CS246 lab Notes #7 Makefiles and Multiple Source Files, and Miscellany

- Miscellany
    - touch
        - Accesses a file without actually doing anything to the file
        - Changes timestamp in ‘ls –l’
        - Also creates a non-existent file as an empty file.
        - A way to force make to recompile
    - Miscellaneous array stuff
        - Note: ARRAYS in C do not check Index-out-of-bounds
            - Arrays are just adjacent cells of memory.
            - There is no Virtual Machine to watch whether you go out of range of the array.
            - Thus you can overwrite other parts of memory if you go over
            - Particularly bad about this is scanf("%s", foo); This lets other people add malicious code instead of running yours by overwriting the stack.
        - When passing around multidimensional arrays, you can only omit the first dimension, all others have to be specified. This is because multidimensional arrays are in reality one-dimensional, and index calculation requires knowledge of all other dimensions.
        - The levels of indirection go left to right. In other words, if you need to perform your dereferencing one step at a time, note that the array that is dereferenced is the first one.
    - The –D flag to gcc (reminder)
        - Works just like **`#define`**
        - If the compilation is executed with –D<expression>, (no spaces) it is equivalent to have the #define <expression> in the beginning of your code.
- Multiple source files
    - Think about multiple source files almost like different classes in Java, as far as where you separate the different functionality.
        - Though keep in mind that in C this distinction is artificial, not built into the language as it is in Java.
        - However the advantages of the separation are still present.
    - Header files correspond to Java interfaces. A header file provides the interface between the client (any function wishing access) and the implementor.
    - In the header files, include the header files from whatever other source file you need the functions or preprocessor definitions from.
- Splitting your program into multiple files:
    - To use functions from another file, make a *.h* file with the function prototypes, and use `#include` to include those *.h* files within your *.c* files.
    - Separate the functions into meaningful modules.

    - Design the modules carefully so that you will not have to include all *.h* files in all *.c* files – very easily leading to circular **`#include`**, but can be avoided by the enclosing with **`#ifndef`** trick
- Compiling multiple source files.
    - .o files
        - A .o file is called an "object file"
        - It contains the compiled code for the functions within the file, but does not contain a "main" function
        - This file cannot be executed (since it does not have a main function).
        - To make a .o file, compile with the –c flag. You do not need to specify an output file.
        - Example gcc –g –Wall –c foo.c
            - Result: foo.o
        - Once you have all the .o files you need, you can compile/link them all together as normal.
        - Example gcc –g –Wall foo.o bar.o baz.o –o myprog
        - Note that this requires a lot of calls to gcc, and what is performed here is actually known as linking, The linker ld is called by gcc to link all object files in order to form a single executable.
- Makefile
    - A Makefile is a way to specify compilation dependencies.
    - Copy Makefile_example and Makefile_example_simple from /home/dxu/handouts/cs246 into your directory.
        - To use either, make sure you rename it "Makefile", with a capital M. make only works with a file with that exact name.
    - make is basically a programming language for doing all the steps of a complicated compile automatically.
    - It's language independent, but in this lab we show how to do it for C in particular
    - make is **VERY UNFORGIVING** about extra white spaces
    - macro assignment:
        - <MACNAME> = value
        - Example:
            - BINNAME = myprog
        - value is the rest of the line, including any white space on the line
    - Accessing variables $(<MACNAME>)
        - The variable is replaced directly with its value
    - Actions (tabs only for spacing!!)
        - Form:

            -
              ```
              <actionname>:     <dependencies>
                                <what-to-do line>
              ```
            - Example:

                -
                  ```
                  all:    foo.o bar.o baz.o
                          gcc –g –Wall foo.o bar.o baz.o –o myprog
                  ```

            - Note: the white spaces you see btw all: and foo.o, as well as before gcc are tabs!!
        - To execute an action, if you type: "make" it executes the first action.
        - Otherwise use "make <action>" to call that action
    - Everything in a Makefile is completely customizable.
    - Not just compilation:
        - clean:

          ```
          	-rm *.o $(BINNNAME) core
          ```
        - Note: "-" at the beginning of a flag causes the dependency to be executed even if an error occurs. Thus rm will work if all files aren't there (like no core dump in the directory).
        - run:

          ```
          run:	$(OBJ) $(INCLUDES)
          	gdb $(BINNAME)
          ```
    - There are a lot of dynamically assigned variables, can be used to greatly increase the power of make.
    - For more information, man make
    - Makefile tutorial
        - http://ailab.uta.edu/cse6362/make.html
        - http://www.mtsu.edu/~csdept/FacilitiesAndResources/make.htm