

Basic Debugging

Do you know *exactly* what your trying to do? Before you can make your code work correctly you must know what correct is. Is your code well laid out? Is it commented? Remember that well written code has no bugs... If you always write your programs like this you will have fewer bugs to start with and bugs you don't have are easiest to solve.

Asking the Compiler For Help

You should always give the `-Wall` argument to the compiler when compiling your program...i.e.

```
gcc -Wall test.c
```

This turns on the compiler warnings. A warning is something that the compiler thinks is wrong but its not absolutely sure. The compiler is often right though. Make sure you fix (or understand why the warning is wrong) every warning you are given. The messages may seem a bit confusing at first but if you read them carefully you can work out what they mean. Warnings a wonderful because they tell you exactly what's wrong...much easier to fix a compiler warning than it is to debug a program that just occasionally does the wrong thing.

There are more warning flags that you can (and probably should use)...I tend to use:

```
-W -Wall -Wbad-function-cast -Wcast-align -Wcast-qual -Wchar-subscripts  
-Wconversion -Wmissing-prototypes -Wnested-externs -Wpointer-arith  
-Wredundant-decls -Wshadow -Wstrict-prototypes -Wundef -Wwrite-strings
```

Obviously you don't want to have to type all that it every time but if you are using `make` to compile your program (and you should be) then you can add these flags to the `CFLAGS` variable. An example Makefile is given in figure 1 that you can use for your programs (you don't have to worry about everything the Makefile does yet).

Check the `gcc` documentation (see below for a reference) for still more warning flags.

Watching Your Program From the Inside

If you have fixed all the warnings but your program still doesn't work as you would expect then you can use a debugger. A debugger allows you to "look inside" your program as it runs. You can examine the value of variables and see the flow of control in your code. The debugger used in this course is `gdb` - the GNU debugger. To use the debugger you must pass the `-g` flag to the compiler when compiling your code...i.e.

```
gcc -Wall -g test.c
```

Starting the Debugger

```
gdb a.out  
b main  
run
```

This runs the debugger on the executable `a.out`, sets a breakpoint at `main` and then starts the program executing. If you know which function is going wrong you can specify that function, rather than `main`.

```

# Modify these variables as appropriate
PROG=      editor
SRCS=      ${PROG}.c utils.c
LDADD=     -lm

# You probably won't need to modify anything below here
CC=        gcc
CFLAGS=    ${WARNS}
CLEANFILES= ${OBJS} ${PROG} a.out *.core core
WARNS=     -W -Wall -Wbad-function-cast -Wcast-align -Wcast-qual \
           -Wchar-subscripts -Wconversion -Wmissing-prototypes \
           -Wnested-externs -Wpointer-arith -Wredundant-decls -Wshadow \
           -Wstrict-prototypes -Wundef -Wwrite-strings

# Beware! Here is magic! Do not descend beyond this line unless you know what
# you are doing. Much of this is Solaris specific - certainly it's not the
# BSD way.
OBJS=      ${SRCS:%.c=%.o}

${PROG}:   ${OBJS}
           ${CC} ${CFLAGS} -o $@ ${OBJS} ${LDADD}

clean:
           -${RM} ${CLEANFILES}

```

Figure 1: Example Makefile

Debugger (GDB) Commands

Figure 2 is a list of some of the more common gdb commands. There are actually lots more - see the documentation for those. You will only understand them (and remember them) if you practice them so make sure you try the tutorial exercises.

`gdb` allows you to abbreviate commands (that is you only have to type `p` for `print`, `b` for `break` etc.).

Core Dumps

If your program core dumps you can use the debugger to see what went wrong. Use the command:

```
gdb a.out a.out.core
```

where `a.out` is a copy of your program compiled with `-g` and `a.out.core` is the core file. You can then use `bt`, `up` and `print` to see happened.

break	<p>set a breakpoint</p> <p>You can say:</p> <ul style="list-style-type: none"> • <code>b 23</code> - set a break point at line 23 • <code>b foo</code> - set a break point at the function <code>foo</code> • <code>b 23 if i == 17</code> - set a break point at line 23. Only breaks if the variable <code>i</code> equals 17 when you reach the break point
info break	lists all the break points you have set
delete	<p>remove a breakpoint</p> <p><code>d 1</code> - removes breakpoint 1</p>
run	<p>set the program running</p> <p>The program will run as normal until it hits a breakpoint or your program finishes (or crashes).</p>
step	<p>steps into the code</p> <p>Execute the next statement. If that statement is a function call then stop just inside the function.</p>
next	<p>steps over the code</p> <p>Execute the next statement. If that statement is a function call then run it.</p>
cont	<p>continue running the program</p> <p>If your program has stopped at a break point this will set it running again. It will run as normal until it hits a breakpoint or your program finishes (or crashes).</p>
print	<p>print the value of a variable</p> <ul style="list-style-type: none"> • <code>p i</code> - prints the value of the variable <code>i</code> • <code>p (i + 1)</code> - prints the value of the variable <code>i + 1</code> (<code>i</code> stays the same) • <code>p *a</code> - prints the value stored in the memory who's address is stored in <code>a</code>
set variable	<p>set the value of a variable</p> <ul style="list-style-type: none"> • <code>set variable i = 3</code> - sets the variable <code>i</code> to the value 3
list	<p>lists your source code</p> <p><code>l 23</code> - list code around line 23 <code>l io.c:23</code> - list code around line 23 of the file <code>io.c</code> <code>l foo</code> - list code around the beginning of function <code>foo</code> <code>l io.c:foo</code> - list code around the beginning of function <code>foo</code> in the file <code>io.c</code></p>
bt	<p>print a back trace</p> <p>shows which functions were called to get to the current position. If you say <code>bt full</code> you also get the value of each of the called function's variables.</p>
up	move up the function stack (as printed by <code>bt</code>)
help	display the online help
quit	quit the debugger (and kill your program if it is still running)
Figure 2: GDB Commands	

Conclusion

There is much more you can do with the debugger that isn't covered here. There are also graphical interfaces to the debugger. If your interested read the man page (`man gdb`) and the info page (`info gdb`). There is lots of info on the web - just do a search for `gdb`.

A complete set of man and info pages are available on the web if you prefer reading them that way. Have a look at <http://www.au.FreeBSD.org/docs.html>.

A good manual for `gdb` can be found at http://sourceware.cygnum.com/gdb/onlinedocs/gdb_toc.html.