In this meeting, we will write several small programs in C or Java and create our own Makefiles to compile our code. We will also use a debugger to examine and control our running programs.

Hello World

We have gone way too long in this course without writing a "Hello World" program. So take a minute to write either `HelloWorld.java` or `hello_world.c`. Including whitespace, these implementations should both take < 10 lines of code.

Now compile your code from the command line.

- What commands did you execute?

Run your program, and verify that it prints "Hello World!" to the screen.

Next, please load your program in a debugger (`gdb` or `jdb`). Place a breakpoint in the main method, and run your program (in Java, use the "help" command from within `jdb` if you get stuck). Once your program has hit the breakpoint, use the `list` command to examine the state of your program. What happens?

- What command starts a program in a debugger?
- What command sets a breakpoint? Runs the program?
- Basically, by the end of this meeting, we should have two very simple tutorials (with pictures and examples) of how to use a debugger: one for `jdb`, one for `gdb`, in a wiki page called "debugging"

Now compile your program again, this time including debug information.

- What option "turns on" debugging info?
- (For C only: what do the `-Wall`, `-Werror` gcc options do?)
- Repeat the same steps as above to set a breakpoint and examine the state of your running program's main method.

Making a Makefile

Now we will create a Makefile for compiling our hello world programs. The GNU Make webpage is excellent, and the Make manual's intro is a good starting point. Although you can use `make` for many tasks (and languages), a lot of the resources explain how to compile code in C. After you have read through *at least* the introduction (and actually read it), you might want to check out this resource for Java+make written by professor at Swarthmore, and another general example for those of you writing code in C.

For the C Crowd

Please create a Makefile that has the following properties:

1. A set of default compiler flags that include: -g and -Wall
2. a target called "hello" that compiles your source file from above into an executable called "hello_world".
3. a target called "clean" that deletes the executable "hello_world" and any object files (end in .o) that are created as part of the make process.
4. a target called "dist_clean" that deletes everything that the "clean" target does, plus all other automatically generated files (such as the backup files created by your text editor).

Verify that: when you type `$ make hello`, it builds your executable; when you type `$ make hello` again, it does not do anything; and when you type `$ touch hello_world.c`, then `$ make hello`, it rebuilds your executable.

For the Java Crowd

Please create a Makefile that has the following properties:

1. a set of default compiler flags that include: -g
2. a target called "hello" that compiles your source file from above into "HelloWorld.class".
3. a target called "clean" that deletes the class file "HelloWorld.class" and any other class files (end in .class) that are created as part of the make process.
4. a target called "dist_clean" that deletes everything that the "clean" target does, plus all other automatically generated files (such as the backup files created by your text editor).

Verify that: when you type `$ make hello`, it builds your executable; when you type `$ make hello` again, it does not do anything; and when you type `$ touch HelloWorld.java`, then `$ make hello`, it rebuilds your executable.

rot128

The next (blue) part the assignment is borrowed from Roberto Hoyle at Oberlin. I thought the assignment is fun and it continues to build on bash and Makefile basics.

Next you will create a program called rot128 that will "encrypt" files by using a rot-128 encryption algorithm. This algorithm is based on the classic text encryption scheme rot-13 which shifts each letter 13 positions in the alphabet (i.e., 'a' becomes 'n', 'b' becomes 'o', 'z' becomes 'm', etc.). Two applications of rot-13 returns you to the original. (Sadly, this has been used as actual protective encryption in commercial settings.)

```
Guvf vf zl irel frperg zrffntr rapbqrq va ebg13!
This is my very secret message encoded in rot13!
```

Instead of just rotating 13 positions, I want you to rotate by half the allowed range of a char, that way we can encrypt and decrypt any file on the system. Normally, a char is 8-bits, but since we can't be sure of that, you should have the program calculate using the proper constant.

```
In C:    UCHAR_MAX
In Java: '\uffff' (or 65,535)
--->  rot = (max + 1) / 2  <---
```

Add in lines to the Makefile that compile `rot128.c` to the target `rot128` (or `Rot128.java` to the target `Rot128.class`) and add the relevant parts of the files removed by clean (being careful not to delete your source code file).

When you write your program, you should just add the above value to all characters you read in, and immediately write them out.

- In C, you can use getchar() and putchar() to handle the input and output. Be aware that getchar() returns an int and you will need to do your processing on a char to have things loop around correctly. (Note that the resulting output won't be comprehensible, see the next section about how to store it.)
- In Java, it is a bit more difficult. You can create a Scanner from System.in, and then read characters one at a time using:

```
Scanner scanner = new Scanner(System.in);
scanner.userDelimiter("");  // after this, scanner.next()
                            // will yield 1-chararcter strings
char c = scanner.next().charAt(0);
```

Be careful to not write out the EOF signal.

Redirecting files in and out of a program

Your "rot128" program reads in from the user typing and writes out to the console. You can redirect the input from a file and also redirect the output to a file. Use the "<" character to redirect an input file, and the ">" to redirect output as follows:

```
./rot128 < inputfile > outputfile
```

(You may want to use gdb or jdb to debug your programs)

Using `diff`

There is a tool called `diff` on Unix systems that will report the differences between files. You can use this to check to see if your program is indeed working:

```
% emacs mytext.orig                          # create a text file
% java Rot128 < mytext.orig > mytext.enc  # create an encrypted file

% diff -q mytext.orig mytext.enc    # ask to see if they are different
                                    # -q tells it to not show the
                                    # differences

Files mytext.orig and mytext.enc differ

% java Rot128 < mytext.enc > mytext.dec # run your program on the rot128 text

% diff -q mytext.orig mytext.dec     # should have no output as they are
                                    # the same
```

Enhancing your Makefile

Add the following rules to your Makefile:

- add a rule to your Makefile called `rot128` that builds *just* your rot128 program.
- add a rule to your Makefile called `all` that builds both your hello world and your rot128 program.
- add a rule called `hellorot` that runs your hello world program and passes its output as input to your rot128 program (this should print "encrypted" Hello World! to your terminal)
- add a rule called `hellorotrot` that runs your hello world program and passes its output as input to your rot128 program, then passes that output back to your rot128 program (this should print Hello World! to your terminal).
- What does .PHONY mean, and should any of the above rules be .PHONY?

More Bash

The following guide is another good manual for Bash. There are also ebooks in the shared class directory, and physical Bash books in the lab. Please complete the following tasks (some with slight modifications) borrowed from http://www.tldp.org/LDP/abs/html/writingscripts.html:

Self-reproducing Script: Write a script that backs itself up, that is, copies itself to a file named backup.sh. (Hint: Use the cat command and the appropriate positional parameter.

Home Directory Listing: Perform a recursive directory listing on the user's home directory and save the information to a file. Compress the file, and ask the user the pathname to save the compressed file.

Converting for loops to while and until loops: Convert the for loops in Example 11-1 to while loops. (Hint: store the data in an array and step through the array elements.) Having already done the "heavy lifting," now convert the loops in the example to until loops.

Backwards Listing Write a script that echoes itself to stdout, but backwards.

Changing the line spacing of a text file: Write a script that reads each line of a target file, then writes the line back to stdout, but with an extra blank line following. This has the effect of double-spacing the file. Include all necessary code to check whether the script gets the necessary command-line argument (a filename), and whether the specified file exists. Then write a script to remove all blank lines from the target file, single-spacing it.

Backup: Archive as a "tarball" (*.tar.gz file) all the files in your home directory tree (/home/your-name) that have been modified in the last 24 hours. (Hint: use find.) Name your the tarball using the format:
`<USER>.backup.YYYY-MM-DD.tar.gz`.

Primes: Print (to stdout) all prime numbers between input integer i and input integer j. The output should be nicely formatted in columns (Hint: use printf).

Lottery Numbers: One type of lottery involves picking five different numbers, in the range of 1 - 50. Write a script that generates five pseudorandom numbers in this range, with no duplicates. The script will give the option of echoing the numbers to stdout or saving them to a file, along with the date and time the particular number set was generated.