CS 222: Aggregate Data, Pointers, File I/O

Chris Kauffman

Week 5-1

Logistics

Reading

- Ch 8 (pointers)
- Ch 9 (file i/o)

Exam 2

Next Week Thursday

Homework

- HW4 due tonight
- HW5 up tomorrow
- Multidimensional arrays
- Some string processing

Goals Today

- Feedback
- More memory allocation
- Structs and memory
- Multidimensional Arrays
- File I/O

Midterm Feedback

- Will post online
- No major overhauls indicated
- Comments on time spent on HW

HW 4 Due Tonight

Advice

- Don't need to allocate any memory for structs, only `doubles`
- Make sure to allocate the right amount of memory, manual inspection may lose points
- Make sure to `free()` in appropriate problems
- Favor square brace/array access

  `x[i] + y[i]`

  to pointer arithmetic

  `(*x) + (*y)`

  `x=x+1; y=y+1;`

Questions?

Composing Elements: Arrays of Structs

See `solarsys.c`

A `struct` with an array

```
typedef struct{
  char name[128];
  double dist;
} planet_t;
...
{
  planet_t earth =
    {"earth", 1.0};
}
```

An array of `structs`

```
planet_t solarsys[9];
double d5 = solarsys[5].dist;
```

Later, `structs` with `structs` as elements

Modifications to `solarsys.c`

1. Dynamically allocate an array of 8 planets; copy over the first 8 into this array and print the new system
2. Dynamically allocate space for a single planet. Copy Pluto into that space and print its name and distance
3. Free the dynamically allocated space

Q: How does one dynamically allocate memory in C? Where does it come from?

Code Vs Data

In `solarsys.c` we have a nice way to express the layout of some data in code.

- This doesn't happen very often in C, C++, Java, etc.
- It happens *a lot* in Lisp, ML, Haskell, Python, etc.

Compose

Seen so far

- Structures with array fields
  - `planet_t` with character array
- Arrays of structures
  - `planet_t solar_sys[9]`
- Structures with `struct` fields
  - `planet_t` with `axis_t` as a field

New

- Arrays of arrays (of arrays of . . .)
  - `double matrix[4][3];`
  - `cubicle office[2][3][2];`

Multidimensional Access/Assign

Use multiple `[ ]`, one for each dimension of multi-D array

```
double matrix[4][3];
matrix[0][0] = 1.0;
matrix[0][1] = 10.0;
matrix[0][2] = 100.0;

matrix[1][0] = 2.0;
...
matrix[3][2] = 400.0;

double z = matrix[0][1] * 4;
```

Alternative: initialize whole thing, see `matrix_init.c`

Pointers to pointers (to pointers . . .)

Can point to another pointer

```
int i = 1;
int *ip = &i;
int **ipp = &ip;
int ***ippp = &ipp;

printf("%d\n", *ip);
printf("%d\n", **ipp);
printf("%d\n", ***ippp);
```

Dynamically allocated matrix

What if we want the following:

```
Input matrix size: 8 2
El 0 0: 0.0
El 0 1: 0.1
El 1 0: 1.0
El 1 1: 1.1
...
El 7 1: 7.1
```

i.e. read size and elements from user?
See `matrix_dynamic_alloc.c` for examples

- Modify to read rows/cols from user

Ragged Matrix

Not required to have every row the same length

- Not strictly a matrix then
- Can be useful: `char **strs`

BREAKTIME

Back in 15 minutes

Convenience

Going to work with files in a moment. Usually tell programs what files to operate on the command line

```
gcc sin_sample.c -lm
ls -l read_planets.c
rm a.out sin_sample.c~
```

It's time to communicate with command line arguments

`main` unveiled

Simplest: `int main(){`

More common: `int main(int argc, char **argv){`

- What kind of a thing is `argv`?

Command Line Arguments

Passed to C programs through main.

- `argc` how many, always 1 which is name of program running
- `argv` array of strings which are the actual arguments, `argv[0]` is always name of running program

See `cmdlineargs.c`

Exercise: Yoda-ize

Write a `main()` which prints the first two command line arguments last but the rest in order

```
> gcc yodaize.c
> a.out
> a.out hi
hi
> a.out hi there
hi there
> a.out i am bored
bored i am
```

```
> a.out are you ready
ready are you
> a.out you need more patience
more patience you need
> a.out that is the shadow of greed
the shadow of greed that is
```

- Check `argc` for number of arguments
- `argv[0]` is always the program name (`a.out` or `a.exe`): ignore it
- Look for `argv[1]` and `argv[2]`; print them last
- Print `argv[3]`, `argv[4]`, `...` first

-option

Manually handling command line args is good for simple programs

```
myprog input.dat output.out
myprog input.dat output.out options.opt
```

What about gcc or other complex programs?

```
gcc prog.c
gcc -o prog prog.c
gcc prog.c -o prog.c
gcc -o prog prog.c -lm -O3 -std=C99 -g -Wall -I ../include
```

This gets out of hand fast: *use a library like GNU getopt.*

Common uses of command line arguments

Instruct program to do something multiple times

- `int n = atoi(argv[1]);`
- Converts string to int
- `double n = atof(argv[1]);`
- Converts string to double
- See `range_cmdline.c`

Name one or more files on which to operate

- Read from file
- Leave output in files
- Next topic: File I/O

```
> gcc range_cmdline.c
> a.out
usage: a.out start stop
  start: starting integer
  stop: ending integer
> a.out 1 5
1
2
3
4
5
> a.out 2 9
2
3
4
5
6
7
8
9
```