

C Programming Tools

MOTIVATION

The most commonly used programming languages on UNIX systems are C and C++. This isn't surprising, since UNIX was written in C and many versions come with a standard C compiler. Most UNIX utilities and many commercial products are written in C or C++. It's therefore likely that you will find knowledge about writing, compiling, and running C programs very useful. Of course, UNIX supports many other popular programming languages, but this chapter applies primarily to the C language and its supporting tools, because these are so fundamental to UNIX environments.

C COMPILERS

Until recently, a C compiler was a standard component in UNIX, especially UNIX versions that came with source code. Unfortunately, some vendors have chosen to "unroll" the C compilers from their UNIX distributions and sell them separately. Depending on your needs, you should check on this in any version of UNIX you are considering using. However, even if your version of UNIX no longer ships a C compiler, you have an alternative, thanks to the GNU Project: GNU C (gcc) and GNU C++ (g++) are freely available C and C++ compilers, respectively, on most UNIX platforms at the GNU Compiler Collection web site, <http://www.gnu.org/software/gcc/gcc.html>

SINGLE-MODULE PROGRAMS

Let's examine a C program that performs a simple task: reversing a string. To begin with, I'll show you how to write, compile, link, and execute a program that solves the problem using a single source file. Then I'll explain why it's better to split the program up into several independent modules, and I'll show you how to do that. Here's a source code listing of the first version of the reverse program:

```

1 / *REVERSE.C */
2
3 #include <stdio.h>
4
5 /* Function Prototype */
6 void reverse () ;
7
8 /*****
9
10 void main ()
11 {
12     char str [100]; /* Buffer to hold reversed string */
13     reverse ("cat", str); /* Reverse the string "cat" */
14     printf ("reverse ("cat") = %s\n", str); /* Display result */
15     reverse ("noon", str); /* Reverse the string "noon" */
16     printf ("reverse ("noon") = %s\n", str); /* Display Result */
17 }
18
19 /*****
20
21 void reverse (before, after)
22 {
23     char *before; /* A pointer to the source string */
24     char *after; /* A pointer to the reversed string */
25
26     {
27         int i;
28         int j;
29         int len;
30
31         len = strlen (before);
32
33         for (j = len - 1; i = 0; j >= 0; j--; i++) /* Reverse loop */
34             after[i] = before[j];
35
36         after[len] = NULL; /* NULL terminate reversed string */
37     }
38 }

```

Compiling a C Program

To write and run the reverse program, I first created a subdirectory called “reverse” inside my home directory and then created the file “reverse.c”, using the UNIX **emacs** editor. I then compiled the C program with the **cc** utility. To prepare an executable version of a single, self-contained program, follow **cc** with the name of the source code file, which must end in a “.c” suffix. **cc** doesn’t produce any output when the compilation is successful. By default, **cc** creates an executable file called “a.out” in the current directory. To run the program, type “a.out”. Any errors that are encountered are sent to the standard error channel, which is connected by default to your terminal’s screen.

Here’s what happened when I compiled my program:

```

$ mkdir reverse          ...create subdirectory for source code.
$ cd reverse
$ ... I created the file reverse.c using emacs.
$ cc reverse.c           ...compile source.
"reverse.c", line 16: syntax error at or near variable name "cat"
"reverse.c", line 18: syntax error at or near variable name "noon"
"reverse.c", line 35: syntax error at or near symbol ;
"reverse.c", line 35: syntax error at or near symbol )
$

```

As you can see, cc found a number of compile-time errors, listed together with their causes as follows:

- The errors on lines 16 and 18 were due to an inappropriate use of double quotes within double quotes.
- The errors on line 35 were due to an invalid use of a semicolon (;).

Since these errors were easy to correct, I copied the error-laden "reverse.c" file to a file called "reverse.old.c" and then removed the compile-time errors via emacs. I left the original file in the directory so that I could see the evolution of my programming attempts.

If there are many compilation errors, it is a good strategy to debug the first few errors since some remaining compilation errors may not be actual errors. They exist due to the limitations of the compiler. Hence, fixing earlier errors will usually result in fewer errors encountered in the debugging process.

If there are so many compilation errors that it cannot fit into one screen. One way is to compile by the following command:

```
cc reverse.c | & more
```

which means that the compilation errors will be displayed screen by screen. That is, it will display the first screen of errors and wait. After the user examines the errors, he/she can choose to display the next screen of errors by hitting RETURN or quit by hitting Control-C. Then, the user can go back to modify the source code.

A Listing of the Corrected Reverse Program

Here is the second, corrected version of the reverse program, with the lines containing the errors that I corrected in italics:

```

1  /* REVERSE.C */
2
3  #include <stdio.h>
4
5  /* Function Prototype */
6  void reverse ();
7
8  /*****
9
10 main ()
11 {
12 {
13     char str [100]; /* Buffer to hold reversed string */
14
15     reverse ("cat", str); /* Reverse the string "cat" */
16     printf ("reverse (\\"cat\\") = %s\\n", str); /* Display */
17     reverse ("noon", str); /* Reverse the string "noon" */
18     printf ("reverse (\\"noon\\") = %s\\n", str); /* Display */
19 }
20
21 /*****
22
23 void reverse (before, after)
24 {
25     char *before; /* A pointer to the source string */
26     char *after; /* A pointer to the reversed string */
27
28 {
29     int i;
30     int j;
31     int len;
32
33     len = strlen (before);
34
35     for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
36         after[i] = before[j];
37
38     after[len] = NULL; /* NULL terminate reversed string */
39 }

```

Running a C Program

After compiling the second version of “reverse.c”, I ran it by typing the name of the executable file, “a.out”. As you can see, the answers were correct:

```

$ cc reverse.c          ...compile source.
$ ls -l reverse.c a.out ...list file information.
-rwxr-xr-x 1 glass      24576 Jan5 16:16  a.out*
-rw----r-- 1 glass      439   Jan 5 16:15  reverse.c
$ a.out                ... run program
reverse ("cat") = tac
reverse ("noon") = noon
$

```

Overriding the Default Executable Name

The name of the default executable file, “a.out”, is rather cryptic, and an “a.out” file produced by a subsequent compilation would overwrite the one that I just produced. To avoid both problems, it’s best to use the -o option with **cc**, which allows you to specify the name of the executable file that you wish to create:

```
$ cc reverse.c -o reverse          ...call the executable “reverse”.  
$ ls -l reverse  
-rwxr-xr-x  1 glass      24576 Jan 5 16:19  reverse*  
$ reverse                          ...run the executable “reverse”.  
reverse (“cat”) = tac  
reverse (“noon”) = noon  
$
```

THE UNIX DEBUGGER: **dbx**

The UNIX debugger, **dbx**, allows you to debug a program symbolically. Although it's not as good as most professional debuggers on the market, **dbx** comes as a handy standard utility in most versions of UNIX. It includes the following facilities:

- single stepping
- breakpoints
- editing from within the debugger
- accessing and modifying variables
- searching for functions
- tracing

*Utility: **dbx** `executableFilename`*

dbx is a standard UNIX debugger. The named executable file is loaded into the debugger and a user prompt is displayed. To obtain information on the various **dbx** commands, enter **help** at the prompt.

To demonstrate **dbx**, let's debug the following recursive version of palindrome ():

```

1  /* PALINDRO5.C */
2
3  #include "palindrome.h"
4  #include <string.h>
5
6
7  enum { FALSE, TRUE };
8
9
10 int palindrome (str)
11
12 char *str;
13
14 {
15     return (palinAux (str, 1, strlen (str)));
16 }
17
18 /*****
19
20 int palinAux (str, start, stop)
21
22 char *str;
23 int start;
24 int stop;
25
26 {
27     if (start >= stop)
28         return (TRUE);
29     else if (str[start] != str[stop])
30         return (FALSE);
31     else
32         return (palinAux (str, start + 1, stop - 1));
33 }

```

Preparing a Program for Debugging

To debug a program, it must have been compiled using the `-g` option to `cc`, which places debugging information into the object module.

Entering the Debugger

Once a program has been compiled correctly, you may invoke `dbx`, with the name of the executable file as the first argument. `dbx` presents you with a prompt. I recommend that you enter `help` at the prompt to see a list of all the `dbx` commands:

<code>\$ dbx main2</code>	<i>....enter the debugger.</i>
<code>dbx version srlO.3(4) of 7/6/90 l752</code>	
<code>reading symbolic information ...</code>	
<code>Type 'help' for help.</code>	
<code>(dbx) help</code>	<i>.. . obtain help.</i>
<code>run [args]</code>	<i>- begin execution of the program</i>
<code>stop at <line></code>	<i>- suspend execution at the line</i>
<code>stop in <func></code>	<i>- suspend execution when <func> is called</i>
<code>stop if <cond></code>	<i>- suspend execution when <cond> is true</i>
<code>trace <line#></code>	<i>- trace execution of the line</i>
<code>trace <func></code>	<i>- trace calls to the function</i>
<code>trace <var></code>	<i>- trace changes to the variable</i>
<code>trace <exp> at <line#></code>	<i>- print <exp> when <line> is reached</i>
<code>status</code>	<i>- print numbered list of traces and stops in effect</i>
<code>delete <#> [<#> ...]</code>	<i>- cancel trace or stop of each number given</i>
<code>cont</code>	<i>- continue execution from where it stopped</i>
<code>step</code>	<i>- execute one source line, stepping into functions</i>
<code>next</code>	<i>- execute one source line, skipping over calls</i>
<code>return</code>	<i>- continue until the current function returns</i>
<code>call <func>(<params>)</code>	<i>- execute the given function call</i>
<code>print <exp> [, <exp> ...]</code>	<i>- print the values of the expressions</i>
<code>where</code>	<i>- print currently active procedures</i>
<code>whatis <name></code>	<i>- print the declaration of the name</i>
<code>assign <var> = <exp></code>	<i>- assign the program variable the value of <exp></i>
<code>dump <func></code>	<i>- print all variables in the active function</i>
<code>list [<line#>], <line#>></code>	<i>- list source lines</i>
<code>use <directory-list></code>	<i>- set the search path for source files</i>
<code>sh <command-line></code>	<i>- pass the command line to the shell</i>
<code>quit</code>	<i>- exit dbx</i>
<code>(dbx) _</code>	

Running a Program

To run your program, enter the run command, which runs the program to completion:

<code>(dbx) run</code>	<i>... run the program.</i>
<code>palindrome ("cat") = 0</code>	
<code>palindrome ("noon")= 0</code>	
<code>program exited</code>	
<code>(dbx) _</code>	

Oops! The string "noon" is a palindrome, but my function thinks that it isn't. Time to I delve into **dbx**.

Tracing a Program

To obtain a line-by-line **trace step**, use the trace command. When any kind of trace is requested, **dbx** returns an index number that can be used by the **delete** command to

```

(dbx) trace step                               ... request a trace.
[1] trace                                     ... request is #1.
(dbx) rerun                                   ... run the program from the start
trace:      9      printf ("palindrome (\\"cat\\") = %d\n", palindrome
("cat"));
trace:     10      int palindrome (str)
trace:     15          return (palinAux (str, 1, strlen (str)));
trace:     20      int palinAux (str, start, stop)
trace:     27          if (start >= stop)
trace:     29              else if (str[start] != str[stop])
trace:     30              return (FALSE);
trace:     33      }
trace:     33      }
trace:     16      }
palindrome ("cat") = 0
trace:     10      printf ("palindrome (\\"noon\\") = %d\n", palindrome
("noon"));
trace:     10      int palindrome (str)
trace:     15          return (palinAux (str, 1, strlen (str)));
trace:     20      int palinAux (str, start, stop)
trace:     27          if (start >= stop)
trace:     29              else if (str[start] != str[stop])
trace:     30              return (FALSE);
trace:     33      }
trace:     33      }
trace:     16      }
palindrome ("noon") = 0
trace:     11      }
trace:     11      }
program exited
(dbx) _

```

Tracing Variables and Function Calls

A trace may be placed on a variable's value or a call to a particular function by adding parameters to the trace command. The syntax for trace is shown below. Here's the output from dbx after three new traces were added and then the program was restarted:

<i>(dbx) trace change palinAux`start</i>	<i>...trace the variable called “start”.</i>
<i>[2] trace start in palinAux</i>	
<i>(dbx) trace change palinAux`stop</i>	<i>... trace the variable called “stop”.</i>
<i>[3] trace stop in palinAux</i>	
<i>(dbx) trace in palinAux</i>	<i>... trace the function “palinAux”.</i>
<i>[4] trace palinAux</i>	
<i>(dbx) rerun</i>	<i>... run the program from the start.</i>

```

trace:          9      printf ("palindrome (\`cat\`) = %d\n", palindrome
("cat"));
trace:          10      int palindrome (str)
trace:          15          return (palinAux (str, 1, strlen (str)));
trace:          20      int palinAux (str, start, stop)
calling palinAux(str = "cat", start = 1, stop = 3) from function
palindrome.palindrome
trace:          27          if (start >= stop)
initially (at line 27 in "/home/glass/reverse/palindrome.c"): start = 1
initially (at line 27 in "/home/glass/reverse/palindrome.c"): stop = 3
trace:          29          else if (str[start] = str[stop])
trace:          30          return (FALSE);
trace:          33      }
trace:          33      }
trace:          16      }
palindrome ("cat") = 0
trace:          10      printf ("palindrome (\`noon\`) = %d\n", palindrome
("noon"));
trace:          10      int palindrome (str)
trace:          15          return (palinAux (str, 1, strlen (str)));
trace:          20      int palinAux (str, start, stop)
after line 20 in "/home/glass/reverse/palindrome.c": stop 4
calling palinAux(str = "noon", start = 1, stop 4) from function
palindrome .palindrome
trace:          27          if (start ,= stop)
trace:          29          else if (str[start] !=str[stop])
trace:          30          return (FALSE);
trace:          33      }
trace:          33      }
trace:          16      }
palindrome ("noon ") = 0
trace:          11      }
trace:          11      }
program exited
(dbx) _

```

trace change `function`variable

and to trace a call to a named function, use the syntax:

trace infunction

The Bug

By now, the nature of the bug is fairly clear: The values of *start* and *stop* are incorrect, each being one greater than it should be. It's a very common error to forget that C array indices begin at zero rather than one. You may call up the editor specified by the \$EDITOR environment variable by using the edit command. This is handy for correcting errors on the fly, although you must remember to recompile the program

before debugging it again. Here is the correct version of the `palindrome()` function:

```
int palindrome (str)
char *str;
{
    return (palinAux (str, 0, strlen (str) - 1));
}
```

I'll end this section with a brief discussion of some useful miscellaneous **dbx** commands for setting breakpoints, single stepping, accessing variables, and listing portions of a program.

Breakpoints

To make **dbx** stop when it encounters a particular function, use the `stop` command. This allows you to run a program at full speed until the function that you wish to examine more closely is executed. The following **dbx** commands are illustrative:

```
(dbx) stop in palinAux           ... set breakpoint.
[7] stop in palinAux
(dbx) rerun                     ...run the program from the start.
trace:    9    printf ("palindrome (^cat^") = %d\n", palindrome ("cat"));
trace:   10    int palindrome (str)
trace:   15    return (palinAux (str, 1, strien (str)));
trace:   20    int palinAux (str, start, stop)
calling palinAux(str = "cat", start =1, stop =3) from function palindrome.palindrome
[7] stopped in palinAux at line 27 in file "/home/glass/reverse/palindrome. C"
 27  if (start >= stop)
(dbx)_
```

Single Stepping

To step through a program one line at a time, use the `step` command. This command causes **dbx** to redisplay its prompt immediately after each line of the program has been executed. The command is useful for high-resolution interrogation of a function. In the following example, I entered **step** after my program stopped at line 27:

```
(dbx) step                     ... execute line after *27 and then stop.
trace:   29    else if (str[start] !=str[stop])
initially (at line 29 in "/home/glass/reverse/palindrome.c"):
start =1
initially (at line 29 in "/home/glass/reverse/palindrome.c"):
stop = 3
stopped in palinAux at line 29 in file "/home/glass/reverse/palindrome.c"
 29    else if (str[start] != str[stop])
(dbx)_
```

Accessing Variables

To print the value of a particular variable at any time, use the **print** command. The **whatis** command displays a variable's declaration, and the **which** command tells you where the variable is declared. The **where** command displays a complete stack trace, and the **whereis** command tells you where a particular function is located. All of these commands appear in the following example:

```
(dbx) print start          ... display current value of start.
1
(dbx) whatis start         ... get type information.
int start;
(dbx) which start          ... find its location.
palindrome.palinAux.start
(dbx) where                ... obtain stack trace.
palinAux(str = "cat", start = 1, stop = 3), line 29 in
"/home/glass /reverse/palindrome. c"
palindrome.palindrome(str = "cat"), line 15 in
"/home/glass/reverse/palindrome. c"
main(), line 9 in "/home/glass/reverse/main2.c"
unix_$main () at 0x3b4e7a14
_start(), line 137 in  "//garcon/unix_src/lang/sgs/src/crt0/crt0 .c"
(dbx) whereis palinAux     ... locate a function.
palindrome.palinAux
(dbx) whereis start        ... locate a variable.
palindrome.palinAux.start
(dbx)_
```

Listing a Program

The list command allows you to list the first few lines of a function, and the *l* and *?* commands allow you to search forwards and backwards through text, respectively, as shown in the following example:

```
(dbx) list palindrome                               ...list ten lines.
5
6
7  enum { FALSE, TRUE };
8
9
10 int  palindrome (str)
11
12 char* str;
13
14 }
15  return (palinAux (str, l, strlen (str)));
16 {
17
18  / *****/
19
20  int palinAux (str, start, stop)
21
(dbx) ?palinAux          ... search backward for string "palinAux".
20  int palinAux (str,start,stop)
(dbx) /palinAux          ...search forward for string "palinAux".
32  return (palinAux (str, start + l, stop - l));
(dbx) _
```

Leaving the Debugger

To quit dbx, use the quit command:

```
(dbx) quit      ... leave the debugger.
$
```

You will now be back in your shell.

Summary

In this section, I've presented a smattering of the commonly used **dbx** commands. Utilized wisely, they can provide useful hints about the errors in your program. In my own opinion, **dbx** is actually a pretty poor debugger compared with some of the popular PC debuggers, such as Borland's Turbo Debugger, and advanced UNIX system debuggers, such as ObjectWorks/C++. On the positive side, at least **dbx** is available on most UNIX machines, which makes a rudimentary understanding of its operation handy.

WHEN YOU'RE DONE: strip

The debugger and profile utilities both require you to compile a program using special options, each of which adds code to the executable file. To remove this extra code after debugging and profiling are finished, use the strip utility. Here's an example of how much space you can save:

<i>\$ ls -l main2</i>		<i>...look at original file.</i>
<i>-rwxr-xr-x</i>	<i>1 gglass</i>	<i>5904 Jan 8 22:18 main2*</i>
<i>\$ strip main2</i>		<i>... strip out spurious information.</i>
<i>\$ ls -l main2</i>		<i>...look at stripped version.</i>
<i>-rwxr-xr-x</i>	<i>1 gglass</i>	<i>3373 Jan 8 23:17 main2*</i>
<i>\$</i>		

Synopsis: strip {fileName}+

strip removes all of the symbol table, relocation, debugging, and profiling information from the named files.