## CS2303 Systems Programming Concepts C14

## Lab 2 5 points

# Make, gdb and Arrays

Lab 2 provides an opportunity for students to become familiar with creating make files, using the gdb debugger and writing C routines involving a global array.

You are to create four separate files: motion.h, print_field.c, motion.c and the source file for your main program.

Note – all output from this program is to be redirected on the command line to a file named, motion.txt.

## motion.h

The header file, motion.h, begins with the following lines:

```
#define LENGTH 18
#define WIDTH  12
#define SAMPLES 8
enum direction {Right, Left, Up, Down};
int field [WIDTH] [LENGTH];
…
```

and continues with any other global declarations that you need for your main program.

## print_field.c

This external routine does the following:

Prints out the current contents of the field with a border that contains a row of '-' chararacters above and below the field and a column of '|' characters to the left and right of the field. Print out the field such that the number of output columns is LENGTH and the number of output rows is WIDTH.

print_field prints out exactly three decimal digits for each integer element of field.

## motion.c

This external function takes the form:

**int motion (int nodeid, int row, int column, int direction)**

Starting in the **row** and **column** of the field specified in the function arguments, this routine places **nodeid** in that field location. Then, based on the direction integer (0-3), motion moves one spot in the appropriate direction, increments **nodeid**, and deposits the updated **nodeid** in the next position in that direction. Motion continues placing integers on the field until after it has placed an integer on the edge of the field.

## main.c

**main** then reads in **SAMPLES** lines of test data. Each sample line of test data contains four integers corresponding to:

**nodeid row column direction**

For each sample of test data your program does the following:

1. Clear the field (i.e., fill the field array with zeros).
2. Call **motion** using the sample test data as arguments.
3. Call **print_field** to print out the resultant field to *stdout*.

## Lab 2 Assignment

0. Prior to coming to the lab prepare a preliminary solution to the program above.
1. Create a make file*.
2. Use the gdb debugger** to debug your program.
3. Test the program under your own test data input from the terminal.
4. Run the program on the provided test data file 'lab2.dat' redirecting the output to motion.txt.
5. Create a README file that contains any useful information to assist in the grading of your lab program.
6. Create a tarred file that contains all the source and header files, the make file and the README file.
7. Use the Unix version of the 'turnin' to turn-in the tarred file. [The deadline for all lab turn-ins is 24 hours after the beginning of your assigned lab.]

*see make tutorial below

** see gdb debugger tutorial below

## Make Tutorial

Makefile is normally used when developing a complicated project. Programmers use make to avoid having to type the same commands repeatedly during program development. More importantly as the compiled code grows larger, using separate files for project components allows make to only recompile those components that are changed between subsequent testing instances. We demonstrate the use of make with a simple example:

```
$ ls
btree.c  btree.h  buffer.c  buffer.h  main.c
```

In this sample project, there are three source files and two header files. **btree.h** is accessed via an include directive in **btree.c**. Similarly there are include directives for **buffer.h** in **buffer.c** and include directives for **btree.h** and **buffer.h** in **main.c**. For this example, the **Makefile** consists of:

```
## Start of the Makefile
main: main.o btree.o buffer.o
	gcc -o main main.o btree.o buffer.o

main.o: main.c btree.h buffer.h
	gcc -Wall -c main.c

btree.o: btree.c btree.h
	gcc -Wall -c btree.c

buffer.o: buffer.c buffer.h
	gcc -Wall -c buffer.c

clean:
	rm *.o
## End of the Makefile
```

Note that a **Makefile** is composed of several segments as below:

```
OBJECTIVE: FILE1 FILE2 FILE3 ...
	COMMAND_THAT_ACQUIRES_THE_OBJECTIVE
```

FILES1, FILE2 ... are the files this 'OBJECTIVE' is dependent on. Namely, as long as one of the FILES is changed the 'OBJECTIVE' needs to be recompiled. Note, the first character of the second line needs to be a 'tab' and not several spaces.

To compile a program, simply run make in the directory where **Makefile** exists. For additional make examples see Professor Hamel's website: http://web.cs.wpi.edu/~cs2301/common/lab4.html

# gdb Tutorial

The purpose of a debugger is to allow the programmer to see inside a program while it executes [1]. Thus, if a program crashes a debugger can provide a good idea of what the program is doing at that moment. For this course, we use **gdb**, a free debugger from the GNU project.

**gdb** has many features and this tutorial will not try to enumerate all of them. You are encouraged to try to use the debugger and go to the man pages for assistance. This tutorial only covers the basic features **gdb**.

Starting from a standard C compile command line:

**$ gcc prog.c -o prog**

To use the debugger set the 'g' flag during compile, namely:

**$ gcc -g prog.c -o prog**

This creates your executable **prog.**

To run this executable with the debugger, type the following command:

**$ gdb prog**

This puts you inside the debugger.

**(gdb)**

Some of the basic commands you can use within the debugger are ([2]):

- ***run*** – This executes the program in the debugger, just as it would otherwise, but the program will halt at specified breakpoints.
- ***break (or b)*** - Enables the user to set break points within the code. For example, after a segmentation fault, one might set a break point near to a pointer manipulation. Additionally, setting break points inside functions helps to localize where the program is in error. Generally, one sets several break points to check variable values at a variety of locations within the code.
- ***step (or s)*** – Causes a single line of code to be executed. ***step*** will step the program control into a function if it encounters one.
- ***next (or n)*** – Goes to next line of code, but it does not enter functions.
- ***print (or p)*** – Is used to print out the value of any variable at any given point.
- ***backtrace (or bt)*** - Prints a stack 'backtrace'. This provides information about the programs traversal through functions.
- ***Quit (or q)*** – Causes you to exit the debugger.

We now walk through one example to demonstrate the use of these commands:

First, to set a breakpoint at line 20 with gdb in the code, type:

**(gdb) break prog.c:20**

Next, set a breakpoint at line 63 in the code:

**(gdb) break prog.c:63**

With the desired breakpoints set, now run the program in the debugger:

**(gdb) run**

The program runs as it should **until** it encounters a breakpoint. One might now use the step command after the program halts at a breakpoint. The essence of the step command is that you are now at the point where you believe there is a problem with the program and you can examine the results of each step of code carefully.

**(gdb) step**

At any point, if it is useful to print out the value of a variable, e.g., **x1**, then type:

**(gdb) print x1**

Now, **backtrace** can be used to show where the program currently has halted.

**(gdb) backtrace**

If you need the values of all variables currently in the stack, type:

**(gdb) backtrace full**

Finally to exit the debugger, type

**(gdb) quit**

References {from the CS2303-C14 Course Webpage}

[1] The gdb Debugger
http://www.delorie.com/gnu/docs/gdb/gdb_toc.html
[2] gdb Commands
http://www.yolinux.com/TUTORIALS/GDB-Commands.html