1.16.24

Fundamentals of Computing: Lecture 1:

Interpreted Languages: abstract, executed directly, slower
ex. Python
Compiled Languages: more exact, converted into assembly, faster once compiled
ex. C, Java

course is taught in C on a linux environment

To Access Linux Environment:
1. Open Windows PowerShell
2. ssh netID@machinelocation
   ex. ssh akarama2@student[10, 11, 12, 13].cse.nd.edu
3. enter netID password

Commands:
pwd → displays location in directory
whoami → displays user
users → displays all users on machine
w → gives info about all users on machine
mkdir [name] → makes directory
ls → lists all subdirectories
cd [name] → enters subdirectory
cd .. → leaves subdirectory
tree → displays all directories
exit → logs out

Lecture 2: 10.9.24

- where you land in the machine is your own private home directory
- there is another directory to submit homework

* ssh - secure shell
* if you are off campus you need to VPN to NDIP

commands:

echo $HOSTNAME → shows your location
pwd → print working directory
"your home directory"
"absolute path" →

symbols:

~ → home directory
.. → back up one level
* → show everything

can use:
ls ..
tree ..
cd ..

* directory you are in is your working directory *

commands:

tree [name] → makes tree for subdirectory

* GUIs are restrictive, Unix / Linux is a command-line based OS *
* a node must only have one parent, but can have any # children *

symbols:

/ → directory separator
(\ ) → windows directory separator

commands:

cd [name] / [name] → one step directory navigator
cd .. / [name] → backs up and enters new directory

commands:
cd ../.. → backs up 2 directories
cd [name]/[name] → relative path
cd /escnts/home/[user]/[name]/[name]...
→ absolute path

relative path: from one point to another
absolute path: fixed from top

cd ~/[name]/[name] → relative path
anchored from home directory
clear → clears screen
cd ~ → takes you to home directory
(or just cd)
cd ~[user] → takes you to another
persons home directory

flags:
-l → long listing ← ex. ls -l

commands:
touch [name].[filetype] → creates a file
info [command] → gives info
◦ space flips page
◦ q exits screen
man [command] → gives manual on command
◦ enter goes down a line

symbols:
. → shows current level

commands: ↱ ornament

mv [filename] [new filename]

"rename" (source)

* tab autofills entries *

* tab again gives autofill options *

mv [filename] [directory]/[directory]...

- moves file through directories
- based on current directory

* you may not have two entries with the same name in a directory *

* good habit to always put a / after mv command

*can move and rename at the same time

cp → copy

*touch updates timestamp if file already exists

rm → removing a file

rmdir → removes an empty directory

lecture 3:   1.22.24

PuTTY → different way to access linux machines
- allows saving sessions and different configs
- auto-logs into ssh
- putty.org

vim → text editor
- learn it

create a file:
vim [name].c

vim:
has command mode vs edit mode
i → INSERT (edit mode)
- anything typed goes into text
- not the case in command mode
esc → command mode
- what is typed does not go into text
O → goes down (insert after)
dd → deletes a line
x → deletes
a → appends
:w → saves file
:set number → shows line numbers
- local setting
  - need a personalized setting
:set mouse=a → allows use of mouse
shift + ZZ → saves and exits

command:
from home directory
vi .vimrc → dot/hidden file
- global vim configurations
ls -al → can combine flags

ls doesn't show ls -a does

compiling:
saves
exit ← annoying
compile *use two sessions
run executable to avoid this*

*have one session for compiling/running,
and one for editing files*
u → undo in vim
compiling:
gcc → gnu c compiler
gcc [name of file]
no msg means no error
creates a.out file
./a.out → runs the program
(. means current directory)
(only one can exist at a time)
-o [name] names the executable

C:
I/O → input/output
printf(); → prints string
\n → new line
\t → tab character
\v → vertical tab
\a → rings bell
%d → placeholder

lab 01: 1.22.24

vimtutor → opens a VIM tutor

moving cursor:

^
k
<h l>
j
v

esc → normal/command mode
:q! <ENTER> → exits vim, discarding changes
x → deletes char under cursor
i → inserts text to the LEFT of cursor
a → appends text to the RIGHT of cursor
:wq → saves and exits a file
vim <FILENAME> opens a VIM file
dw → deletes a word to the RIGHT of cursor
d$ → deletes a line to the RIGHT of cursor
d motion: 'delete' operator + motion
w → start of next word, EXCLUDING its first character
e → end of the current word, INCLUDING its last character
$ → to the end of the line, INCLUDING the last character
w → moves cursor to start of next word
e → moves cursor to end of next word
* typing a number before a motion repeats it that many times *
0 → moves to the start of the line
* works for d operator *
dd → deletes entire line
u → undo
U → returns a line to its original state
CNTR-R → undo the undo's
* operator [number] motion *

wget <LINK> → downloads files from links
- highlight link, copy, and right-click
  - NOT Control-c / v

gcc <FILENAME> -o <NAME> → compiles file with given name

Lecture 4: 1.24.24

stdio.h → standard input/output library
#include → includes a library
rm .ex*.swp → removes all swap files with * acting as a variable
.*.swp → removes all swap files
\ → escapes " "
\\ → escapes \
%d → placeholder for int
%f → placeholder for floats
%lf → placeholder for double
& → address operator
* white space means nothing on input
* numbers are always right justified

Zybooks assignment 1:
- compiler will sometimes predict an error past the actual error
  - check preceding lines / never succeeding lines
- compiler will sometimes mistakenly report an error

Syntax errors → detected by compiler
→ compile-time error
logic error → bug, doesn't do what expected but still compiles
compiler warning → indicates a logic error but does not prevent code from being compiled
gcc -Wall → shows warnings while compiling

Lecture 5: 1.26.24

selection statements model decision making
1. if statement
2. switch statement

If statement:

```
if (condition) {
   then-statement
};

if (condition) {
  then statement;
} else {
   else-statement;
};

printf ("string");
printf ("string %d\n", variable);
scanf ("%d", &variable);
```

% operator → modulo

Expressions:
1. Arithmetic → +, -, *, /, %
   A / B = Q
   A % B = R
2. Relational → >, <, >=, <=, ==, !=
3. Logical → &&, ||, !
   (AND) (OR) (NOT)

if an expression evaluates to 0 its T/F value is false
if an expression evaluates to a nonzero, its T/F value is true

Diagrams:

expression

non-zero (true)

then statement

zero (false)

next statement

always use curly brackets, but it is only needed if there is more than one statement

non-zero (true) expression zero (false)

then-statement else-statement

next-statement

Switch Statements

```
switch (expression)
   case value1 :
        case1-statement ;
        break;
   case ...
     :
     :
  default:
     default-statement
next-statement
```

lecture 6: 1.29.24

logic → boolean algebra → mutually exclusive → either TRUE or FALSE

* in C, logical levels are reflected by indents
* in Python, indents indicate logical levels

vim:

yy → "yanking" lines into memory

[#]yy → yanks multiple lines

* any indenting style works, but be consistent
* if there are no curly-brackets, only first command is part of if-else statement

m = 3 assigns 3 to m

m == 3 checks if m = 3

= → assignment

== → equality (mathematical)

include <stdbool.h>

allows use of booleans, like "TRUE" and "FALSE"

true, false

and declare bool vars

boolean → integer

true : 1

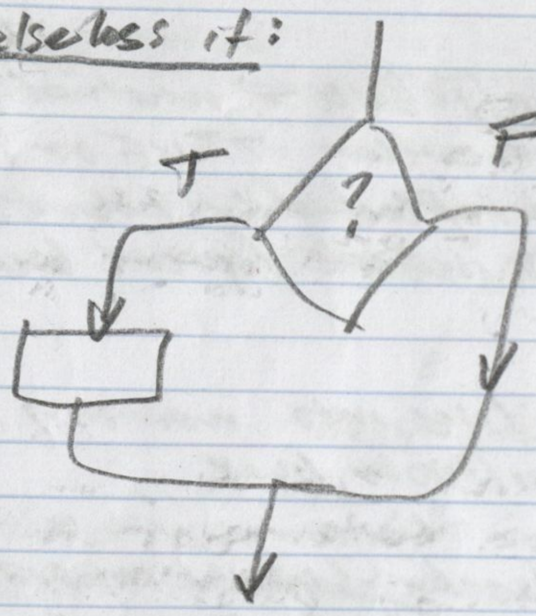
false : 0

integer → boolean

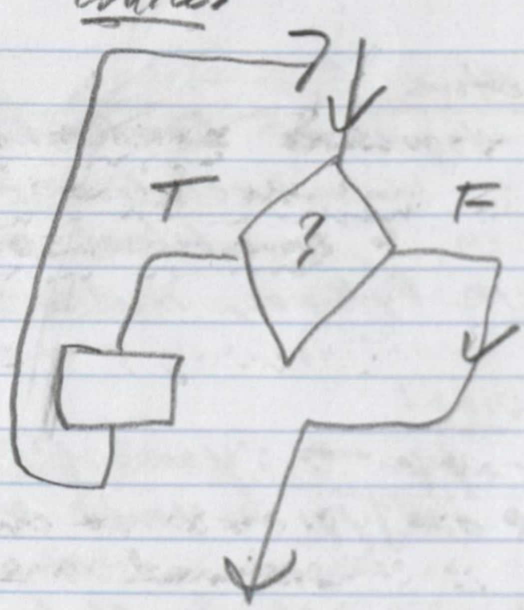
non-zero : true

zero : false

elseless if:



while:



while loop → condition - controlled loop
for loop → counter - controlled loop

3 - key parts of loops:
1. initial condition
2. repitition / termination condition
3. update (variable that controls loop needs to be modified so the loop is eventually false)

* anything that can be done with a while loop can be done with a for-loop vice-versa

for (A; B; C)
A is done once and before anything else
B is done at the beginning of each iteration
C is updated at the end of each iteration

break; → kicks you out of the CURRENT structure
continue; → jump to the end of the current iteration

Lab 2: 1.29.24

#include <stdio.h>
#include <math.h>
- have to use -lm in gcc
  = "library math"

pow(value, power)

Lecture 7: 1.31.23

* one BLOCK of code (if, for, ...)
  is considered one-statement and
  above blocks do not necessarily need
  curly-brackets

  float → %f (for 2 decimals → %.2f
  double → %lf

Lecture 8: 2.2.24

%g → no decimals
operations on the same data types gives
the same data type
operations on different data types gives
precision matching the most precise number
  float • double = double

* lots of notes in notes.txt and ex14.c *

lecture 9:   2.5.24

- functions are called from above
- int main() is called from the command line
- return 0 means the function was a success
- never put functions above main()
- all of C is written in functions
    - functions in C cannot return more than 1 variable
- [type] function() dictates what it returns
- variables inside a function are local
- we are not passing a variable yet, just a copy of its value
- must declare a function after main
    - function prototype
- void → nothing (no return value)
- default return is int
- void can return; just cannot return value
- exit; leaves the whole program

Lecture 10: 2.7.24

```
if (x>y) z=x;
else z=y;
```

syntatic sugar: ternary expression

```
z = (x>y) ? x : y;
```

%g removes excessive zeros in output

global variables are bad

Lecture 11:

Loops:

A: initialization

B: condition

C: update

you have to finish an iteration to exit a loop

flag → boolean operator like a mailbox flag

![var] → [var] = false

one break is not enough to get out of a double for-loop, obviously

Lecture 12: 2.12.24

arrays in C are static → fixed memory size
arrays in matlab and python are dynamic
cannot hold different data types in C arrays.
similar to subscripts in math

$t_1, t_2, \dots t_n$

$\Rightarrow t_{ave} = \sum_{i=1}^{12} t_i / 12.$

- each integer is 4 bytes.
  arr[5] is 20 bytes
- indexing begins at 0
  (matlab starts at 1)
  (python starts at 0)
- elements of an array are contiguous in memory

* no negative indexes *

in matlab, python, ect. → unassigned array indexes are 0

* in C, there is junk "garbage" in memory → values are not 0

- any values that are 0 is coincidence
- must arr[size] = { };
  ← empty list

will initialize everything to 0
cannot initialize everything to n
0 is also the null character

* array offset

arr[5] = {1, 2, 3}; arr[4] & arr[5]
will be 0 b/c of { };

extra elements in array initialization gives a warning, but never do it

int arr[]; → not allowed (in main)
int arr[] = {n1, n2, ... ni} → is allowed

sizeof() → returns # of bytes allocated to something
- ints → 4 bytes ← float too
- double → 8 bytes
- char → 1 byte

int size = sizeof(arr) / sizeof(int) ← or arr[0]
- only works in this scenario

* the word size is very ambiguous *
sizeof() is not ambiguous
array size → how many values can it store

* there is no universal way of identifying the size of :
arr[5] = {1, 2, 3};
never initialize this way

initialize like
arr[5] = {1, 2, 3, 4, 5}
arr[] = {n1, n2, ... ni}

n=0 1

0

Lecture Notes 2.14.24

a) safe

int a[5] = {1, 2, 3, 4, 5}; - safe

b) safe, impossible to calculate size

in b[10] = {1, 2, 3}; - safe, rest are 0s

c) not safe

int c[5] = {1, 2, 3, 4, 5, 6}; - not safe
- compiles
- not enough memory

d) implicit

int d[] = {1, 2, 3, 4, 5}; - implicit

to find size: int size = size of(d) / size of (d[0]);

int num[3];
num = {1, 2, 3}; - NOT allowed

& only can fill an array at declaration time

#define - global macro for constants
- directly substitutes its value

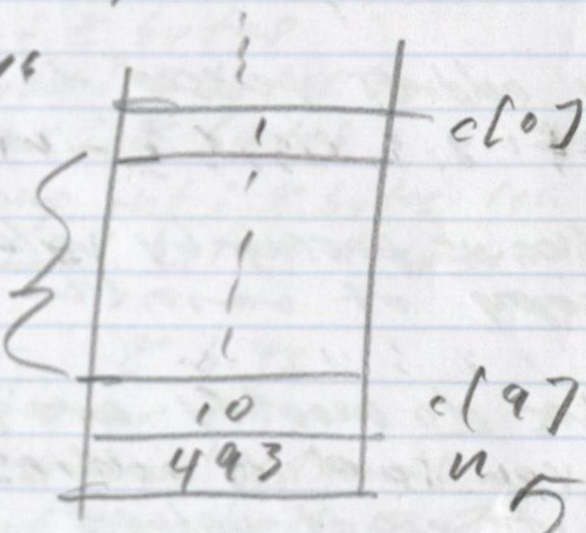
no semicolon b/c preprocessor directive
NOT a statement

const int [var] - prevents change

variables evaluate, macros are replaced

```
int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int n = 493;
c[10] = 77;  ← n now = 77
```

memory:



c[10] places it here

compilers try to be efficient
gcc will not detect out of bounds

* you need to pass the size of an array to a function to pass it
* sending the array only sends its address

Lecture Notes: 2.16.24

* you don't have to pass the size of a String array
* you otherwise always have to pass the size of an array

size of array address passed to a function is 8 bytes

passing variables is passing by value
- photocopy of variable

BUT, functions do modify arrays
- because you send its address and size
- effects its exact values
- passing by reference

make arrays const to "pass" by "value" as it cannot be changed

Lecture Notes! 2.19.24

memory of data types!
int: 4 bytes
float: 4 bytes
double: 8 bytes
short int: 2 bytes
long int: 8 bytes
long long int: 8 bytes (on our systems)
char: 1 byte (represents characters)

1 byte = 8 bits 2^{31} vs 2^{32}
1 bit = 0 or 1
2 bit = 00, 01, 10, 11
amount of numbers = $2^{\# \text{ of bits}}$
negative numbers take up half of memory
signed vs. unsigned variables
variables default to signed
use unsigned int to utilize more space

ASCII code (now use unicode):
encodes characters
256 bits is enough for all characters

%c : placeholder for characters
%d will print a char's ASCII code
(in decimal)

%c will display the character associated
with an integer

you can deal with chars as ints and ints as chars

This has some nice properties:

```
char++;
char = char - #;
char = char - 'char';
```

- if 'a', will return # of letters b/w char and beginning of alphabet

can do loops on chars

can use chars as a for-loop COUNTER

```
for (int i=1; i≤26; i++) {
    printf("%c ", char);
    ch++;
}
```

does the same thing as

```
for (char x = 'a'; x ≤ 'z'; x++) {
    print("%c ", x);
}
```

and

```
for (int n=97; n≤122; n++)
    printf("%c ", n);
```

a = 97
A = 65 → rest can be deduced from here

printf("%c", 8) gives much more
printf("%c", 13) control over cursor

Type casting converts variable of one type to another

- works with chars

char arrays:

string cannot be longer than array size

```
char string[20] = "notre dame";
printf("%s\n", string);
```

%s displays strings
there are NO string data types in C
must use char arrays

individual chars: ' '
strings: " "

size of string + 1

- strings are 2 bytes b/c of \0
- but char arrays are 1 byte per index

```
#include <string.h>
  strlen(string); // displays length of string
  // starts at address of 1st char and counts until '\0'
char string[] = "hello";
```

- can also display strings like this
- sizeof(string); // = 6 b/c of \0
- strlen(string); // = 5

5 \0

*every C-string ends with a NULL character
- acts as a sentinel

You do not have to pass the size of a char array when passing to a function
- because of the sentinel

%s is much different than other values
- takes the address of array passed and keeps printing characters until NULL character

String = "blah" // NOT allowed

Lecture Notes: 2.20.24

' ' indicates chars

" " indicates strings → adds NULL character

'4' = 52

& shows address of a variable

%x → hexadecimal

%p → pointer address

"0x" prefix to hexadecimal

∘ memory block that stores variables

* memory location *

printf("%s", &str[3]);

displays string str from str[3] to '\0'

char str[20];

str = "string"; NOT ALLOWED

can only define string at declaration

use strcpy(str, "string"); to get around this

char * means string

str[6] = 'f'; YOU CAN DO THIS

strcpy(); is doing this bit-by-bit

* you do not need a & to do scanf(); with char arrays *

but scanf(); ends at first whitespace

To get around this: fgets();
never use gets();
use fgets();

fgets(str, n, stdin);
stdin "standard input"
- default input device, keyboard
stdout - default output, monitor
stderror

⇒ includes the NULL character
fgets(storage, maximum entry, file pointer);
gets() has no limiters, which might overwrite parts of memory
looks for newline character as a delimeter
- ENTER
- this is INCLUDED in its input

str[strlen(str)-1] = '\0';
replaces '\n' with '\0'

files:
FILE *fp;
fp = fopen("filename

fscanf (fp, ...

File input/output:

```
#include <stdio.h> //enables use of FILE*
                     variables and functions

FILE* fp;     // file pointer
fp = fopen("file.txt", "r");
                        indicates "read mode"
  // upon success, fopen() returns a pointer to file
  // upon failure, fopen() returns NULL

if (fp == NULL) {
   printf("failure");
   return -1   // returns error

fscanf(fp, "%d %d", &var1, &var2);
// reads two integers from fp
   ...

fclose(fp); //closes file

fopen("file.txt", "w")
                       indicates "write mode"
fprintf(fp, "Hello\n")  // writes to file

feof(); // returns 1 when previous operation reached
         end of file

while (!feof(fp)) {   //example
  ...
```

Lecture Notes: 2.23.24

printf(" "); → # to protect
fgets(str, #, stdin);
puts(" "); → to the monitor, no choice
fputs(" ", stdout);
↳ gives choice
can output to file
or var pointer

puts(); adds '\n'
fputs(); does not add '\n'

* be careful with ENTER's and scanf()
  and fgets();

  %f, %d, %lf → use space as
  a delimiter
%c has no delimiter by definition

getchar(); gets the next char from the
  buffer (memory)

wget [RIGHT-CLICK] (pastes from clipboard)
  → downloads files

2D-Arrays
  reads by rows
  can declare in one set of { };
  or two sets of { { } { } };

  array of arrays

  x → columns
  y → rows

Lecture Notes: 2.28.24

2D Arrays:

int num [n] [m] = { 1, 2, ... n*m }

int num [n] [m] = { { 1, 2, ... m }
,
,
n times

int num [ ] [ ] // cannot do this
*just like you cannot do:
int num [ ]

int num [ ] [n] // you CAN do this
int num [ ] [n] [m] // you can do this too
* you need to specify the "base"
of the data structure
• only the length is variable

2D Size:

int size = sizeof (nums) / sizeof(nums[0][0]) / m;
needs to be multiplied by # of rows

nums [n] // 1D array of ints
nums // 2D array of ints

```
void display()
{ for (int i=0, i<size; i++) {
      for (int j=0; j<m; j++)
```

in declaration:

```
void display(int [][m], int);
```

you don't need to specify name

```
char states[2][15] = { "Indiana",
   {'I', 'o', 'w', 'a', '\0'} };
   //both works
```

in memory:

Indiana 0 0 0 0 0 0 0 0
Iowa 0 0 0 0 0 0 0 0 0 0 0

print strings by char:

```
for (int i=0; i<n; i++) {
   for (int j=0; j<strlen(states[i]); j++) {
```

Exam 1: Review 2.28.24

Initial Linux/Unix:

Commands:

ssh, pwd, ls, clear, users, w, wai, mkdir, cd, rmdir, touch, tree, mv, cp, rm

Characters:

\, ~, .., ., *, —

Flags:

-a, -l, -o, -lm

C Compiler: gcc : [name].c → ./a.out

C Preprocessors: # include <name>

stdio.h, math.h, string.h, ctype.h, stdlib.h, stdbool.h

Escape character: \

\n, \t, \r, \a, \" \\

Data Types: can all be short and long and const

int, float, double, char

Placeholders:

%d (or %i), %f, %lf, %g, %c, %s, %x, %p

%-#.#d

↳ must also account for potential negative sign

↳ left justifies (usually right justificat)

Control Structures

Conditionals/selection:

if, switch

Iteration/loops:

while, for

Comparison operators:

>, >=, <, <=, ==, !=

Logical Expressions:

!, &&, ||

Arithmatic:

+, -, *, /, %

* whitespace is meaningless in C

* {} block together code into one statement

* & points to address of character

Examples: all can be nested as well

```
if (condition) {
  ...
} else if (condition) {
  ...
} else {
  ...
}
```

```
switch (condition) {
  case 0:
    ...
    break;
  default:
    ...
    break;
```

```
for (initialization; condition; update) {
  ...
}
while (condition) {
  ...
}
```

<u>Assignment vs. Comparison:</u>
 = vs. == , very different!

<u>Boolean → integer:</u>
 true: 1
 false: 0

<u>integers → boolean:</u>
 non-zero: true
 zero : false

break; → leaves current structure
continue; → jumps to the end of the current iteration in the current structure

<u>Syntatic Sugars:</u>
 ++n → pre-increment
 n++ → post-increment
 ++, --, +=, -=, *=, /=, %=
 z = (x > y) ? x : y; ← ternary operator
 [var] → [var] is true
 ![var] → [var] is false

Typecasting: makes a data type look like
another data type to the compiler.
* does not change its value
ex. average = (float) int / int

Functions: default return is int
Prototypes:
[data type] [name] ([datatype] var, ...);
↳ can be "void" ↳ can be empty aka "void"
Definitions: ↳ can still return;
* same as above * {
...
}
Variables:
- local to the function
- passed and returned by value

Scope:
- vars declared inside { } only work inside { }
- cannot have two vars with the same name
within the same scope

Global Vars: don't use
instead #define VAR # ← NO semicolon

Flags: boolean value to tell loops to stop

Arrays: data structure
C-Arrays:
- static memory
- cannot hold different data types
- cannot have negative indicies
- start at 0
in declaration, [] is size of array
in expression, [] is index of array

Memory: int, float → 4 bytes
double, long int → 8 bytes
char → 1 byte short int → 2 bytes
1 byte = 8 bits = 2^{bits} numbers

__Declaration:__

int arr[5] = {1, 2, 3, 4, 5}; size is 5

int arr[10] = {1, 2, 3}; //rest are 0s

impossible to calculate size & so bad

int arr[] = {1, 2, ... n};

to find size: int size = sizeof(arr)/sizeof(int)

sizeof(); returns # of allocated bytes

__DON'T:__

int arr[3] = {1, 2, 3, 4, 5}; 4 & 5 lost in memory

int arr[5]; indicies filled with junk

__DO:__

int arr[n] = {}; array filled with 0s

__Sentinel Value:__ out-of-range indicator to notify program end of array has been reached

__Passing Arrays to Functions:__

- passed by reference
- actual values passed and modified
- must also pass size

__ex.__ prototype

void display(int [], int); "length" → size of array

definition

void display(int arr[], int size) { ... }

↳ address of first index

* use a temp var before passing values into an array

__Chars:__ → 1 byte, 256 bits, enough for all chars

%c prints char, or char associated with int

%d prints char's ASCII code

* ints and chars can be used interchangeably

a = 97, A = 65 ↳ chars are stored as ints

uses ' ', " " are for strings

Char Arrays: C Strings
- cannot be larger than declared array size
- size of # of chars + 1 bytes
- ends with an automatic sentinel '\0'
- size of char array does not need to be passed to functions (b/c of sentinel)

scanf() will only read first word of string!
∘ b/c space is its natural delimeter

* to get around problems like these there are multiple libraries of functions for char arrays
∘ cannot assign strings after declaration

* sentinel can also be manually placed and removed

File Pointers:
```
FILE *fp
fp = fopen();
```
* shows that var type is a pointer

fp is NULL automatically
∘ if ran, returns segmentation fault

if file is successfully connected, then fp =! NULL, otherwise still NULL

Formatted I/O:
output: printf, fprintf, sprintf
input: scanf, fscanf, sscanf

Non-formatted I/O:
output:
char: putchar, putc, fputc
string: puts, fputs
input:
char: getchar, getc, fgetc
string: fgets

Multi-Dimensional Arrays:

int arr[rows][columns]
stored and processed by rows
can be declared with:
int arr[m][n] = {1, 2, ... n*m};
or
int arr[m][n] = { {1, 2, ... n}
, {1, 2, ... n}
m times };
CANNOT declare as: int num[][];
NEED to define the "base": int nums[][m];
• only the height is variable
arr[m] → 1D array of ints
arr → 2D array of ints
arr[m][n] → int

Size:

n?
int size = sizeof(arr)/sizeof(arr[0][0])/m;
int size = sizeof(arr)/sizeof(arr[0]);

Passing to Functions:

declaration:

```
void display(int [][n], int);
```

"height"

definition:

```
void display(int arr[][n], int size) {
    for (int i=0; i<size; i++) {
        for (int j=0; j<n; j++) {
            printf("%4d ", arr[i][j]);
        }
        printf("\n");
    }
}
```

Calling:

```
display(arr, size);
```

↳ calculated implicitly

Useful Functions:

#include <stdio.h>

FILE* fopen(char * filename, char * mode);

↳ char[n] ↳ "r", "w", "a"

success: !NULL, failure: NULL

(int) fclose(FILE* stream);

success: 0, failure: EOF

(int) fprintf(FILE* stream, char * format, ...);

↳ "%d", definition

(int) printf(char * format, ...);

writes to stdout

(int) fscanf(FILE* stream, char * format, ...);

uses white space as a delimeter

\n [] points to data type

... must be an address (& for normal vars)

(int) scanf(char * format, ...);

reads by stdin

(int) sprintf(char * str, char * format, ...);

↳ char arr[n]

(int) sscanf(char * s, char * format, ...);

int fgetc(FILE* stream);

- returns char currently pointed by internal file position indicator of specified stream
- internal file position then advanced to next char
- returns EOF when at end of file

char * fgets(char * str, int num, FILE* stream);

- reads chars from stream until
  - num-1 chars read
  - '\n' detected
  - EOF reached
- stores in char array * str
- newline included in string copied
- null character '\0' automatically appended

int fputc (int character, FILE * stream);
- writes char to stream and advances position indicator
success: char written is returned, failure: EOF + error
int fputs (char * str, FILE * stream);
- writes char arr[] to stream
  o until it reaches '\0', which is not written
success: non-negative value returned, failure: EOF + error
int getchar (void);
- returns next char from stdin
int putchar (int character);
- writes char to stdout
int feof (FILE* stream);
- checks whether EOF of stream is set
- indicator is set by previous operation on stream
  o will not set until an operation tries to run on EOF
returns nonzero int if EOF is set
otherwise returns 0

include <math.h>
* double sin (double x);
sin, cos, tan, acos, asin, atan in normal range
double atan2 (double y, double x)
returns $\tan^{-1}(y/x)$ in radians from [-pi, pi]
double exp (double x)
exp, log, log10, ...
* double pow (double base, double exponent);
double sqrt (double x)
double abs (double x)

#include <string.h>
char * strcpy (char *destination, char * source);
- includes '\0'
char * strcat (char *destination, char * source);
- appends source to destination, starting at '\0' (overwrites '\0' and adds its own at end of string)
int strcmp (char * str1, char * str2);
- compares str1 to str2
returns 0 if equal, !0 if not equal
char * strchr (char * str, int character);
- locates first occurrence of character in str
- returns pointer to character
- works with '\0', returning pointer to end of string
char * strstr (char * str1, char * str2);
- returns pointer of first occurrence of str2 in str1, NULL if str2 is not in str1
size_t strlen (char * str);
- returns length of str
  - beginning of str to '\0'
    - does NOT include \0 in length

#include <ctype.h>
int [name] (int c);
- checks whether char is [name]
- returns !0 if true, 0 if false
- isalnum, isalpha, isblank, isdigit, islower, isupper, isspace
int tolower (int c);
returns lowercase c char
int toupper (int c);

Practice Exam 1 2.29.24

1.
```
void drawsquare(int n)
{
    for(int i=0; i<n; i++)
        printf("*");
    printf("\n");
    for(int i=0; i<(n-2); i++) {
        printf("*");
        for(int j=0; j<(n-2); j++)
            printf(" ");
        printf("*\n");
    }
    for(int i=0; i<n; i++)
        printf("*");
}
```

2.
```
bool is_perfect(int n)
{   int sum=0;
    for(int i=1; i<n; i++) {
        if(n%i==0)
            sum+=i;
    }
    if(n==sum) return true;
    else return false;
}
```

3.
```
void show_scramble(char str[])
{
  printf("%c", str[0]);
  for(int i=1; i<(strlen(str)-2); i++)
   printf("%c",str[strlen(str)-i]
}
```

4. 0 1 4 9 16
30

Old Fund Comp Exam Problems 3.1.24

5.
```
int findchar(char thestring[], char ch)
{
  for (int i=0; i<strlen(thestring)-1; i++) {
     if (ch == thestring[i]) return i;
  }
  return -1;
```

6.
```
void flip(int arr[], int size)
{
  for (int i=0; i<size; i++) {
    arr[0+i]==arr[size-i];
  }
}
```

lecture Notes: 3.4.24

Command-Line Arguments:

- interprets strings written after executable call
  - ./a.out startupit++
- works by passing arguments to int main()

common convension: ↱ come after executable

argc → argument count
argv → argument vector

* int main(int argc, char * argv[])
  ↳ # of arguments after executable

just typing executable makes argc = 1

char * means string, so:
char * argv[] → array of strings
- strings entered in command line
- argc total strings

argv(argc) → accesses string

char ** argv = char *argv[] = char argv[][]
but argv[][] cannot be used here because second dimension must be declared

$\underbrace{\text{./a.out "Mary Sue"}}_{\text{interpreted as 1 string}}$

*argc == 0 is impossible

Detecting return codes:

echo $?
* returns code of the very last command you did

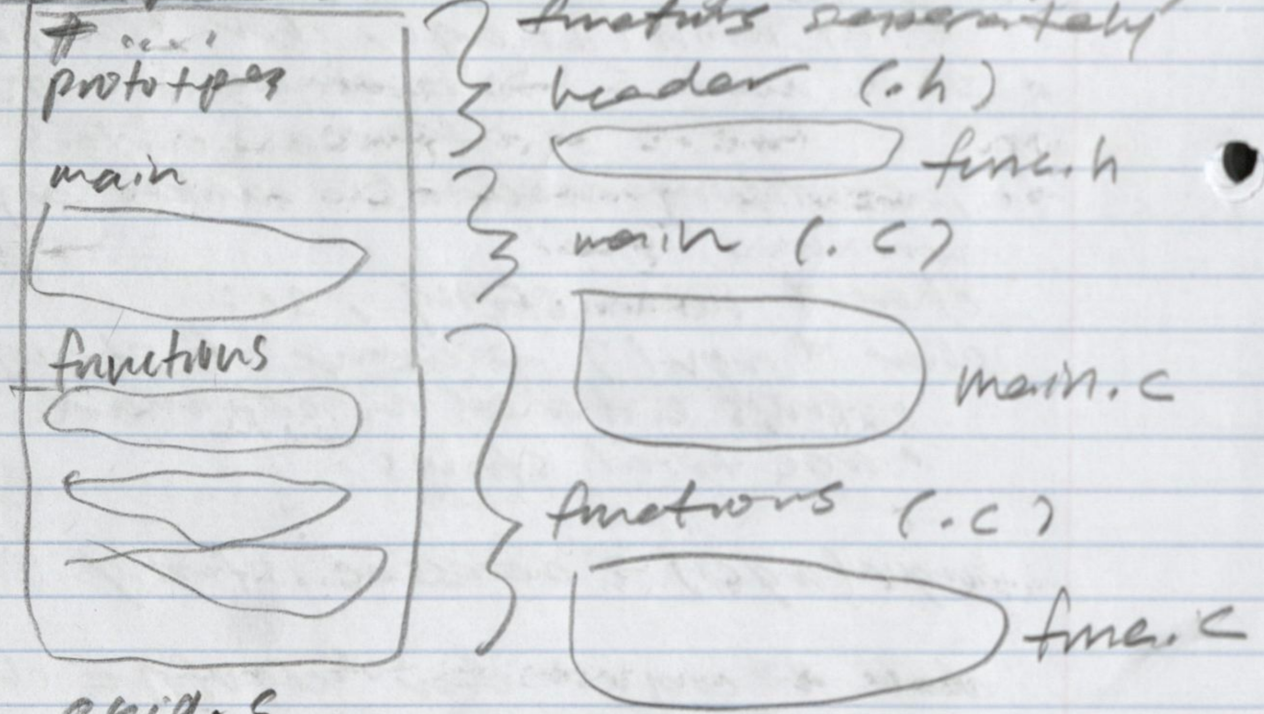
env list
→ lists environment commands

echo
→ returns stuff

return EXIT_SUCCESS;
= return 0;

Splitting Files:

splitting and compiling functions separately



orig.c

gcc orig.c -o runit
- translating C to machine code
- linking libraries

must #include "func.h"

→ object file

‹ compiling ›

main.c	gcc -c main.c → main.o
func.c	gcc -c func.c → func.o
func.h	gcc main.o func.o → a.out

‹ linking ›

Lecture Notes: 3.6.24

Make Files:

vi makefile, linux utility

rules

each rule consists of:

target: dependences

[TAB] command

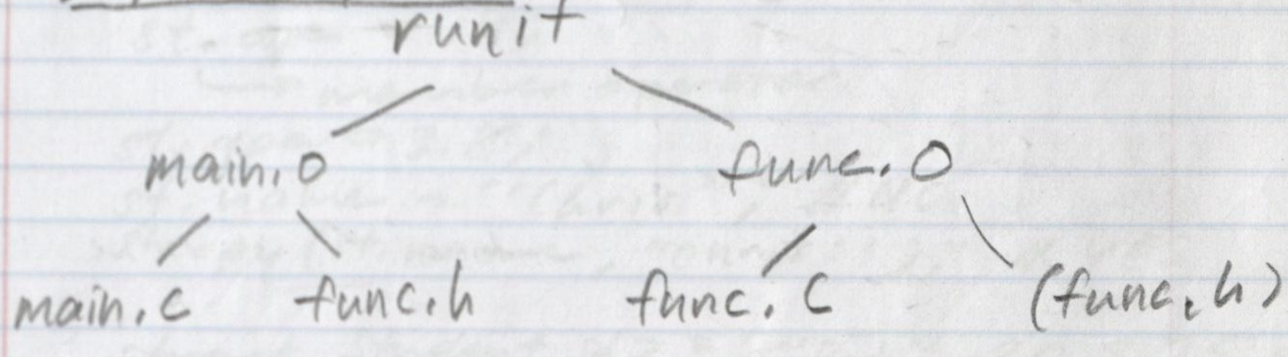
target2: dependences...

[TAB] command2

---

gcc -c main.c -o main.o
gcc -c func.c -o func.o } → compile
gcc main.o func.o -o runit → link

Dependency Tree:



make → autoroutes to top rule

make [target] → only does that rule

Lecture Notes 5

3/8/24

sprintf() → internal I/O
fprintf() → external I/O

* important for graphics

sscanf() → reads from a string
fscanf() → reads from a file

* it is impossible to represent .1 or .2
* you can only represent inverse powers of 2 precisely

any integer can be represented in binary
you cannot express every number btw 0-1 in binary

Lecture Notes: 3.18.24

arrays: only hold one data type
structs: can hold multiple data types

array → column of spreadsheet
struct → row of spreadsheet
* whole spreadsheet is an array of structs

```
struct Student {
  char name[20];
  int age;
  float gpa;
};
```
above main
(custom data type)

var st of type student

```
int main() {
   struct Student st;
```
→ attribute/member/subvariable
```
   st.age = 19;
```
→ member operator
```
   st.gpa = 3.8;
   st.name = "Chris"; // NO
   strcpy(st.name, "Chris"); // YES

   struct Student st2 = {"Pat", 20, 3.75};
   //can also assign at declaration

   struct Student st3;
   scanf("%s %d %f", st3.name, &st3.age ...
```

Passing to Functions:

```
void display(struct Student);

int main() {
   display(st1);
  display(st2);
  display(st3);
}

void display(struct Student s)
{ printf("%s %d %f", s.name, s.age, ...
```

typedef → creating an alias for a data type

```
typedef double real;
real x; → datatype of double

typedef struct Student {
   char name[20];
   int age;            both work
   float gpa;          (struct name and typedef)
} Student_s;
                → CAN be same as
                  struct name "Student"
typedef struct {
   char name[20];
   int age;            shortest way
   float gpa;          but CANNOT use
} Student;             struct name anymore
```

Struct Arrays:

```
int main() {
  Student theclass[3];
    └> struct Student typedef

theclass[0].age = 19
theclass[0].gpa = 3.8;
strcpy(theclass[0].name, "Chris");
// works for other 2 cases as well

theclass -> array of structs
theclass[1] -> struct
theclass[1].name -> char array / string
theclass[1].name[2] -> char

for (int i=0; i < 3; i++)
  display(theclass[i]);
    └> function from earlier
```

* we read CSV "comma delimited file" to read into struct arrays
* we have already used "space delimited files"

```
void display_all(Student[]);

int main() {
  display_all(theclass);


void display_all(Student all[]) {
  for (int i=0; i<3; i++)
    display(all[i]);
}
```

Lecture Notes:

3.20.24

* sometimes read comma delimited files instead of space delimited

parse → break a string into its elements (can just read by character and \n at commas)

OR

printf("%s\n", strtok(hobbies, ","));

↳ char array

* does not advance internal file pointer

printf("%s\n", strtok(NULL, ","));

NULL continues where it last left off

* so repeat with \n as final delimeter

char str[20];

strcpy(str, strtok(hobbies, ","))

// works, but cannot directly assign

char sentence[] = "hello, how you? cool; great! ok; bye \n";

printf("%s\n", strtok(sentence, " ,?:!;\n"));

inefficient

char delim[] = " ,?!!;\n";

fixes inefficiency

↳ add \n

Lecture Notes: 3.22.24

Make File Macros:
"variables" for makefiles
(it is actually a macro)

```
CMP = gcc            in header
$(CMP)               in body

MAIN = playlife      in header
$(MAIN).o            in body
$(MAIN).c

FUNC = lifefunc
$(FUNC).o or $(FUNC).c

EXEC = playlife
clean:
    rm *.o
    rm $(EXEC)
```

* you can also overwrite a macro from the command line

```
make EXEC=~~stuffthing~~
```

you can even assign all of the macros in the command line

Warnings:

```
gcc -Wall      // gives all warnings
```

Pointers:

int n; // declared
n = 15; // n's value is 15 ⇒ defined
printf("n is %d\n", n); // displays value

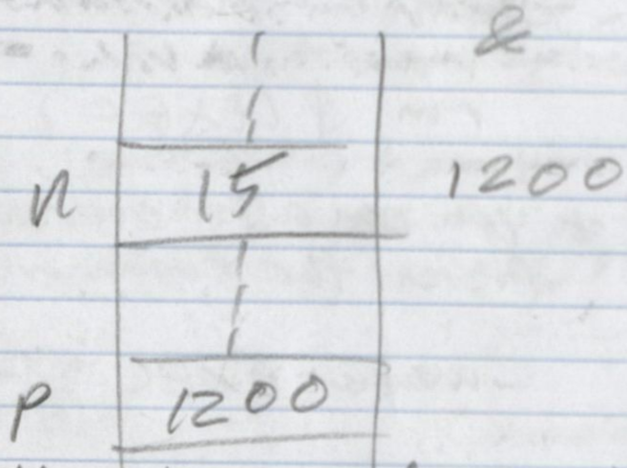
printf("n's address is %x\n", &n);
(or " —— // %p ——)
// displays address
// what n is in memory

int *p; // p is a pointer to int
// take on another var's address
// only *declared*, no content
// will get a *seg-fault* if used

NULL pointers return seg-faults!

int *p;
p = &n;

int* p, q;
is NOT a pointer



*&n will be different at each compilation but pointers will work regardless

Pointers have to point to the address of a *specific* data type (until void pointers)

17 int *p;
18 int * p;
19 int* p;

same to all; ramsey prefers 17, purists prefer 19, 18 is rare

Zybooks Notes: 3.24.25

- struct construct "declaration" just declares a new data type; no memory allocated
- variable definitions allocate memory for each object's member
- Accesses refer to an object's member's memory location
- assigning a variable of struct ty automatically assigns each corresponding data member
  struct1 = struct2

* a struct allows a function to return multiple values

```
TimeHrMin ConvHrMin(int totalTime) {
   TimeHrMin timeStruct;

   timeStruct.hourValue = totalTime / 60;
   timeStruct.minValue = totalTime % 60;

   return timeStruct;
}
```

reference operator: &
dereference operator: *

```
int someInt;
int *valPointer;
valPointer = &someInt;
*valPointer = 10; // updates someInt
```

NULL "nothing" pointer:
NULL defined as 0 because 0 is not valid memory address

Common Pointer Errors:

Syntax ERRORS

```
int someint = 5;
int *valpointer;
*valpointer = &someint; // ERROR, int ≠ int*

int* valpointer1, valpointer2; // need * before each
                               //   pointer
valpointer1 = NULL;
valpointer2 = NULL; // ERROR, int ≠ NULL
```

Runtime ERRORS:

```
int *valpointer
*valpointer = 4; // ERROR, dereferencing
                 //   unknown address

int *valpointer = NULL;
*valpointer = NULL; // ERROR, cannot
                    //   dereference a NULL pointer
```

Lecture Notes: 3.25.24

Graphics: X11 library
- old but reliable, run through gfx.h and gfx.o
Windows Machine: Putty with X-forwarding also launch Xming

Create a Saved Session:
SSH → X11 → Enable X11 forwarding
test with xeyes

copy graphics directory to link files
#include "gfx.h"

gfx_open(int width, int height ... → in pixels
function uses and declarations in gfx.h file

compile with: (or write a Makefile)
gcc ex63.c gfx.o -lX11
↳ "library"

Pause: while(1) { };

gfx_color (R, G, B);
gfx_flush(); → flushes stuff out of memory
// sometimes fixes issues on old PCs

Animation:
while (1) {
gfx_line (100+dx, 100, 300+dx, 200);
animation component of

gfx_wait(); pauses screen and waits for an event (key press etc.)

<u>Buffered Programming</u>: Press a key and then press ENTER for the program to respond

<u>Event Programming</u>: Program reacts immediately after a key press

*X11 reacts to <u>just</u> key press and mouse click

```
char c;
c = gfx_wait();
if (c == 'q') break;
```

↳ or any argument

<u>WASD controls</u>:

```
char c;
while (1) {
    gfx_clear();
    c = gfx_wait();
    if (c = 'a') dx--;
    if (c = 'd') dx++;
}
```

* gfx_color changes whatever comes next

```
int gfx_xpos();  }
int gfx_ypos();  }  if (c = 1)
```

gives the position of the mouse AFTER a click (event)

*integers 1, 2, 3 are reserved for mouse clicks

More Pointers:

```
int arr[5] = {34, 62, 31, 44, 77};
int *p;
p = &arr[0];
  //OR          ——> do the same thing
p = arr;
```

p++ moves the pointer forward to the next number.
(arrays are contingous in memory)

<u>Lecture Notes</u> 3.27.24

```
c = gfx_wait(); //can return char or int
//basically just returns an int
```

examples in public directory

#include "gfx.h" <u>NOT</u> <gfx.h>

top-left is (0,0)
down the window <u>increases</u> y
across the window increases x

} in pixels

Animation:

```
int main()
{ char c; int dx = 0;
  gfx_open(800, 600, "title");

  while(1) {                100+dx
    gfx_clear();
    gfx_circle(100, 100, 20);

    c = gfx_wait();

    if (c == 'q') break;
    gfx_flush();
    dx++;
}
```

* problem: gfx_wait()
- prevents the loop from iterating
- but is needed for interaction

* solution: gfx_event_waiting()

```
if (gfx_event_waiting()) {
   c = gfx_wait();
}
```

also utalize usleep( );
#include <unistd.h>

can also dx += 6;

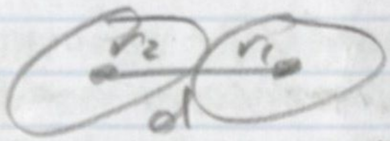
Velocity Vectors:

```
int xc=100, yc=100

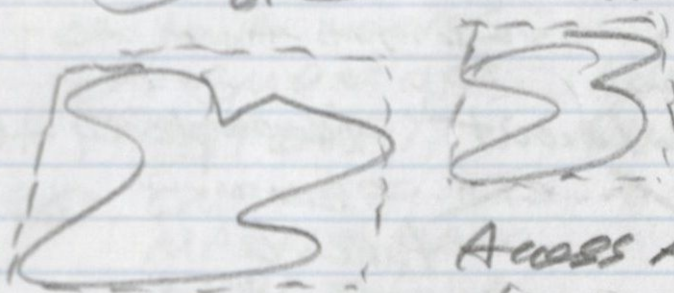
int dx=1;     * can change to floats
int dy=1;       to make even slower

while(1) {
  ...
  xc+=dx;
  yc+=dy;
}
```

Collision Detection:



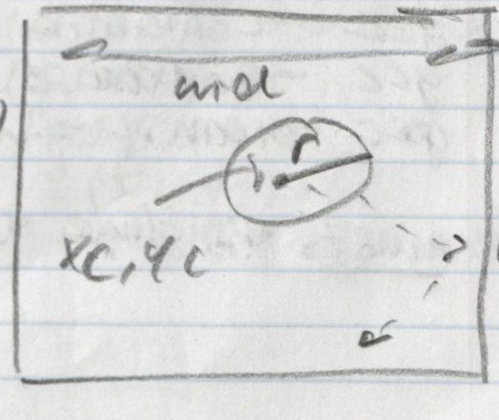
if $(d < (r_1 + r_2))$
collision



AABB

Access Aligned Bounding Box
↳ parallel to screen

```
int rad=20;
int wid=600, ht=500;
while(1) {
  ...
  if((xc+radius)>=wid)
  {
    dx=-dx
```



dx=-dx
dy=dy

Exam 2 Review: 4.4.24

Command-line Arguments: argc/argv

int main(int argc, char *argv[]) { }

argc → # of arguments after excecutable

argv[] → array of char arrays with CL info

ex: argv[0] = "./a.out"   argc = 1

./a.out startup.text   argc = 2

argv[0]   argv[1] ...

note: char **argv = char *argv[] = char argv[][]

argv[][] cannot be used because "base" is undefined

argc == 0 is impossible

./a.out "Mary Sue" interpreted as 1 string

Detecting Return Codes:

echo → returns stuff

echo $? → returns code of last command

env list → lists environment commands

Splitting Files:

```
                 ./a.out        (dependency tree)
               /         \
        main.o               func.o
       /      \             /      \
  main.c    func.h     func.c     func.h
```

gcc -c main.c → main.o

gcc -c func.c → func.o

gcc main.o func.o → a.out

must #include "func.h" to main.c and func.c

↳ file path to header file

func.h → function prototypes, preprocessor directives (#includes), global variables and macros, and data type definitions
func.c → function definitions
main.c → main function

-c means to just compile, while no flag defaults to both compile and link
-o can also be used to name the files directly (good practice)

Makefiles: vi Makefile, linux utility
organized in rules:
target: dependencies
[TAB] command
:

* can call any specific target: make [target]
* or can easily call top target: make
can have macros:
ex. CMP=gcc //declaration
$(CMP) //use

ex.
```
CMP=gcc
MAIN=main
FUNC=func
EXEC=a.out
$(EXEC): $(FUNC).o $(MAIN).o
	$(CMP) $(FUNC).o $(MAIN).o -o $(EXEC)
$(FUNC).o: $(FUNC).c $(FUNC).h
	$(CMP) -c $(FUNC).c -o $(FUNC).o
$(MAIN).o: $(MAIN).c $(FUNC).h
	$(CMP) -c $(MAIN).c -o $(MAIN).o
clean:
	rm *.o
	rm $(EXEC)
```

Standard I/O: stdin, stdout
  scanf("format string", &var1, ...);
  printf("format string", var1, ...);
File I/O: FILE *fp
  fscanf(fp, "format string", &var1, ...);
  fprintf(fp, "format string", var1, ...);
Buffer I/O: char *string
  sscanf(string, "format string", &var1, ...);
  sprintf(string, "format string", var1, ...);

Comparison: == and != should only be used
with ints and char's, never float's or double's
- any integer can be represented in binary
- not every float can be represented in binary

Placeholder: %*s allows *scanf to skip
unneeded strings in buffered or file I/O

Structs: self defined datatype that can hold
various data types as "members"
Definition: 3 ways

```
struct Student {
   ⋮
}; // datatype "struct Student
typedef struct Student {
   ⋮
} Student_s; // datatype "Student_s" OR
                          "struct Student"
typedef struct {
   ⋮
} Student; // datatype "Student"
```

member operator (.): allows you to access
a member of a struct

typedef: creates an alias for a datatype
Declaration: MUST declare variables after defining the datatype
[datatype] [name];
- can assign values at declaration:
  Student = {"Adam", "Smith", 21, 3.82};
  * only works at declaration
- can also read-in values from stdin or file (one member/attribute at a time)
- can also declare struct arrays

NOTE:
- structs are passed to functions by value
- you can return a struct object from a function
- they are passed and returned the same as other datatypes, but not the same as arrays (unless it is a struct array)
- can assign one struct to another and all members copy over
  struct1 = struct2;

Ex- Student nd_students[8000];
nd_students → struct array
nd_students[2] → struct
nd_students[2].name → string (char array)
nd_students[2].name[3] → char

NON-space delimeted files:
Parse: break a string into its elements
strtok() → string.h function that tokenizes (parses) a string
- returns a pointer to the beginning of the next token that is picked up and NULL when it is done
- does not advance internal file pointer

- on first use you pass the address of where the string starts and it returns the token (substring until first delimeter is reached)
- on subsequent use you pass NULL for the first argument and it continues where it left off (so use '\n' as final delimeter)

* also stops if it finds '\0'

Ex:

```
typedef struct {
    char dorm[20];
    char mascot[20];
    int capacity;
} Dorm;
    ⋮
int read_dorms(FILE *fp, Dorm all[]) {
    int n=0;
    char line[60];                    can be ", \n ..."
    while(1){                           delimeter(s)
        fgets(line, 60, fp);
        if (feof(fp)) break;
        strcpy(all[n].dorm, strtok(line, ","));
        strcpy(all[n].mascot, strtok(NULL, ","));
        all[n].capacity = atoi(strtok(NULL, "\n"));
        n++;
    } return n; }
```

where its stored in memory

Pointers: stores a variables address

```
int n = 7; // n's value is 7 and stored in n
int *p;
p = &num; // n's address is &n and stored in p
printf("%x %p", p); //displays address of n
```

* must point to a specific datatype (for now address and pointer datatypes must be the same)

-Wall //flag that generates <u>all</u> warnings

<u>reference operator</u>: *
<u>dereference operator</u>: &
* can be placed anywhere:
int* p; <=> int * p; <=> int *p;

<u>Note</u>: &arr[0] <=> arr

<u>Graphics / Event Programming</u>: X11 library
<u>events</u>: unbuffered (computer reacts right away)
<u>X11</u>: requires gfx.h and gfx.o
must #include "gfx.h"
compile with -lX11 flag
uses many QOL functions
* down the window <u>increases</u> y
* across the window increases x

<u>Animation / Collision Detection</u>:
<u>Ex</u>:

```
int main() {
    gfx_open(wid, ht, "name");
    int xc, yc, dx, dy; //center and velocity vector
    char c;       //all equal to values
    while(1) {
        gfx_clear();
        gfx_circle(xc, yc, RAD);
        c=gfx_wait();
        if(c==1) {
            xc=gfx_xpos();
            yc=gfx_ypos(); }
        if ((xc+RAD)>=wid) dx=-dx;
        if ((xc-RAD)<=0) dx=-dx;
        if ((yc+RAD)>=ht) dy=-dy;
        if((yc-RAD)<=0) dy=-dy;
        xc+=dx; yc+=dy;
        usleep(1000); if(c=='q') break; } }
```

gfx functions

collision detection

updates velocity vector

Random Numbers: rand(), in #include <stdlib.h>

- returns an integer b/w 0 and INT_MAX with normal distribution probability
- by default will use the default seed; will get the same sequence every time
- Use the srand() function to choose the seed
  - use time() as the input to randomize the seed (time() returns the number of seconds since 1/1/1970)
    requires #include <time.h>

```
srand(time());
```

- use the modulo operator (%) to control the range of random numbers

```
rand() % (upper_bound - lower_bound + 1) + lower_bound
```

Useful Functions:

```
#include "gfx.h"
  void gfx_open(int width, int height, char *title);
  void gfx_line(int x1, int y1, int x2, int y2);
  void gfx_circle(int xc, int yc, int r);
  void gfx_color(int R, int G, int B);
  void gfx_clear();
  void gfx_clear_color(int R, int G, int B);
  char gfx_wait();
  int gfx_event_waiting();
  int gfx_xpos;
  int gfx_ypos;
  void gfx_flush; ?
  void gfx_text(int x, int y, char *text);
```

```
#include <ctype.h>
    int (char c)
    isalnum, isalpha, isblank, isdigit, islower,
    ispunct, isspace, isupper, tolower, toupper
#include <math.h>
   double(double)
     cos, sin, tan, acos, asin, atan, atan2,
     exp, log, pow, sqrt, abs, INFINITY
#include <stdlib.h>
   int atoi(char *str), int rand(), srand(seed)
#include <string.h>
   char * strcpy (char *destination, char *source)
   int  strcmp(char *str1, char *str2)
      0 = equal , !0 = not equal
#include <time.h>
   time()
#include <stdio.h>
   FILE *fopen (char *filename, char *mode);
       mode = "r", "w", "a"
   fclose (FILE *fp);
   FILE I/O
   STANDARD I/O     } in notes
   BUFFER I/O
   Character I/O
     int fgetc(FILE *fp);
     char *fgets(char *str, int n, FILE *fp);
     int getchar();
     int feof (FILE *fp);
       default -> 0 , EOF -> !0
       if (feof(fp)); // EOF
       while(!feof(fp)); //default

#include <unistd.h>
   usleep(int microseconds);
```

Lecture Notes: 4.10.24

Pointers: store a vars address
*every var is stored in memory somewhere

```
int *p; //declared, seg-fault on its own
p=&n; //defined

p=&arr[0]; } // same value
p=arr;     }
```

Dereferencing: access memory from the pointer

```
int *p;  // * means symbol to recognize pointer
p=&num;
printf("%d %d", num, *p); //prints same thing
                          <=> * is dereferencing
```

* Pointers give more control over memory
  - int cannot be moved in mem
  - int* can be moved

Ex.

```
int num = 43;
int *p = &num;
*p = 58; // changes value of num
printf("%d", num); //prints 58
```

Arrays:

```
int arr[] = {1, 2, 3, 4, 5};
int *p;
p = arr;
printf("%d", p[2]); // prints 3
printf("%d", *p); // prints 1
p++; // moves the pointer the correct
        amount of bytes of its data type
printf("%d", *p); // prints 2
```

p += 2; // advances pointer 2 elements down
printf("%d", *p); // prints 4
// out of bounds of original array
p += 10; // advances pointer to garbage
printf("%d", *p); // gives seg-fault or garbage

int arr[5] = {1, 2, 3, 4, 5};
int * p;
p = arr ⟺ int *p = arr

*p = arr is **NOT** possible but
int *p = arr is possible

for (int i=0; i < 5; i++) {
  printf ("%d ", *p);
  p++
} ← advances the pointer
// can also indicate the array or the pointer

p = arr; // **MUST** reset the pointer to
print values again

display(int [], int);

display (arr, 5); // do same thing
display (p, 5);

for(int i=0; i<5; i++) {
  *p += 3;
  p++;
} // adds 3 to every value of arr
without touching arr

display(p, 5); // does **NOT** work, need
to reset pointer

→ any regular variable

<u>Functions</u>: ints are normally passed by VALUE

can pass a pointer to an int to pass any data type by REFERENCE

<u>ex.</u>

```
void func(int);
void func_p(int*);

int main() {
  int n = 23;
  printf("%d", n);
  func(n);                 // does NOT change
  printf("%d", n);

  func_p(&n);  // does change
               → send address of n
}
void func(int n) { n=55; }
void func_p(int *p) {
  *p = 77;
}
```

* allows multiple "outputs" from a function

Lecture Notes: 4.12.24

* can also pass structs by reference with pointers

Swap 2 Variables = in a function

```
swap(&a, &b);
void swap(int *p1, int *p2) {
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
} // no return needed
```

prototype would be swap(int *, int *);

Sum and Difference

```
compute(a, b, &sum, &diff);
void compute(int a, int b, int *ps, int *pd)
{   *ps = a+b;
    *pd = a-b;
}
```

*returns of these kinds of functions are usually error codes

Pointers to Strings:

```
char str[] = "notre dame";
char *p;
p = str;
```

(str represents an address, must manually convert to a pointer)

```
printf("%s\n", str);
printf("%s\n", p);
```

(do the same thing)

%s takes an address and prints chars until '\0'

```
printf("%c\n", *p); // prints 'n'
```

b/c *p pointing at first char

CANNOT do str++ to iterate
CAN do p++ to iterate

```
p = p + 3;
printf("%c\n", *p); // prints 'r'
printf("%s\n", p); // prints "re dame"

p = p + 3;
*p = 'f';
printf("%s\n", str); // "notre fame"
printf("%s\n", p); // "fame"

(*p)++;
printf("%s\n", str); // "notre game"
```

parenthesis due to operator precedence

CAN:
pass array to array
pass array to pointer
pass pointer to array
pass pointer to pointer

* and they all work the same
* for now char [] and char * are the same

Zybooks Assignment #7: 4.14.24

```
#include <stdlib.h>
malloc(bytes); //returns pointer to newly
allocated memory of size "bytes"
```

common use: malloc(sizeof(type));
returns: (type*)malloc(sizeof(type));
↳ typecasting

free(pointer); //deallocates memory
allocated by malloc; inverse of malloc function

& often used with structs
Member Access Operator: -> → do the same thing
structPtr->memberName //(*structPtr).memberName

bytes = $\frac{bits}{8}$

Lecture Notes:

4.15.24

Pointers and Arrays:

when printing and passing to functions, pointers and arrays behave the same

BUT pointers and arrays are NOT the same

```
char str1[] = "notre dame";
printf("%s\n", str1);
                        // both do the
char *p1 = "champions";   same thing
printf("%s\n", p1);

char str2[20];
str2 = "computer"; // CANNOT do this
strcpy(str2, "computer"); // solution !

char *p2;
p2 = "program"; // CAN do this
printf("%s\n", p2);
```

- str2 cannot equal an address, it is fixed
- p2 can equal a address, and stores the location of "p"

Memory Leaks: assigning memory to a pointer and then reassigning a pointer thus losing the location of the content

```
char *p3;
strcpy(p3, "memory"); // CANNOT do this
printf("%s\n", p3);
```

- when declaring a char [], you declare memory for the string
- when declaring a char *, you do NOT declare memory for strings

strcpy(); requires memory while
p1 = " " does not; just sends address of first char

strcpy(p3, "memory"); null seg fault

char *p4;
char temp[20];
p4 = temp;
strcpy(p4, "good stuff");
// This WOULD work, but bad practice and static... instead

<u>Malloc Function</u>: dynamically allocate memory
p4 = malloc(20*sizeof(int));
// allocates 20 bytes
strcpy(p4, "good"); // works now

C typecasts automatically, while
C++ requires typecasting of void pointer

<u>Stack vs. Heap</u>:
<u>stack</u> mem → static memory, variables
<u>heap</u> mem → dynamic allocated memory

- anything declared on the stack releases its memory when complete → no memory leaks
- heap does not do this → memory leaks
free(p4); // frees memory and returns pointer to initial NULL state

Lecture Notes: 4.17.24

#define _GNU_SOURCE
strfry(str); //scrambles a string

Pointers to Structs:

```
typedef struct {
  float w;
  char name[20];
} Square;

Square sq1;
Square *p;
p = &sq1;

*p.name; //does not work b/c
         of C Operator Precedence

(*p).name; //must use parentheses
```

Syntactic Sugar: p->name // great

Pointers to Struct Arrays:

```
Square shapes[];
Square *p;
p = shapes //no & since array
Square *p=shapes //also works

for(int i=0; i<4; i++) {
  printf("", p->name, p->w);
  p++; // no indices, just advance
         the pointer
}
p=shapes; //MUST reset pointer after
```

Void Pointers:

```
int n = 5;
int double x = 3.75;
int float y = 36.13;

int *p1;        }  redundant...
double *p2;     }  need a "universal
float *p3;      }  remote control"
```

Want a single pointer that can access multiple vars:

```
void *p; // cannot dereference this yet

//compiler does not know how much memory
  to allocate
```

Must Typecast:

```
p = &n;              typecasts to int pointer
printf("", *(int *)p);
           dereferences new int pointer
```

the (*'s) have different purposes

```
p = &n;
printf("", *(int *)p);
p = &x;
printf("", *(float *)p);
p = &y;
printf("", *(double *)p);
```

Crossword Planning 3

j=0 1 2 3 4 5 6 7 i=0

BASEBALL yPos-k

1 K

2 T xPos, yPos

3 S

4 Y i=1

k: 0 1 2 3 4

SMELL i=2

board->list[i][j] == board->list[word][k]

Lecture Notes: 4.22.24

- everytime a function is called, its contents are sent to the stack
  - and removed after it is ran

Recursive Functions: functions that call themselves

Stack: memory storage like a "stack of plates" → can only retrive from the top of storage
  LIFO → last in first out
  FIFO → first in first out

* functions when stored are LIFO
* recursive functions act similarly

- infinite recursion causes stack overflow
  - seg fault

Modify Parameter:

```
void func(int n) {
    printf("%d\n", n);
    funct(n+1);
}
```

copy of ENTIRE function is stored in stack, and memory back-tracks as soon as it can

- functions can work prior or after the recursive call ex.

```
void func(int n) {
  if (n > 3) return;  // base case
  printf("n is %d\n", n);
  func(n+1);
  printf("n is now %d\n", n);
}
```

output:

n is 1
n is 2
n is 3
n is now 3
n is now 2
n is now 1

because of how memory in the stack functions (LIFO)

Factorials:

iterative (loop) definition:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$

recursive definition

$n! = n \cdot (n-1)!$

$1! = 1$

$0! = 1$

$4! = 4 \cdot 3!$

$3 \cdot 2!$

$2 \cdot 1!$

1

* recursion can waste a lot of time

GCF:

```
int gcd(int a, int b) {
  if (b==0)
    return a;
  else
    return gcd(b, a %b);
}
```

OR

```
return (b==0) ? a : gcd(b, a%b);
```

```
int fact(int n) {
int f;

if(n==1 || n==0)
  f =1;
else
  f = n * fact(n-1);

return f; //NECESSARY
}
```

as stack returns up its backup
in memory these returns are needed

```
return (n==1 || n==0) ? 1 : n * fact(n-1);
```

1. what am i doing?
2. what is modify parameter?
3. what is base case?

Lecture Notes: 4.26.24

```
void recurse() {
 // stuff before
  recurse();
 // stuff after
}
```

called on the way to base case

called during backtracking

- WILL have to rewrite loops as recursive functions on exams
- examples in public directory

Starting C++:

```
#include <iostream>
using namespace std;
int main {
    cout << "hello world" << endl;
}
```

- compile with: g++ "name".cpp
- C++ is back compatible!!
- Use renamed standard libraries!

Either add namespace, OR:

```
std::cout << "hi" << std::endl;
```

scope operator

- in C, I/O are functions
- in C++, I/O is done through streams, << = output stream operator

file extensions: [name].cpp

I/O:

```
cout << "enter a variable: ";
cin >> n; // input/output
             operators
cout << "you entered " << n << endl;

cout << "enter two values: ";
cin >> n >> m;
cout << "entered" << n << " and " << m;
```

- no placeholders!
- no formatting!

directly - most things done with classes

C++ has classes!

classes are like structs that can also hold functions

```
#include <string>   // C++ library
#include <cstring>  // C library

string str;
str = "notre dame";
```

this works now

b/c str is now <u>not</u> an address

- passed by VALUE

Lecture Notes: C++ 4.29
input: cin >> var
output: cout << var
no formatting needed!

C++ is backwards compatable with C
cstrings and C++ strings are very different

```
#include <cstring>              #include <string>
char str[30];                   string str
strcpy(line, "what");           str = "notre dame";
```

can cout << str << endl both!

```
strcat(line, " cont");          str = str + " cont";
```

Length: both the same
 strlen(cstring);
 string.length() // since C++ strings are classes!

Functions in classes are called methods

Size:
sizeof(cstring); // smaller
cppstring.size(); // larger

C++ strings do NOT have '\0' attached

Check parameters of C functions when using C++ strings
 cppstring.c_str();

C++ Passing by Reference:

```
void func(int &);
int main() {
  int n = 5
  cout << n << endl;
  func(n);
  cout << n << endl;
}
void func(int &x)
{ x=57; }
```

Outputs:
5
57

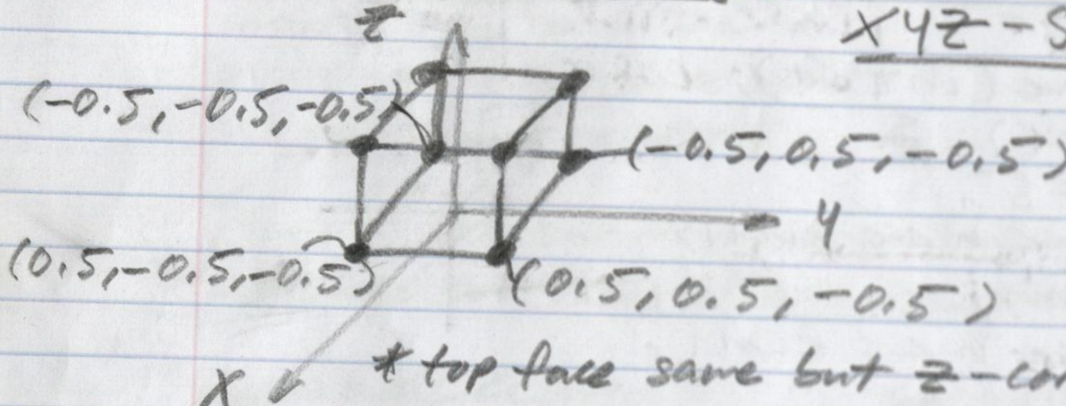
//done inside of methods?
• hides details...
• no way to tell without looking at functions

* functions can have the same name in cpp "function overloading"

• compilers tell which function to call based on data types or arguments

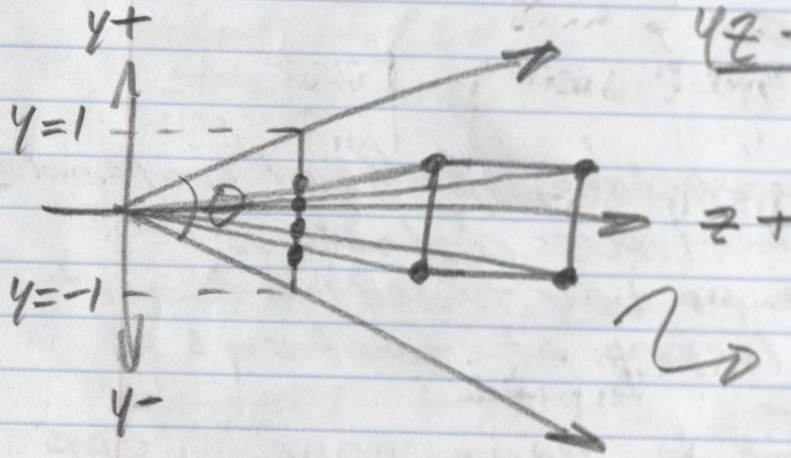
Final Project Planning:

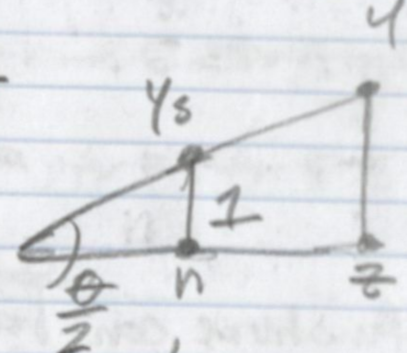
XYZ-Space:



* top face same but z-cords pos

YZ-Space:





$$\Rightarrow \tan\left(\frac{\theta}{2}\right) = \frac{1}{n} \Rightarrow n = \frac{1}{\tan\left(\frac{\theta}{2}\right)}$$

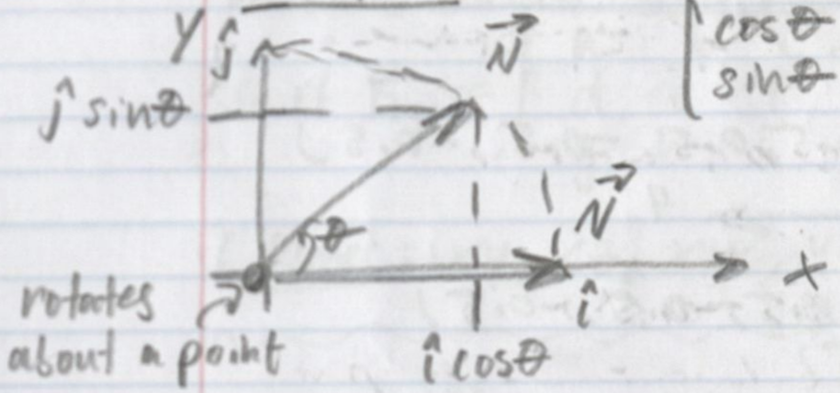
$$\frac{y_s}{n} = \frac{y}{z} \Rightarrow y_s = \frac{yn}{z} = \frac{y\left(\frac{1}{\tan\left(\frac{\theta}{2}\right)}\right)}{z}$$

$$\Rightarrow y_s = \frac{y}{z\tan\left(\frac{\theta}{2}\right)}$$

Same for X:

$$\Rightarrow x_s = \frac{x}{z\tan\left(\frac{\theta}{2}\right)}$$

Rotations:



$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = R$$

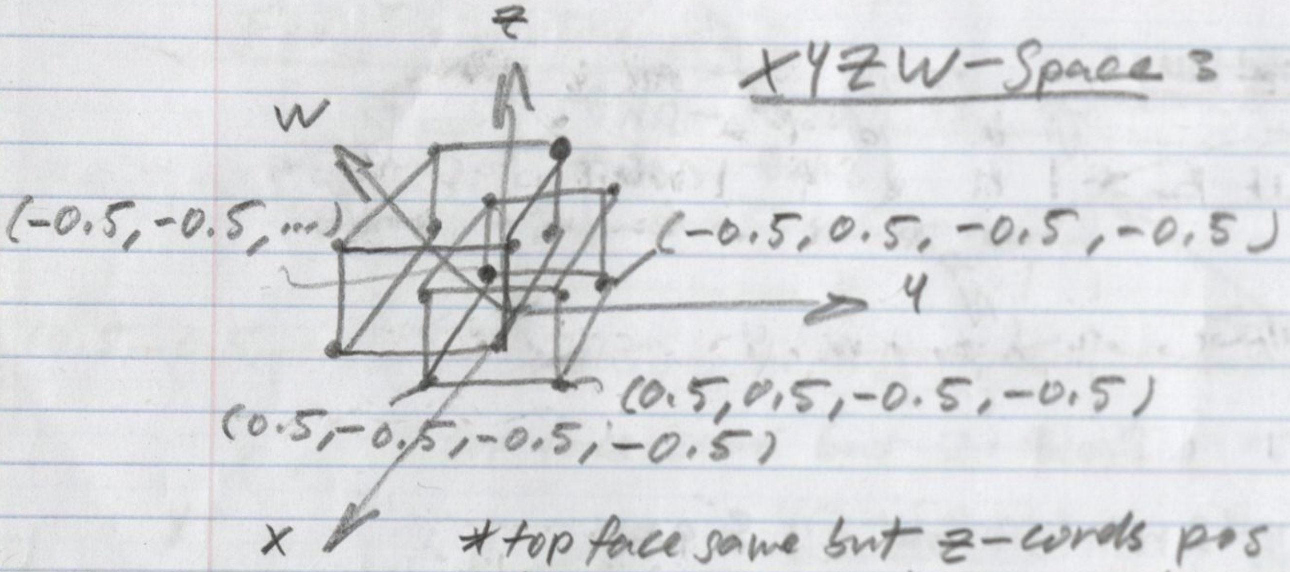
in $\mathbb{R}^3$: rotates about all 3 axis

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}}_{\text{about } x\text{-axis}}, \underbrace{\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}}_{\text{about } y\text{-axis}}, \underbrace{\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{about } z\text{-axis}}$$

$R_x(\gamma)$ "roll" $\quad$ $R_y(\beta)$ "pitch" $\quad$ $R_z(\alpha)$ "yaw"

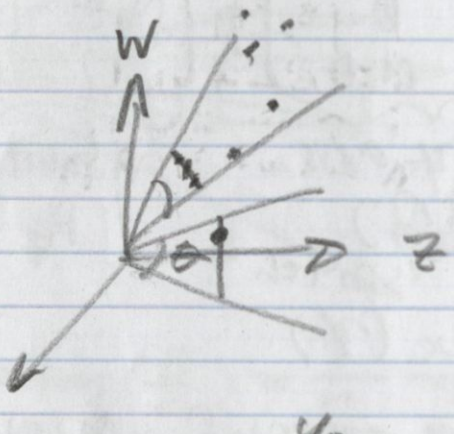
$$R = R_z(\alpha) R_y(\beta) R_x(\gamma)$$

$$\Rightarrow \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix}$$



XYZW-Space 3

*top face same but z-cords pos
*left cube same but w-cords pos



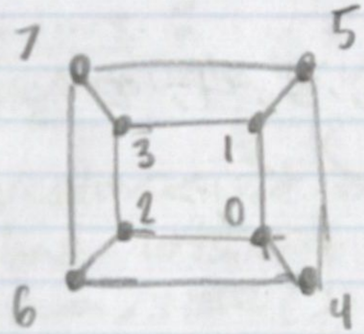
$$y_1 = \frac{y_0}{z\tan(\frac{\theta}{2})}, \quad x_1 = \frac{x_0}{z\tan(\frac{\theta}{2})}$$

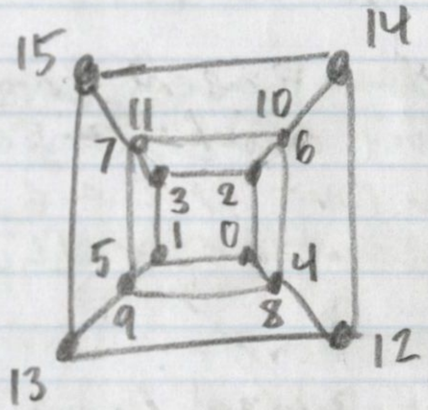
$$y_s = \frac{y_1}{w\tan(\frac{\theta}{2})}, \quad x_s = \frac{x_1}{w\tan(\frac{\theta}{2})}$$

Rotations: in $\mathbb{R}^4$, rotate about all 6 planes

so if $R_3 = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$ where a-i are fixed on other page

6 Planes: XY, XZ, XW, YZ, YW, ZW





12 edges:

0 → 1   3 → 1
0 → 2   3 → 2

0 → 4   3 → 7
2 → 6   1 → 5

4 → 5   7 → 5
4 → 6   7 → 6

Lecture Notes: 5.1.24

A void func(int); // pass by value
B void func(int *); // pass by reference in C
C void func(int &); // pass by reference in Cpp

A and C cannot be function overloaded as it takes the same call
- no way to overcome ambiguity

Index-Based Loop:

```
int arr[] = {0, 1, 2, 3, 4, 5};
for (int i; i < 6; i++) {
  cout << arr[i] << " ";
}
```

Range-Based Loop:

n & arr must be same data type

```
for (int n : arr) {
  cout << n << " ";
}
```

automatically navigates through every element of the set arr
- lose control

```
string college = "Notre Dame";
for (int i=0; i < college.length(); i++)
  cout << college[i] << " ";

for (char w : college)
  cout << w << " ";
```

Static Vars =

static int hi;
- static variables retains its scope after leaving its function

in theory would never need to pass by reference either

Vectors =

```
#include <vector> //dynamic array

vector<int> vec = {0, 1, 2, 3, 4, 5};
```

- has lots of useful members

```
vec.size(); // contains info about its size
```

- can be indexed

```
for (int i=0; i<vec.size(); i++)
    cout << vec[i] << " ";
    cout << vec.at(i) << " ";
```

same thing

2D =

```
vector<vector<int>> vec2d =
        {{0, 1, 2},
         {3, 4, 5}};

for (int i=0; i<vec2d.size(); i++) {
  for (int j=0; j<vec2d[i].size(); j++
    cout << vec2d[i][j] << " ";
}
```

Final Exam Review: 5.6.24

Pointers: variable that stores another variable's address
- int *p; //declare a pointer to int
- p=&n; //assign n's address to p
- "p points to n"

Dereferencing: access memory content from pointer
- *p=58; // changes value of n to 58 (write mode)
- printf("%d", *p); // prints 58 (read mode)

Pointing to Arrays:
- int arr[ ] = {0, 1, 2, 3, 4, 5}; int *p;
- arr → address of arr[0]
- p=arr; // p points to arr[0]
- p=&arr[0]; // equivalent statement
- *p; // access '0' in arr[0]
- Can advance pointer with (++ -- + -):
  - p++; //advances the pointer by 1 block
  - block → amount of memory equal to the data type p was declared to point to
  - since arrays are contiguous in memory this allows p to access any element of arr
  - also means pointer can leave array and begin to point to garbage
    - must reset pointer to prevent this
      p=arr;

Note. int *p=arr; is possible
*p=arr; is not possible
different operations

Note. display(int [ ]);
display(arr); and display(p);
do the same thing (both addresses)

Pointers and Functions: can use pointers to pass any data type by reference

- by using a pointer you are sending an address to a data-type (like arrays), so the function knows where the variable is stored in memory

```
void func(int *); // declaration
int main() {
   func(&n); // calling in main
```

Ex.
```
void swap (int *, int *);
swap (&a, &b);
void swap (int *p1, int *p2) {
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp
} // no return needed
```

Note: can now have multiple "returns" from functions

Pointing to Chars: basically a C string

- ```
  char str[] = "notre dame";
  char *p;
  p = str;
  printf("%s", str); // equivalent
  printf("%s", p);   // statements
  ```
- pointers give more flexibility
  cannot do str++; to iterate
  can do p++; to iterate

Note.
```
(*p)++; // advances char at location p
*p++;  // advances char at location p+1
       // and accesses its memory
```
- check C operator precedence
```

Pointers VS. Arrays:

- Passing to function is very similar
  - can pass an array and take it in as an array
  - can pass an array and take it in as a pointer
  - can pass a pointer and take it in as an array
  - can pass a pointer and take it as a pointer
- But an array allocates (static) memory at declaration while a pointer does not
  - char str[] = "notre dame";
    char *p = str;
    str = "notre lave"; // CANNOT do this
    p = "notre lane"; // CAN do this
  - str is fixed and cannot take in a new address
  - p can take in a new address and stores the location of 'n' (with rest of string being contingently stored in memory)
  - strcpy(str, "CSE"); // CAN do this
  - strcpy(p, "CSE"); // CANNOT do this
  - strcpy() requires memory whereas p = "" does not
    - arrays allocate memory at declaration
    - pointers do not allocate any memory

Malloc(): dynamically allocate memory for pointers

- p = malloc(20 * sizeof(int));
- allocates 20 bytes so strcpy() works now

Free(): must free memory block after use to prevent memory leaks

- free(p); // frees memory and returns p to initial NULL state

Memory Leaks: assigning memory to a pointer and then reassigning that pointer without freeing the memory, thus losing the location of the memory

Stack: static memory → variables
- automatically releases its memory

Heap: dynamic memory → malloc()
- does not release its memory → memory leaks

```
#define _GNU_SOURCE
strfry(str); // scrambles a string
```

Pointing to Structs:

```
typedef struct {
    int l;
    char name[20];
} Square;
Square s1;
Square *p;
p = &sq1;
*p.name; // does NOT work
(*p).name; // fixes this
Syntactic Sugar: p->name;
```

because of C operator precedence

Pointing to Struct Arrays:

```
Square shapes[];
p = shapes; // already an address
for (int i = 0; i < n; i++) {
    printf("%s %d", p->name, p->l);
    p++; // prints entire array with no indexes
}
p = shapes; // MUST reset pointer after
```

Void Pointers: generic pointer that can be given the address of any datatype

- void *p; // cannot dereference yet
- Must typecast to dereference

```
p = &v;
*(int *) p;   // accesses int
*(float *) p;   // accesses float
etc...
```

LIFO: last in first out memory
  ex. stack of plates or pancakes

FIFO: first in first out memory
  ex. standing in line / queues

Note: every time a function is called, its contents are sent to the stack, and stored LIFO (and removed after run)

Stack Overflow: error caused by running out of stack memory (often occurs with infinite recursion)

Recursive Functions: functions that call themselves

```
void recurse(int n) { // recursing var
  if (n > 3) return;   // base case ①
  // commands (called on the way down to base case)
  recurse(n+1);   // recursive call ②
        └→ // modify parameter ③
  // comands (called after base case during
               backtracking)
}
```

* practice rewriting loops as recursive functions * !!

C++ Basics
#include <iostream> (main I/O include
(all #includes have different names)

File Extension: .cpp
Compile with: g++

Must add using namespace std;
otherwise std:: argument for everything!
↳ scope operator
Streams: I/O is done in streams
cin >> variable;
cout << variable << endl;
(no formatting needed)

Strings: #include <string> introduces C++ strings
- treated as objects
- passed by value now (b/c not an address)
- str = "string"; works (also str + "string")
- str.length(); returns length
  ↳ member, OOP paradigm
- no '\0' appended
- cppstring.c_str(); returns C string

Note: Cpp is backwards compatable with C
- heavier focus on abstractions in term
  and object oriented programming

Passing by Reference:
void func(int); passed by value
void func(int *); passed by reference in C
void func(int &); passed by reference in Cpp

called with: func(n); in main
(Cpp abstracts main)

Function Overloading: 2 functions can have the same name if they have different arguments or datatypes
- compiler automatically discerns this
- void func(int) and void func(int&) CANNOT be overloaded because they have the same call
  - must have 0 ambiguity to be overloaded

Range-Based Loops: automatically navigates through array elements of an array set

```
for (int n: arr)
    cout << n;
```

- lose control — more abstraction

```
for (char w: str)
    cout << w;
```

Static Variables: variables that remain in memory after its scope terminates
- permanent variables, kind of like global variables

```
static int n;
```