame_dummy
08048384 t gcc2_compiled.
080483b0 t gcc2_compiled.
08048480 t gcc2_compiled.
080484c0 t gcc2_compiled.
08048460 t gcc2_compiled.
08048450 t init_dummy
080484b0 t init_dummy
08048460 T main
080495e0 b object.2
08049500 d p.0
      U printf@@GLIBC_2.0
[root@boota make]#

```

7.7.3.1 Listing Line Numbers in Source Files

Use of `-l` option is very useful with this command. This option also displays filenames and line numbers where these symbols are used. The following command lists symbols in `libcommon.a` file in more detail. Note that it displays the path to the source code file and the line number. This information is used in the debugging process.

```
[root@boota]# nm -s libcommon.a -l
```

```

Archive index:
msg in common.o
main in ftp.o

```

```
common.o:
00000000 T msg /root/make/common.c:6
          U printf /root/make/common.c:7

ftp.o:
00000000 T main /root/make/ftp.c:7
          U msg /root/make/ftp.c:12
          U printf /root/make/ftp.c:11
[root@boota make]#
```

7.7.3.2 Listing Debug Symbols

When you compile a program with a debug option, many symbols are inserted in the files that are used for debugging purposes. Using the `-a` option with the `nm` command also shows the debug symbols.

Please see the manual pages of the `nm` command for a detail description of all options.

7.7.4 Using the strip Utility

The `strip` command is used to remove symbols from an object or library file. This is useful to reduce the size of the shipped product as these symbols are not required in enduser executable code. Using the command, you can also remove symbols partially. For example, you can remove only debug symbols, local symbols or all symbols with the help of command line options.

At least one object file must be provided at the command line. Note that the `strip` utility modifies the object files; it does not create new files. To get an idea of the difference in size of object file before and after using the `strip` command, let us take the example of a C file that prints the “Hello world” string only. The size of the executable `a.out` file with symbols on my computer is 13640 bytes. After using the following command, the size is reduced to 3208 bytes.

```
strip a.out
```

This is a considerable reduction in size. However, in some time-sensitive and embedded systems, stripping files may cause timing problems and code may behave differently in some cases.

7.7.5 Using the objcopy Utility

The basic function of the `objcopy` utility is to copy one object file to another. This functionality can be used to change the format of an object file. A common use is to create S-record or binary files from ordinary object files. S-record or binary files can be used to burn ROM in embedded systems. The following command converts the `ftp` file (which is a statically linked file) to an S-record file `ftp.S`, which may be downloaded to a ROM/PROM using the EPROM programmer.

```
objcopy -O srec ftp ftp.S
```

You can see types of both input and output files by using the `file` command as follows. The output file is of type S-record.

```
[root@boota]# file ftp
ftp: ELF 32-bit LSB executable, Intel 80386, version 1,
statically linked, not stripped
[root@boota]#
[root@boota make]# file ftp.S
ftp.S: Motorola S-Record; binary data in text format
[root@boota]#
```

The S-record file is in ASCII hex format and first few lines can be displayed using head command as shown below:

```
[root@boota]# head ftp.S
S00800006674702E532C
S31508048094040000001000000001000000474E5500CB
S315080480A40000000020000000200000005000000B1
S315080480B45589E583EC08E84500000090E8DB0000F0
S30D080480C400E836610400C9C393
S315080480E031ED5E89E183E4F05054526880E2080871
S315080480F068B4800408515668E0810408E80B010056
S3150804810000F489F65589E55350E8000000005B81C0
S31508048110C3826C05008B830C00000085C07402FFC3
S31508048120D08B5DFCC9C389F690909090909090FE
[root@boota]#
```

The following command creates a binary image of the ftp file. The binary output is a memory image of the executable and all symbols are removed.

```
objcopy -O binary ftp ftp.bin
```

Here is how you display a type of the new binary file.

```
[root@boota]# file ftp.bin
ftp.bin: X11 SNF font data, LSB first
[root@boota]#
```

It is a good idea to strip object files before converting them to S-record or binary files. Start addresses can be set using the command line options `--set-start` and `--adjust-start`.

7.7.6 Using the objdump Utility

The `objdump` command displays information about an object file. It can also be used to disassemble an object file. You can use different command line options to display particular information about an object file.

7.7.6.1 Displaying Header Information

The following command displays the header information of the binary file `a.out`.

```
[root@boota]# objdump -f a.out
```

```
a.out:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048360
```

```
[root@boota]#
```

7.7.6.2 Displaying Section Headers

The following command displays information about all section headers in the `a.out` file.

```
[root@boota]# objdump -h a.out | more
```

```
a.out:      file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp        00000013  080480f4  080480f4  000000f4  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag  00000020  08048108  08048108  00000108  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash          00000034  08048128  08048128  00000128  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym        00000080  0804815c  0804815c  0000015c  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr        00000095  080481dc  080481dc  000001dc  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version   00000010  08048272  08048272  00000272  2**1
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000030  08048284  08048284  00000284  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.got       00000008  080482b4  080482b4  000002b4  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt       00000028  080482bc  080482bc  000002bc  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init          00000018  080482e4  080482e4  000002e4  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt           00000060  080482fc  080482fc  000002fc  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text          00000160  08048360  08048360  00000360  2**4
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini          0000001e  080484c0  080484c0  000004c0  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata        00000015  080484e0  080484e0  000004e0  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data          00000010  080494f8  080494f8  000004f8  2**2
    CONTENTS, ALLOC, LOAD, DATA
 15 .eh_frame      00000004  08049508  08049508  00000508  2**2
```

```

CONTENTS, ALLOC, LOAD, DATA
16 .ctors      00000008 0804950c 0804950c 0000050c 2**2
CONTENTS, ALLOC, LOAD, DATA
17 .dtors      00000008 08049514 08049514 00000514 2**2
CONTENTS, ALLOC, LOAD, DATA
18 .got        00000024 0804951c 0804951c 0000051c 2**2
CONTENTS, ALLOC, LOAD, DATA
19 .dynamic    000000a0 08049540 08049540 00000540 2**2
CONTENTS, ALLOC, LOAD, DATA
20 .sbss       00000000 080495e0 080495e0 000005e0 2**0
CONTENTS
21 .bss        00000018 080495e0 080495e0 000005e0 2**2
ALLOC
22 .stab       00000f9c 00000000 00000000 000005e0 2**2
CONTENTS, READONLY, DEBUGGING
23 .stabstr    00002ec6 00000000 00000000 0000157c 2**0
CONTENTS, READONLY, DEBUGGING
24 .comment    00000144 00000000 00000000 00004442 2**0
CONTENTS, READONLY
25 .note       00000078 00000000 00000000 00004586 2**0
CONTENTS, READONLY

```

```
[root@boota]#
```

7.7.6.3 Disassembling a File

Perhaps the major advantage of this utility is its ability to disassemble object files. Usually the disassembly code is quite long but still you can make sense of it. The following is a segment of disassembly code from the `a.out` file.

```
[root@boota]# objdump -d a.out | more
```

```
a.out:      file format elf32-i386
```

```
Disassembly of section .init:
```

```

080482e4 <_init>:
  80482e4:55          push   %ebp
  80482e5:89 e5      mov    %esp,%ebp
  80482e7:83 ec 08   sub   $0x8,%esp
  80482ea:e8 95 00 00 00   call  8048384 <call_gmon_start>
  80482ef:90        nop
  80482f0:e8 2b 01 00 00   call  8048420 <frame_dummy>
  80482f5:e8 86 01 00 00   call  8048480
<__do_global_ctors_aux>
  80482fa:c9        leave
  80482fb:c3        ret

```

```
Disassembly of section .plt:
```

```
080482fc <.plt>:
```



```

80482fc:ff 35 20 95 04 08    pushl  0x8049520
8048302:ff 25 24 95 04 08    jmp     *0x8049524
8048308:00 00                add     %al,(%eax)
804830a:00 00                add     %al,(%eax)
804830c:ff 25 28 95 04 08    jmp     *0x8049528
8048312:68 00 00 00 00      push   $0x0
8048317:e9 e0 ff ff ff      jmp     80482fc <_init+0x18>
804831c:ff 25 2c 95 04 08    jmp     *0x804952c
8048322:68 08 00 00 00      push   $0x8
8048327:e9 d0 ff ff ff      jmp     80482fc <_init+0x18>
804832c:ff 25 30 95 04 08    jmp     *0x8049530
8048332:68 10 00 00 00      push   $0x10
8048337:e9 c0 ff ff ff      jmp     80482fc <_init+0x18>
804833c:ff 25 34 95 04 08    jmp     *0x8049534
8048342:68 18 00 00 00      push   $0x18
8048347:e9 b0 ff ff ff      jmp     80482fc <_init+0x18>
804834c:ff 25 38 95 04 08    jmp     *0x8049538
8048352:68 20 00 00 00      push   $0x20
8048357:e9 a0 ff ff ff      jmp     80482fc <_init+0x18>
Disassembly of section .text:

```

```

08048360 <_start>:
8048360:31 ed                xor     %ebp,%ebp
8048362:5e                    pop     %esi
8048363:89 e1                mov     %esp,%ecx
8048365:83 e4 f0             and     $0xffffffff0,%esp
8048368:50                    push   %eax
8048369:54                    push   %esp
804836a:52                    push   %edx
804836b:68 c0 84 04 08      push   $0x80484c0
8048370:68 e4 82 04 08      push   $0x80482e4
8048375:51                    push   %ecx

```

7.7.6.4 Disassembling with Source Code

The command can also be used to disassemble an object file so that the source code is also displayed along with assembly output. This is done using the `-S` option. Following is output of disassembly for code that is used to display string the “Hello world”. You can see the assembly language instructions used for this purpose.

```

08048460 <main>:
#include <stdio.h>
main()
{
8048460:55                    push   %ebp
8048461:89 e5                mov     %esp,%ebp
8048463:83 ec 08             sub     $0x8,%esp
    printf ("Hello world\n");
8048466:83 ec 0c             sub     $0xc,%esp

```

```

8048469: 68 e8 84 04 08      push   $0x80484e8
804846e: e8 c9 fe ff ff      call  804833c <_init+0x58>
8048473: 83 c4 10             add    $0x10,%esp
}

```

7.7.6.5 Displaying Information about Library Files

The command can also be used to display information about library files. The following command displays information about the library file `libcommon.a` and shows the object file names from which this library is built.

```

[root@boota]# objdump -a libcommon.a
In archive ../../chap-04/make/libcommon.a:

common.o:      file format elf32-i386
rw-r--r-- 0/0 11668 Nov 13 19:48 2001 common.o

ftp.o:        file format elf32-i386
rw-r--r-- 0/0 11824 Nov 13 19:53 2001 ftp.o

[root@boota ltrace]#

```

Common options that are used with this command are listed in Table 7-5.

Table 7-5 Common options used with `objdump` command

Option	Description
-a	Display information about library files
--debugging	Show debugging information
-d	Disassemble an object file
-f	Show file headers summary
-S	Display source code with disassembly information
-t	Display symbol table entries

7.7.7 Using the size Utility

The `size` utility displays sizes of each section in an object file. The following command displays sizes for the object file `a.out`.

```

[root@boota]# size a.out
   text  data  bss   dec   hex    filename
   1015  232   24   1271   4f7    a.out
[root@boota]#

```

7.7.8 Using the strings Utility

The `strings` utility displays printable strings in an object file. By default it displays strings only in initialized and loaded sections of the object file. The object file may be a library file as well. The following command displays strings in the `a.out` file.

```
[root@boota]# strings a.out
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
printf
__cxa_finalize
__deregister_frame_info
_IO_stdin_used
__libc_start_main
__register_frame_info
GLIBC_2.1.3
GLIBC_2.0
PTRh
QVh`
Hello world
[root@boota]#
```

A complete list of all strings in a file can be displayed with the `-a` option on the command line. Multiple filenames or wild cards can be used on the command line. Using the `-f` option in this case will display each filename associated with a symbol.

7.7.9 Using the addr2line Utility

The `addr2line` utility maps an address in the object file to a line in the source code file. Consider the following output section of the “`objdump -S a.out`” command that shows addresses and source codes.

```
08048460 <main>:
#include <stdio.h>
main()
{
  8048460:55                push   %ebp
  8048461:89 e5                mov    %esp,%ebp
  8048463:83 ec 08            sub    $0x8,%esp
  printf ("Hello world\n");
  8048466:83 ec 0c            sub    $0xc,%esp
  8048469:68 e8 84 04 08      push  $0x80484e8
  804846e:e8 c9 fe ff ff      call  804833c <_init+0x58>
  8048473:83 c4 10            add    $0x10,%esp
}
```

The `a.out` file is generated from the following `a.c` file.

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

```
2 main()
3 {
4     printf ("Hello world\n");
5 }
```

Now you can display the line number of the source code file corresponding to address 8048466 in the object file with the following command:

```
[root@boota]# addr2line -e a.out 8048466
/root/chap-07/ltrace/a.c:4
[root@boota]#
```

This is the line where the `printf` function is called in `a.c` file. The `addr2line` command is useful in debuggers where you need to map addresses to a line in source code file.

7.8 Using the `ldd` Utility

The `ldd` utility is very useful in finding out the dependencies of an executable on shared libraries. This is necessary when you are copying executable files from one computer to another to make sure that required libraries are present on the destination computer also. The following command shows that `libc.so.6` must be present to execute `a.out` file on a computer.

```
[root@boota]# ldd a.out
libc.so.6 => /lib/i686/libc.so.6 (0x4002c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[root@boota]#
```

So if you copy `a.out` to another computer, make sure that this library is also present on the destination computer.

7.9 References and Resources

1. GNU web site at <http://www.gnu.org/>
2. The `cbrowser` home page at <http://cbrowser.sourceforge.net>
3. The `cscope` home page at <http://cscope.sourceforge.net>