10/18/17
CS 2750

File *infile, *outfile
float sides;
int count;

fscanf

- C does NOT keep track of anything w/ strings
- char mystring 18;
  ↳ 17 spaces + null
- Nobody checks string characters for you. Array of bounds's been prompted
- String or Array name w/ no index is a pointer to first spot.
- strchr is a found function; it wants a second I; use another to loop ???? Useful for taking a string & breaking it up into pieces.
- Examples are online on schultz's website
  compile: gcc name of file
- Dynamic Memory → one of most powerful things
  (malloc → takes argument for how many bytes you want.
  ↳ return is a pointer to that memory
- I want an array, how do I know how big array elements are?
  ↳ Answer: declare variable int *myarray = (int *) malloc (17 * sizeofint);
  ↳ 17 ints long

☆ YOUR Job to make sure you do NOT go outside your array ☆

10/23/17
CS2750

- The C compiler gcc (cc); Compiler job is to transform your program into something that will run. It translates C into an executable
- C-File → [preprocessor] (#include, #define, #if)

preprocessor feeds its code into [compiler] which converts C code into assembly which then runs that [assembly] which converts it machine code. Then we pass it to a [linker] the linker takes .o files & libraries and gives them all together to make executables

- File extensions count in gcc →
  - .c C-code
  - .cpp C++ code
  - .i pre processed code
  - .o single file machine code
  - .a libraries

- Both gcc & g++ will compile both .c files & .cpp files
- Same extension, same actions, only difference is libraries
- Each of these takes a variety of options to gcc
  - -o file name of output file; by default for object file is code file name w/ .o extension
    the default is a.out
  - -l name lets you include libraries in linkage; -lm includes math library
    need both -lm on command line & #include <math.h> in source
  - -c stop after assembly & before linkage
  - --help help about the options
  - -O [1 2 3] example -O1 -O2; Optimization; make it run faster in some way
  - -Wall produces warnings → example if (i = 3) → wrong! should be if(i==3)
  - -g some debugging information
  - -p or -pg some info for profiling; can extract info about where program spends most time
    w/ a public program it lets me get info about how much time is spent in functions
  - -D name=value defines a symbol for preprocessor
    example: -D YYDEBUG
  - -I directory, this dir is a place to look for files to #include

10/25/17
C52750

Compiler gcc (C)
g++ (c++)

-wall [warns against everything possibly bad]
[possible errors]

Options: -o
-c
-lm [math library]
-O

c-file → [gcc] → [Linker] → executable

- a project may have a large # of files.
- Next thing is a tool to help us w/ compilation of big projects
- The tool is called MAKE → command make
  command make [-f name of description file] default file - makefile or Makefile
- command make [-f] [target] → if no target, default is 1st target on file

---

- format of make file; pairs of lines; dependency line, command
- ☆ Command line must start with <tab>
- a dependency: is a target : files it depends on (blank seperated)

convert.c dependency convert.o dependency

c-file → [gcc] → [linker] → executable

▷ example:
convert.o : convert.c [global.h] → since convert.c has #include "global.h"

- a command is what you would type to do conversion by hand
  ▷ we would type | gcc convert.c
- We'd like to have "variables"
  often most .c files depend on all the .h files
- Iteration I would force us to type all of them for each dependency line
- Lets produce a "variable" that contains all the .h filenames
- To define the variable, we put in a line that has the variable name, an (=) sign, list. — traditionally VARS is all upper case

var = list

$(variable name)

- a variable usage is $(var name)

- for many dependencies, the command we run is often the same.
  - Example: for every .o file w/ a .c source we run gcc
    for every .o file w/ a .f source we run gfortran
  - It would be nice if we could build a rule that says just that
  - There is a set of built in such rules & there is a facility for us to write our own.
- To write our own rule, we write a special dependency line, and use special variables in corresponding command line

old way: source definition prefix

.c .o :

<tab> command

$@ = target [o file]

$< = source

example

<tab> gcc -o $@ $<

CS 2750
10/30/17

Make - a command to help with compilation

- ~~command make~~ - by default it reads - makefile / Makefile -f file
- command make [target]
  the default target = 1st in file
- The make file is pairs of lines
  target : source files
  <\t> : command source -> target

Abbreviations : NAME = list (comma seperated)
  $(Name) in rules is the list

- There are a number of predeclared abbreviations for the compiler
  CC .c-compiler CPP c-plus-plus compiler

- pattern rules ; there are many cases where for a set of extensions we always do the same thing
  - For example: for every .c file we run gcc to get a .o file

the old form | .source.ext.destination.ext:
                command

the new form | %.destination.ext : %.source.ext command

For both $@ is full name of target
  $< is full name of source

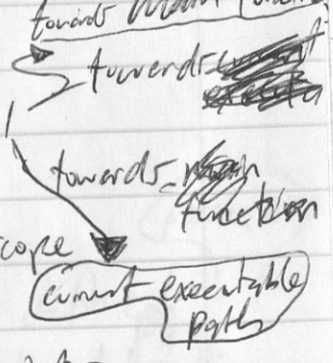
- for example: %.o : %.c
  $(CC) $< -o $@

- You can write your own
  for the old form, make must know which extensions are used
  put in a .suffix: extension list

- A target need not be an executable all: <u>all targets to make</u>
  for example: all: $(OBJS)
  clean:
  realclean:
- Debugger: We remember ctf A pointer, write a program; compiler great, run it, A we get "segmentation fault, core dump"
- The debugger lets us run our program under control
  - where we are in our program @ fault
  - step through the execution
  - examine values of a variable
  - tells us when a variable value changes
- For the debugger to work, it needs information, For every use of gcc/gtt must add -g option which makes executable bigger this does <u>NOT</u> affect the speed

11/1/17
CS 2750

- Debugger = what to see the program as it's running, and @ points as it runs
- To gather debugger info, use -g option for every step in compilation
  gcc -g ——— or g++ -g ———
- Usually puts in makefile CC = gcc -g
- The program runs as fast as original, but is bigger in size
- To start debugger gdb exec name [no command line arguments for executable]
  gdb args exec command line args
- What commands does gdb have? quit
  - run : run until end of program or fault
  - step # : execute # of lines (default 1)
- when executing it pauses on steps
- bt (blacktrace) show call stack function called & not yet exited
  - it gives line number that has started arguments to all functions
- At any point where exec has paused, bt can be used
- Variable scope is scope for top of call stack.
  - command to change scope - up # move # scope level
    down #



- I can show the value of the variable in current visible scope
  p var or p expr
- I'd like to stop executable automatically @ particular points
  this is a breakpoint.
  break { funct. name — breaks @ start of function
  # file : line — line # in current function
  file : function }

Debugging

Nov 6, 2017
CS 2750

- gdb
- Debugger – we need -g option on gcc (g++)
- break { function name / line number / file:funct. } stop executing @ a point

---

p expression [step # / s # / next #] execute # lines & stop again &

& [stepping into function / stepping over function]

---

watch / rwatch / awatch } expression stop execution when expression is { changed / read / both }

- delete range remove breakpoints/watch points in that range
- disable range make inactive
- info breakpoints list breakpoints
  info watchpoints list watchpoints
  - For both; I can attach gdb actions to execute
    - for example: break mkTree
      commends
      { p node x
      { p i
      end

continue / cont } continue execution

After a break/watch [run restarts from beginning]

- It might be nice if I could set the same breakpt/watchpt in every file
  - ex: A project that I am not sure if fixed. Want to set the _____ every run

BACK

Nov. 6, 2017
CS 2750
Page 2

- There is a file read by gdb as it starts.
  - .gdbinit

  You can put a things to happen when gdb starts
  ex: break main
  break check stats
  rwatch root
  break turnin.c:23

  Put in current dir [apply only there]
  Put in home dir [apply for all gdb sessions]
- Maintaining Versions. We'd like information to be able to recover prior version of software.
  1. To recover from alterations gone bad
  2. To keep a stable version while building a new version.

  possible commands
  SCCS original
  RCS recover

Nov. 8. 2017
CS 2750

Version Control - keep current version & all previous versions

RCS → Revision Control System - keep the current version & difference between past versions

- The version numbering system has 4 levels; numbers separated by dots.

major version . update . current version

- All file storage files are in a subdirectory RCS

step 1: mkdir RCS

step 2: put initial version files into RCS by command ci filename

by default, ci removes file from current directory

step 3: get files command co filename - read only copy

co -l filename - locks file so only 1 person can have it for changes

- Often we don't want the file to be remove from our directory when we check in

When I check in, use option ci -l filename saves & leave copy still locked

ci -u filename saves & leave a copy read-only

- for both options -r version

- There is an admin command rcs ① an option to override the lock

② option to set an access list ③ an option to remove versions

Nov. 15, 2017
CS 2750

process.pdf

- A typical use of fork

```
if ((pid == fork()) != 0) {
    do stuff }
else
{ execute() }
```

	List of Value	Explicit Ternary
	exec	execv
search $PATH	execlp	execvp

not finished

6 exec functions how argv is passed

exex * (file to run < arg[0], arg[1] ... NULL
array of arg i w/ last entry being NULL

example: run vi (in PATH) w/ argv[0] be "ex" on both proj2.c & proj2.h

```
exec lp ("vi", "ex", "project2.c", "proj2.h", NULL)
```

```
char args [4] = {"ex", "proj2.c", "proj2.h", NULL}
    execvp ("vi", args)
```

CS 2750
Nov. 29, 2017

Time - measured from Epoch
32-bit number, it rolls over in 2038

int time (int *tloc)
# of seconds from epoch

- int gettimeofday (struct timeval *tp, struct timezone *tz)
  (comma)
- struct timeval → tv_sec # seconds from Epoch
  → tv_usec # milliseconds after that second
- struct timezone → tz_minutewest which time zone
  → tz_dsttime details about DST
- We'd like day, hour, minute, second
- struct tm int tm_sec
  " " tm_min
  " " tm_mday → day of month
  tm_year
  tm_wday
- int mktime (struct tm now)
- double difftime (time_t one, time_t two)
- ALL above defined in time.h

const char * format, const struct tm *tmptr
format is a string (file format from printf)
%% → a literal percent
%a → abbreviated weekday name %A full name
%b → abbrev month name %B
%d → abbrev. day of month
- ASCTime (struct tm)
  ctime (Atime, vclock)