Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1999 Handout 2– Tools of the Trade February 12

---

1 Introduction

This document hopes to acquaint you with the development tools you will use in the 6.033 lab. After following the directions in this document, you should be sufficiently well versed in the GNU build and debugging environments to handle the later assignments.

This document covers five tools: the manual pages, the compiler, the debugger, MAKE, and CVS. All these tools can be found in the `gnu` locker on Athena. Documentation for the tools is available off of the lab home page (`http://web.mit.edu/6.033/www/lab-tools.html`).

2 Manual pages

Most UNIX variants have extensive on-line documentation. The documentation encompasses user and system programs, library procedure calls, system calls, and even general topics like network protocols.

The `man` program is used to browse the on-line documentation. It takes at least one argument: the name of the topic. Usually, this is the name of a command or procedure. For example, to learn more about the `read` system call:

```
% man read
READ(2)                    OpenBSD Programmer's Manual                    READ(2)

NAME
     read, readv - read input

...
```

The manual pages are divided into multiple sections. The section number of the current topic appears in parentheses after the topic name. In the above example, the `read` command is located in section 2.

On most UNIX systems, Section 1 is devoted to shell and user commands, section 2 is dedicated to the UNIX system calls, and section 3 describes the functions of the Standard C library.

Often a page will appear in multiple sections and it is necessary to specify the section number on the command line to access the correct page. For example, since there are at least two `printf` manual pages on OpenBSD, I must type:

```
% man 3 printf
```

to get the manual page for the C `printf` function. Under other UNIX systems, the `-s` option must be used:

```
% man -s 3 printf
```

The Sun machines on Athena require the `-s` option.

The `-k` option is especially useful – it does a keyword search across the titles of all of the man pages. For example:

```
% man -k tree
btree (3) - btree database access method
mtree (8) - map a directory hierarchy
tsearch, tfind, tdelete, twalk (3) - manipulate binary search trees
XQueryTree (3X11) - query window tree information
lndir (1) - create a shadow directory of symbolic links to another directory tree
```

3 Compiler

To compile C programs into programs executable from the command line, we will use gcc, the GNU C compiler.

A C program is made up of one or more procedures and their associated data. The text of a C program can be split over multiple files called source files. For gcc to recognize the source files as C source files, the files must end with `.c`.

3.1 Basic GCC

To compile the `hello` program into an executable file, simply type:

```
% gcc hello.c
```

If there are no errors, this will generate a file named `a.out`. The `a.out` file can be invoked directly from the shell.

The `-o` flag specifies the name for the output file. Thus,

```
% gcc -o hello hello.c
```

will create a file named hello. Like the `a.out` file, this program can be directly launched from the shell.

To add debugging information to the final result, the `-g` flag is added. The level of optimization can be controlled using the `-O` flag. To generate an optimized program with debugging information, the following command line can be used:

```
% gcc -g -O3 -o hello hello.c
```

If a program consists of multiple source files, they are listed on the command line.

```
% gcc -o hello hello.c greeting.c world.c
```

Often, we will want to link in a library of already compiled procedures to use with our program. In fact, gcc already links in the standard C library by default. The command above is actually:

```
% gcc -o hello hello.c greeting.c world.c /usr/lib/libc.a
```

We can add other libraries we are interested in. This version links in the procedures from the math library:

```
% gcc -o hello hello.c greeting.c world.c /usr/lib/libm.a /usr/lib/libc.a
```

Which can be abbreviated:

```
% gcc -o hello hello.c greeting.c world.c -lm
```

Note that libc is included by default. The `-l` option is followed immediately by a library name, for example, `-lfoo`. Given the `-lfoo` option, GCC will search the directories in its internal library path for a file named `libfoo.a` and try to link it with the program.

The library path can be augmented using the `-L` option. For example:

```
% gcc -L/usr/local/lib -o hello hello.c greeting.c world.c -lyo
```

will cause gcc to search `/usr/local/lib` for the library `libyo.a` in addition to the default directories.

Similarly, the path along which GCC looks for C header files can be augmented using the `-I` option.

```
% gcc -I/usr/local/include -o hello hello.c greeting.c world.c -lyo
```

The `-W` option gives you varying degrees of warning messages on the quality of your C code. The `-Wall` option is highly recommended as it finds many flaws in the code.

Finally, the `-D` option declares some preprocessor constants. Thus,

```
% gcc -Wall -DVERSION=3 -DDEBUG -o hello hello.c greeting.c world.c
```

is the same as adding the lines:

```
#define DEBUG
#define VERSION 3
```

to the top of every source file.

3.2 Optimizing the compilation process

The method of invoking GCC described in the previous section rolls together the four stages involved in generating the file executable:

1. preprocess the C source files (`.c`) into preprocessed C (`.i`)
2. compiling the `.i` files to assembler files (`.s`)
3. compiling the assembler files (`.s`) to object files (`.o`)
4. link the object files (`.o`) with the standard C library (`crt0.o, /usr/lib/libc.a`) and other libraries (`.a`) to generate the final executable (`a.out`)

In programs involving large amounts of code spread over multiple source files, it can be time-consuming to undergo all four stages on all of the files every time a change is made. Oftentimes, this is unnecessary because only a single source file changed. Instead, gcc allows us to stop the compilation at intermediate stages and store the results.

The `-c` option stops the compiler right before the link stage. Here's how to compile a program using the `-c` option:

```
% gcc -Wall -g -O3 -c hello.c
% gcc -Wall -g -O3 -c greeting.c
% gcc -Wall -g -O3 -c world.c
% gcc -Wall -g -o hello hello.o greeting.o world.o -lyo
```

The `.o` extension of the files in the last command tells GCC to ignore these files until it reaches the link stage. Since no files with a `.c`, `.s`, or `.i` extension are specified, no work is done at stages 1 through 3.

Though the above approach requires more commands, it allows us to regenerate the hello program quickly in the case where only one source file changes:

```
% gcc -Wall -g -O3 -c greeting.c
% gcc -Wall -g -o hello hello.o greeting.o world.o -lyo
```

3.3 Libraries

You can bundle a group of object files into a library file. The `ar` command is used to do this. For example,

```
% ar cq mylibc.a myprintf.o myscanf.o
```

creates an archive file named `mylibc.a` containing files `myprintf.o` and `myscanf.o`.

On some architectures, the `ranlib` program is run on the resultant archives to generate a symbol table for faster linking.

```
% ranlib mylibc.a
```

4 MAKE

Tracking which files have changed and need to be recompiled can be tricky if done by hand. The job becomes significantly more difficult when header files are modified since they can be included in multiple files or even other header files. To simplify book-keeping and automatic recompilation, the MAKE language was introduced.

An example file in the MAKE language:

```
# This is a comment in a Makefile

hello: hello.o greeting.o world.o
	gcc -Wall -g -o hello hello.o greeting.o world.o

hello.o: hello.c hello.h
	gcc -Wall -g -O3 -c hello.c

greeting.o: greeting.c hello.h
	gcc -Wall -g -O3 -c greeting.c

world.o: world.c hello.h
	gcc -Wall -g -O3 -c world.c

clean:
	-rm -f *.o hello
```

If the file is named makefile, the hello program can be rebuilt by typing `make`.

For the most part, a make file consists of one or more rules of the form:

```
TARGET ...:  DEPENDENCY1 DEPENDENCY2 ...
	commands
```

The target and dependencies are the names of files. The commands usually update or create the target, but that is not always necessary.

The whitespace used is very important to `make`. The TARGET must start at the beginning of the line. All the lines containing commands must be indented using a tab character. Repeat: all the commands must be indented using a tab character.

The target to rebuild is specified at the command line to `make`. For example, "make clean" will run the clean rule. As long as there is no file named clean in the directory, the rule will always execute its command. Similarly, `hello.o` can be rebuilt by typing "make hello.o".

If no target is specified on the command line, the target of the first rule is used.

Before a rule is run, the `make` program tries to recursively rebuild the dependencies. If a dependency is the target of another rule, the `make` program tries to satisfy that rule first. Some pseudo-code summarizing the process appears below.

```
bool rebuild_rule(r)

execute_rule = !file_exists(r.target);

for each dependency d in r
        if (find rule with target of d)
                rebuild_rule(rule)

        if (!file_exists(d))
                error(``Could not create dependency d'')

        if d.date > r.target.date then
                execute_rule = TRUE

if (execute_rule)
        run r.commands
```

The make format supports variable substitution as well as implicit rules. For example, `make` knows how to build object files (`.o`) from C source files. Thus, the following Makefile is equivalent to the one at the beginning of the section.

```
CC = gcc
CFLAGS = -Wall -g -O3
OBJECTS = hello.o greeting.o world.o

hello: $(OBJECTS)
        $(CC) $(CFLAGS) -o hello $(OBJECTS)

### Dependencies:
hello.o: hello.c hello.h
greeting.o: greeting.c hello.h
world.o: world.c hello.h
```

More information on the GNU MAKE program can be found in the on-line documentation.

5 Debugger

The GNU Debugger (GDB) is a useful tool for quickly tracking down a variety of defects. To take full advantage of the gdb, programs should be compiled and linked with the `-g` flag. This flag adds additional debugging information to the final executable files.

GDB is an interactive debugger with its own command line interface. The interface allows you to control every aspect of GDB's operation. For example, to start a session the `hello` executable, type `file hello` at the GDB command line. To actually run the executable, type `run` *args* where *args* are any arguments to pass to the executable. On-line help can be browsed using the `help` command.

It is even more convenient to run GDB from `gdb-mode` inside of Emacs. To invoke gdb in this fashion, type `M-x gdb`. One of the big advantages of this approach is that source listings of the program being debugged appear in an adjacent Emacs buffer. It is highly recommended that you use gdb in this fashion rather than straight from the command line.

5.1 Simple debugging with GDB

You can use GDB to find where a program crashes or hangs. The easiest way to do this is to run the program under GDB and wait until it crashes or hangs. You should be returned to the GDB command prompt at that time (in the case of a hang you have to type Ctrl-C). Typing `where` will usually give a backtrace of the program's call stack. The backtrace will list the chain of procedure calls which led to the offending instruction.

Each procedure call, its arguments, and its associated local variables constitute a stack frame. In the backtrace, the stack frames are numbered for convenient navigation. Use the `up`, `down`, and `frame` commands to move between frames.

The value of variables and expressions (even those involving procedure calls) can be discovered using the `print` command. The print command accepts virtually any C-style expression and understands casting. Local variables in the scope of a given procedure can only be accessed when GDB is in the procedure's stack frame.

5.2 Debugging harder problems

Sometimes, by the time the program crashes, the data structures and pointers available are so corrupted as to be totally useless for discovering the error. The best way to flag this behavior early in your programs is to write code which runs sanity checks on the program's data structures and reports errors early. This will save you much debugging time as compared to the alternatives.

It can be very useful, in the course, of debugging, to stop the program at a point in its execution to examine the internal state. The breakpoint and display commands provide this feature.

A breakpoint is an instruction inserted into a program's code. Each time the instruction is executed, control returns to the debugger. Usually, GDB returns to its command line for user input. In fact, GDB is highly configurable and can be made to do all sorts of complicated things at breakpoints, but we will not talk about that here.

The `break` command inserts a breakpoint in the code. The break instruction allows you to name locations in programs in several different ways. You can specify a function name (e.g., `main`) or a source file and line number (`printf.c:34`) or any expression that evaluates to an address (e.g., `0xa040000`). The first two approaches are generally the most usable.

The `clear` command clears a breakpoint. It takes as its arguments the same locations specified by `break`.

The `break` command returns the index of the breakpoint. This index can be used to disable or delete the breakpoint, using (not surprisingly) the `disable` and `delete` commands. See `help breakpoints` for more breakpoint-related commands.

The `step`, `next`, and `continue` commands continue execution of the program. The `continue` command continues execution until the next breakpoint is hit. The `step` and `next` command just execute the next source line. The step command goes into subroutine calls, whereas the next command treats a subroutine call as one line.

Finally, the display command can be used to specify expressions to be printed each time execution stops. The display command, like print, accepts almost any valid C expression.

This quick summary only scratches the surface of the features available in GDB. More information can be found on the summary card attached to this handout and in the GDB documentation available on the web page.

6 CVS

A CVS repository is a directory hierarchy rooted at the CVS root. Unlike a standard file system hierarchy, the CVS repository keeps a history of changes which have been made against it. Thus, users can retrieve older versions of individual files or even older versions of the entire repository. In addition, the CVS repository merges concurrent changes submitted by multiple users, allowing multiple users to safely edit the same set of files.

The CVSROOT environmental variable specifies the location of the CVS repository. You can set this variable (under the C shell) by using the setenv command. For example, setenv CVSROOT /mit/aphacker/cvs will cause CVS to use the repository located in /mit/aphacker/cvs.

6.1 Creating the repository

The functions for manipulating the repository are located in the cvs program. To create a new CVS repository:

```
% cd ~/mycvsroot
% setenv CVSROOT ~/mycvsroot
% mkdir mycvsroot
% cd mycvsroot
% cvs init
```

This creates a CVSROOT directory in the current directory and initializes the repository. You can then import your already existing sources into the repository. For example, if you already had a project1 subdirectory, the following command will create a myproject tree in the repository and place the contents of the directories in there:

```
% cd ~/project1
% cvs import -m ``Initial Import'' myproject vendortag start
```

The vendortag is not especially important. You can probably use your user name.

`cvs add file1 ...`	Add file(s) to the repository
`cvs remove file1 ...`	Remove a file
`cvs commit`	Commit local changes and added files to the repository
`cvs update`	Receive changes from the repository

Table 1: The basic CVS commands

6.2 Getting a local copy

All editing in CVS is done on a local copy of the files, not on the central repository. You should never need to edit the files in the repository.

To get a local copy of a directory tree from the repository, type "cvs checkout". For example,

```
% mkdir ~/freshdir
% cd ~/freshdir
% cvs checkout myproject
```

The checkout command should never be issued inside of the repository hierarchy.

The primary functions for manipulating the repository are shown in Table 1. Note that the `remove` command does not actually remove the file permanently from the repository. The file can still be accessed if an older version of the repository is checked out.

More information on CVS can be found on the manual page.

7 Assignment 0

The code for this assignment is located in `/mit/6.033/lab/zero`. For this assignment, you should:

1. Create a `Makefile` that compiles and links the two files.
2. Create a CVS repository and import the files into the repository. Check the files out of the repository, update them, and put the back in.
3. There are a few bugs in both source files. Find and eradicate them.

There is nothing to hand in for this assignment.

8 Recommended Books

Stevens, W. Richard. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

Stevens, W. Richard. *UNIX Network Programming.* Addison-Wesley, 1990.