Building & Debugging

Outline

- Announcements:
  - HW I due by 5PM via e-mail
  - HW II on line, due in one week
- Building with make
- Old fashioned debugging
- Debugging tools

Good Design & the Genesis of Dependencies

- Modularity is a key feature of good programming
- Modularity begets lots of subroutines (functions/classes)
- Lots of subroutines begets lots of files
  - Keep related subroutines in their own file (library)
- Lots of files begets dependencies
  - changes in a subroutine often require changes in other subroutines in other files

Compiling multiple files

- Compiling multiple files is not a problem
  - cc -oappname -02 f1.c f2.c f3.c … fN.c
- but it can be frustratingly slow!

Compiling multiple files

- If you're only modifying one file,
  - 1) compile the files you're not working with to object code
    - cc -c -O2 f1.c f2.c…fM.c
  - 2) compile the files you're working with & link with objects
    - cc -oappname -02 f1.o f2.o…fM.o fM+1.c …fN.c
    - saves you the time of compiling the first files
    - if functions in f2 depend on fN, then the scheme before wouldn't work

Make

- make--standard UNIX tool for building software
  - typing "make" will force the make program to build your code according to the file "Makefile" in the current directory
  - At its simplest level, Makefiles are just scripts that control the build process
  - But, make allows you to define dependencies so that only the files that need to be compiled will be
    - very nice for development

Makefile syntax

- Make files contain 3 types of statements
  - Comments (start with "#")
  - Macros or variables (name = value)
  - Dependencies (two lines)
    - filename : files it depends upon
    - <tab>   command to execute if files are newer than filename
- Usually, Makefiles define macros first and then dependencies

Makefile Example

```
#Makefile for firsttry
#These are Macros--variables for use in the file
CC = gcc      #the c compiler we'll use
CFLAGS =     #place compiler flags here
PROGRAM = firsttry #the application name

$(PROGRAM): firsttry.c
        $(CC) $(CFLAGS) firsttry.c -o $(PROGRAM)
   #line must start with tab
```

When to use Makefiles

- Make really shines with large projects, with several files
- It is very useful when debugging
  - Use -c option and only compile files that change
- A good way to have others use your code
  - Hopefully, they'll just have to type make to build
  - May have to edit some lines: CC and CFLAGS

Generating Dependencies

- Some systems have the command "mkdepend" (mkdep on some systems)
  - mkdepend newmakefile *.c will look at the #include statements in the .c files and write dependency information to newmakefile.
  - You will still need to do some work
- Or you can do this yourself
  - Design descriptions and diagrams should be helpful

Old-Fashioned Debugging

- The point of debugging is to find your errors
- Simplest technique is **checkpointing**
  - Place an output statements around calls to subroutines
    - Printf("Entering subroutine A")
    - A();
    - Printf("Completed subroutine B")
  - If your program crashes in A, you won't see the second line
- Work into subroutines, bracketing sections of code with outputs until you find where the error occurs.

Old-Fashioned Debugging

- Checkpointing is nice because it works on any system that can run your code
- But, requires lots of compiles as you zero in on bug.
- WARNING: Finding the line where the program crashes is not enough, you need to know why!
  - The problem could result from a previous statement
  - In this case, figure out where the variables on the offending line are set, and work backwards