C Programming Tools

2012

1 Introduction

This handout contains descriptions of several tools which you will find useful throughout CS033 and the rest of your C-programming experience.

2 man

The shell command `man` allows you to read the *manual pages* for programs, C functions and libraries, and a variety of other utilities. Manual pages are exactly what they sound like — in-depth, highly technical descriptions of a program or function, detailing proper usage and functionality. You will likely find them invaluable, if you haven't begun using them already. To access the manual page for a given program — say, `echo` — you would type

```
man echo
```

on the command line. Once you've accessed the manual page for a utility, you can scroll up, down, left, and right with the arrow keys, and exit out of the display page by pressing *q*.

2.1 Manual Sections

The manual pages are divided into numbered sections, which can be accessed specifically by including the corresponding number between the program name `man` and the argument. For example, to access the manual page for the C syscall `read()`, which is located in section 2 of the manual pages, you would type

```
man 2 read
```

Sometimes there are multiple programs or functions with the same name that reside in different sections; you'll need to use their numbers to access their manual pages. The sections are as follows (as explained in the manual page for the `man` utility (`man man`)):

- 1 - Executable programs and shell commands
- 2 - C system calls (you'll see what these are later)
- 3 - C library functions
- 4 - Special files
- 5 - File formats and conventions

- 6 - Games
- 7 - Miscellaneous
- 8 - System administration commands
- 9 - Kernel routines

Of these sections, you'll be using the first three most frequently. You don't have to include the section name most of the time when using `man`; however, if at first you don't find what you're looking for, check the appropriate section directly. For instance, if you type

```
man printf
```

you will get the manual page for a `bash` command. To access the manual page for the C library function `printf()`, you'll need to include the section number (3):

```
man 3 printf
```

3 gcc

The command `gcc` is used to compile your C programs from the command line. The basic syntax is

```
gcc file1.c file2.c ... filen.c
```

This will create an executable binary called *a.out*, which can be run with the command `./a.out`.

There are a number of options that can be applied to `gcc` to allow you more control. Those below are the most relevant for this class.

3.1 -o

The `-o` flag allows you to specify the name of the created file. As mentioned above, the default filename for binaries created by `gcc` is *a.out*. The following command will instead output a binary named *helloWorld*:

```
gcc file1.c file2.c ... filen.c -o helloWorld
```

3.2 -I

The `-I` flag is used to include a directory to be searched for header files. This can be used to override a system header file by providing your own version. For example,

```
gcc foo.c -I barDir -o foo
```

would look for header files in *barDir*.

3.3 `-l, -L`

The `-l` flag is used to tell `gcc`'s linker, which turns assembled *.o* files into an executable binary, to include functions in the specified library (an archive consisting of object files). So `-l foo` would search several directories for the archive file *libfoo.a* and allow its functions to be used by the linker for any object files occuring after the `-l` flag.

The `-L` flag adds a directory to the front of the list of directories searched by `-l`.

You can have as many -L and -l flags as you desire.

Some libraries have their own linking flag. A commonly-used example is the library *math.h*, which must be linked with the `gcc` flag `-lm`.

3.4 `-S`

Adding the `-S` flag to the `gcc` command will create an x86 assembly file instead of an executable binary. This allows you to see exactly how your C file is being converted to assembly and to understand any optimizations the compiler may be making.

For example, running the command

```
gcc file1.c file2.c ... filen.c -S
```

will create the assembly files `file1.s, file2.s, ..., filen.s`.

Provided that you specify only one input file, this flag may be combined with the `-o` flag to change the name of the assembly file created. Instead of compiling *foo.c* to *foo.s*, the following command compiles it to *bar.s*:

```
gcc foo.c -S -o bar.s
```

By adding the `-fverbose-asm` flag, the compiler will provide comments in the assembly code to improve readability. This may be helpful in debugging your code.

3.5 `-std=c99`

You may be surprised to know that an instruction like

```
for (int i = 0; i < 10; i++) {
   /* code here */
}
```

would be rejected by the default settings of `gcc`. Some standard versions of C require all variables to be declared at the start of the code block[1] in which they are used. However, in this course, we are permitting you to use the C99 standard of the C language, in which the above example is allowed. To have `gcc` compile according to this standard, thus allowing your code, you will need to add the `-std=c99` flag.

[1] A code block is anything within matching sets of braces or the single instruction following an instruction like `for`, `while`, or `if` without braces.

3.6 Warnings (`-pedantic, -pedantic-errors, -Wall, -Wextra`)

`gcc` by default does not display warnings when compiling. We encourage you to add the flags that check for and display warnings, as many such warnings suggest either that you are violating rules of C standards or that you may have entered code that does not do what you intended.

The compiler may accept code that goes against the current standard. Adding the -pedantic flag will cause the compiler to give you a warning when some standards violations occur, although compilation will not stop.

For example, using `//` for comments violates the ISO C90 standard, which is the default for `gcc`. The code will compile correctly, but the `-pedantic` flag will provide a warning that this is disallowed. This flag is recommended to ensure that you conform to C standards.

Using the `-pedantic-errors` flag instead will cause the compiler to give errors instead of warnings when standards violations occur, stopping compilation.

In C, forgetting the `return` statement in a non-void function is not an error like it is in Java. However, this may cause the program to have unexpected behavior[2]. It is therefore likely that not including a `return` statement is a mistake on the programmer's part.

The `-Wall` flag tells `gcc` to provide warnings for this and other possible mistakes, such as having an unused variable or function declared in your code. Another potential error is having code like

```
if(a)
  if(b)
    foo();
else
  bar();
```

where it could be confusing which `if` the `else` goes with[3]. As mistakes of this sort can be hard to find, `-Wall` provides warnings for them.

The `-Wextra` flag tells the compiler to provide additional warnings. For example, code like

```
if (a < b);
  foo();
```

is likely a mistake (as `foo()` will be called no matter what) but will still compile. This will, however, raise a warning with the `-Wextra` flag.

These warnings do not always mean that a mistake has been made, but it is worth checking to reduce the number of issues you may have to debug.

[2] When `return` is called in a non-void function, the return value is pushed onto the program stack and then popped by the callee. If no `return` instruction is reached, no value will be pushed to the stack, but the callee will still pop from the stack.

[3] In C, much as in Java, an `else` goes with the nearest `if`, even though in this example the indentation suggests that this was not the programmer's intention.

3.7 `-g`

Adding the `-g` flag provides debugging symbols, allows you to debug your code using the command `gdb` or to disassemble an object file using `objdump`. More information about `gdb` and `objdump` is provided below.

3.8 Optimization (`-O[0-3], -Os`)

In general, `gcc` attempts to generate code as quickly as possible, rather than performing optimizations. As a result, instructions are turned into assembly code independently of each other, making debugging with `gdb` easier. When one adds an optimization flag, `gcc` attempts to create more efficient code at the cost of compile time. Adding an optimization flag may make debugging with `gdb` a little more difficult, as the assembly code will not exactly match your C program.

The flag `-O0` is equivalent to performing no optimizations. On the other hand, `-O1, -O2,` and `-O3` correspond to increasing levels of optimization but increasing compilation times. `-O3` attempts to optimize speed at the cost of space, while `-O1` and `-O2` do not attempt to make space vs. time tradeoffs. Finally, the flag `-Os` will optimize while avoiding increasing the size of the optimized code.

4 `make` and Makefiles

The command `make` allows you to greatly simplify compilation, especially of multiple files, by reducing the necessary calls to `gcc` to a single command. With the `make` command and a Makefile, you can build, test, and/or "clean" (delete executables and other unnecessary files once you've finished) your project in a simple, concise fashion.

Makefiles can be somewhat difficult to write from scratch. To learn make, we recommend that you start by looking at examples; simply modifying existing Makefiles (we've provided you one in maze and one in life) is often the best way to learn the tool.

4.1 Writing a Makefile

A Makefile is just a text file, with the name "Makefile" or "makefile" (the capitalized version is preferred, but it doesn't matter), located in your project directory. At a high level, it is basically a script to be run by the `make` utility. The meat of a Makefile is a list of commands, organized as follows:

```
<target>: <dependencies>
    <shell command(s)>
```

The "target" is the object to be built — it need not match the name of the file that will be created, but it's helpful if it does. The "dependencies" are files needed to build the target; i.e. source, object or header files if you're buliding a C program. Finally, you can specify the shell command or commands that will performs the necessary actions to build the target.

Important: *You must indent commands with tabs, not spaces!* `make` *won't work if you use spaces! Also, having a stray tab on a line with no command will cause* `make` *to complain.*

4.2 Executing `make` and Special Targets

To run `make`, just type `make <target>` on the command line, in the directory where your Makefile is located. You can also run `make` without a target — it will default to the first target of the Makefile.

As you may have guessed, the targets `default` and `all` are generally not files, but rather are used, respectively, when `make` is run with no arguments, and to build the entire project. While you could, theoretically, have a file called `all` or `default`, that's not how these targets are used. These targets take as dependencies those targets that will need to be built to satisfy the `all` or `default` case, and usually don't have an explicit command. For instance, you might have:

```
default: program

all: program tests

program: program.o file.o
	gcc -g -Wall -o program program.o file.o

tests: program.o file.o tests.o
	gcc -g -Wall -o tests program.o file.o tests.o

program.o: program.c header.h
	gcc -g -Wall -o program.o -c program.c

file.o: file.c header.h
	gcc -g -Wall -o file.o -c file.c

tests.o: tests.c header.h
	gcc -g -Wall -o tests.o -c tests.c

clean:
	rm -f *.o program tests
```

Convention is to put `default` at the top, and `all` right below it, assuming both are used (neither need be). Running `make` with no target will always build the first target in the makefile, regardless of its name.

Another commonly-used target is `clean`, which is coupled with a command that deletes the executable and object files. Take a minute to look over the example above, and make sure you understand it — Makefiles are invaluable, and definitely worth the investment of learning how to use them.

4.3 Macros

It's also possible to specify "Macros" in a Makefile. These are declared (generally) at the top of the Makefile, and take the form of `<identifier> = <value>`. Generally, the identifier is written entirely in capital letters. To reference an identifier later, you must surround it with `$()`, much as a `bash` script requires a variable reference to begin with `$`. Macros are very useful, and we suggest that you use them when you can.

As an example, we've re-written the above makefile using macros:

```
CC = gcc
CFLAGS = -g -Wall
EXEC = program
TESTS = tests
OBJS = program.o file.o  # object files for program, NOT INCLUDING tests.o

default: $(EXEC)

all: $(EXEC) $(TESTS)

$(EXEC): program.o file.o
    $(CC) $(CFLAGS) -o $(EXEC) $(OBJS)

tests: $(OBJS) tests.o
    $(CC) $(CFLAGS) -o $(TESTS) *.o

program.o: program.c header.h
    $(CC) $(CFLAGS) -o program.o -c program.c

file.o: file.c header.h
    $(CC) $(CFLAGS) -o file.o -c file.c

tests.o: tests.c header.h
    $(CC) $(CFLAGS) -o tests.o -c tests.c

clean:
    rm -f *.o $(EXEC) $(TESTS)
```

5 `objdump`

The command `objdump` can be run on object (*.o*) files (obtained by running `gcc` with the `-c` flag) to obtain more information about them. This may be helpful in debugging. `objdump` produces much more useful information with the debugging information provided by `gcc`'s `-g` flag, as explained above, so it is often desirable to assemble the object files with that flag on.

There are two important flags for `objdump` that you may find helpful.

5.1 `-S`

The `-S` flag will show the disassembled object file interspersed with the corresponding C code. To use this flag on the object file *foo.o*, run the command

```
objdump -S foo.o
```

5.2 `-l`

When used in combination with the `-S` flag, the `-l` flag will also include information about which source files and lines correspond to the shown object code.

6 valgrind

Another useful tool for debugging programs is `valgrind`. `valgrind` is useful for checking to see if your binaries have memory errors. These memory errors include reading from or writing to invalid memory and performing read or write operations of the wrong size on valid memory. `valgrind` can also determine whether all space allocated on the heap by your program is freed by the end of the program (we'll talk about dynamic allocation of memory later in the course). `valgrind` works best if your binary was generated using the `-g` and `-O0` flags.

6.1 Memcheck

Memcheck is the default option for `valgrind`. Memcheck, which provides information about invalid memory accesses as your program runs, can be run on the executable *hello* with the following command:

```
valgrind ./hello
```

Memcheck is not perfect but is very helpful when you're debugging your program's memory usage.

6.2 `--leak-check=<no|summary|yes|full>`

When this flag is provided, `valgrind` will both run Memcheck and perform the specified level of checks when the program terminates to determine if there were memory leaks (memory that was allocated but not freed). If on, it will alert you how much memory was allocated but not freed. The command

```
valgrind --leak-check=full hello
```

will run a full memory leak check on the program hello while also running Memcheck.