Unix System Tools

Makefiles, GDB, Valgrind

Compiling C

- We have already been compiling C with gcc
  - GNU compiler
- In addition to GNU there is clang (on a mac) and icc
- The C standard is a minimum specification for these compilers
  - Each compiler can have different additional options and libraries

Compiler flags

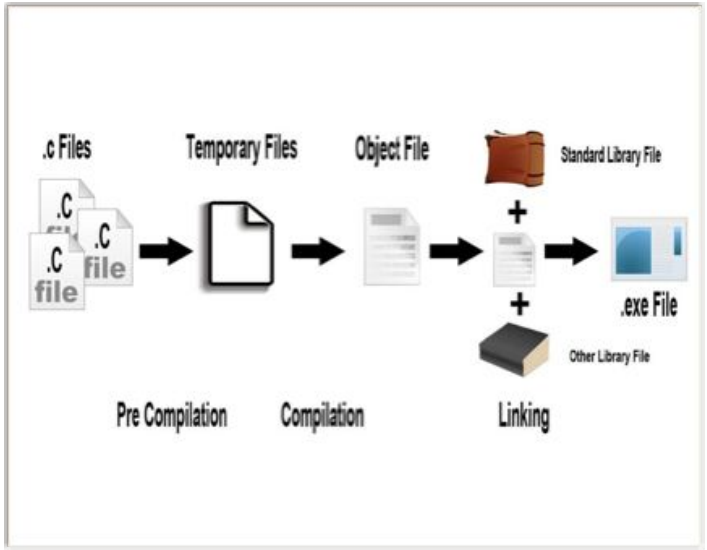
- Compiler flags are additional options to change the way your code compiles
  - Many of you have already used -lm
    - *tells the compiler to load the math library*
- Using compiler flags is as simple as preceding the flag code with a dash
  - gcc myfile.c -o myprog -lm

More Compiler flags

- Other compiler flags we will use:
  - -Wall
    - *Add all warnings*
  - -g
    - *Add debugging information*
  - -std=gnu11
    - *sets the standard to gnu11*
  - -pedantic
    - *Issue all the warnings demanded by strict ANSI C*

Compiling with Make

- Make is a command line program on *nix systems with two modes:
  - Uses system defaults for simple compilation of a single file
  - Looks for a compilation script called a makefile for complex compilation



Makefiles

- make without arguments looks for ‘makefile’ or ‘Makefile’ script file in the current folder
  - makefile is just another text file
- A makefile is a script containing
  - Project structure (files, dependencies)
  - Instructions for a program’s creation

Example Makefile

```
P=<program name>
OBJECTS=
CFLAGS= -g -Wall -std=gnu11
$(P): $(OBJECTS)
	gcc $(CFLAGS) $(P).c -o $(P)
```

Makefile Syntax

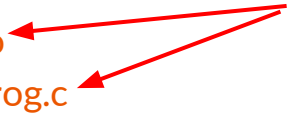
- Makefile is a line-oriented, whitespace sensitive file
  - That means each command must fit on a single line
- Spaces are not the same as tabs
  - Makefiles use whitespace to know what instructions should be run

Makefiles: Targets

- Executing a make with a parameter jumps to that ‘target’ within the makefile and runs the script
  - target: flags that mark locations in the file
- targets are fully left justified followed by a ‘:’
  - Examples:
    - *all:*
    - *main.o:*
      - When called without a target, make automatically looks for the target ‘all’

Makefiles: Dependencies

- Dependencies follow targets
  - If the dependency is a file name, make will check timestamps to see if the file has changed
  - If the dependency is a target, make will check to see if that target has been run
- Example:
  - all: prog.o
  - prog.o: prog.c



Makefiles: Scripts

- A short script should follow each target, each command should be a single line
  - Scripts are a series of commands that get executed automatically
- A command requires one tab (do not use spaces), and immediately follows the target
- Example:
  - prog: prog.c

    gcc prog.c -o prog

Object Files

- What if you have a large program of 10000 files?
  - Do you recompile all 10000 files every time 1 file changes?
- You can compile down all files to object files first
  - Just the machine code without any additional libraries
- The -c flag compiles to an object file
  - gcc -c file.c -o file.o
- You can then link together object files to make an executable
  - gcc file.o -o prog

Only Compile What's Changed

- How can this save compilation time?
  - Make checks the timestamps of its dependencies
  - If the dependency target modified time is newer than the target output file, then it needs to be recompiled
- Only the files with more recent modified timestamps get recompiled
  - The other object files can be left as is
- Everything then gets linked back together

Makefile Variables

- make allows us to put variables in the makefile
- Variables are conventionally uppercase
- Variable are essentially text replacement, so whatever you set the variable to is what will be put into its place
- To use a variable, you must use the $ and parenthesis
    - P= this would be an error

      ...

      $(P)

Classwork: Makefiles

GDB

- gdb is a unix environment debugger
  - Type "gdb <executable>" to use
    - *<<executable>> is the name of the compiled file you wish to debug*
- run your program by typing "run"
- You can display a chunk of code by typing list (or ‘l’)
  - list main
    - *displays 10 lines of code centered on the function main*
  - list
    - *displays the next 10 lines of code*

Moving through your Code

- Stepping through the code
  - Executes the next line without stepping into functions
    - *next*
  - steps to the next line or into a function if a function call
    - *step*
  - executes to the next function call or return
    - *finish*
  - continues execution to the next breakpoint
    - *continue*

Breakpoints

- Breakpoints allow you to stop the code during execution and inspect the data
- You can set breakpoints in various ways:
  - To set a breakpoint at a specific line number within a specific file
    - *break <filename>:<linenumber>*
  - To set a breakpoint at the beginning of a function declaration
    - *break <functionname>*
  - List all breakpoints
    - *info break*
  - Delete breakpoint by number
    - *del #*

Using gdb to inspect

- to inspect the value of local variables and parameters
  - info local
  - info args
- To examine a variable value,
  - print <variablename>
    - *Must exist within current scope and be after it was initialized*
  - For an array, dereference and use the @num to show that number of elements
    - *p *ptr@5*
- To set the value of a variable
  - set <variablename> = <valuetoset>

More Commands

- To inspect the stack
  - backtrace
    - *Shows the stack trace and the parameter values to each function call*
      - #0 reverseInts (n=0x7fffffffe8e0, size=4) at pointers.c:15
        #1 0x000055555555480b in main () at pointers.c:25
- Command help
  - help - lists command categories
    - *help <category> - lists commands*
      - example: help stack

Classwork: GDB

Valgrind

- Valgrind is a set of tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.
- Call using valgrind command and run command
  - valgrind ./<executable>

Valgrind Output

- Valgrind offers many different tools and many different options. We will use it for memory checking
  - For full leak checking use the flag --leak-check=full
    - *valgrind ./myprog*

      ...

      HEAP SUMMARY:

      in use at exit: 4 bytes in 1 blocks

      total heap usage: 1 allocs, 0 frees, 4 bytes allocated

      LEAK SUMMARY:

      definitely lost: 4 bytes in 1 blocks

      indirectly lost: 0 bytes in 0 blocks

      ...

      ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)

Valgrind Errors we care about

- Illegal read / Illegal write errors
  - When your program reads to or write from uninitialized or invalid memory
- Use of uninitialised values
  - When you use values that were never defined or initialized
- Illegal frees
  - Memory that was never allocated is being freed

Classwork: Valgrind