C for Java Programmers & Lab 0

Don Porter

Portions courtesy Kevin Jeffay

Same Basic Syntax

- Data Types: int, char, [float]
  - void - (untyped pointer)
  - Can create other data types using typedef
- No Strings - only char arrays
  - Last character needs to be a 0
    - Not ‘0’, but ‘\0’

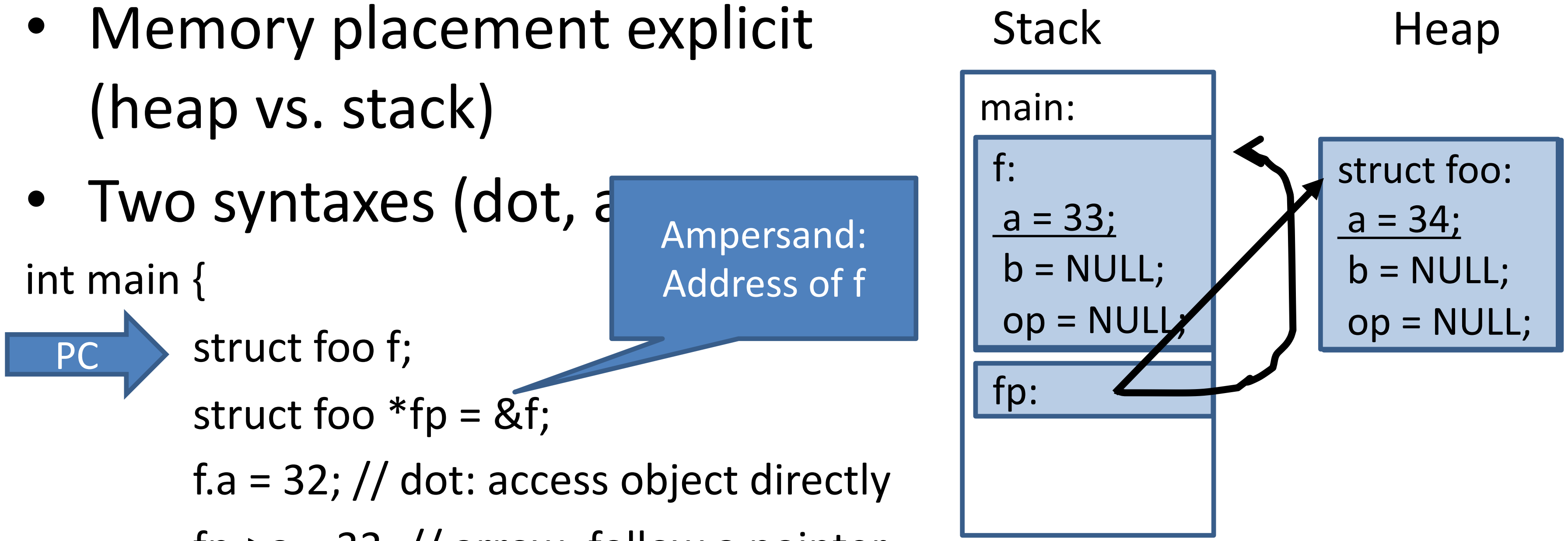
struct – C's object

- typedef struct foo {
  ```
      int a;
      void *b;
      void (*op)(int c);  // function pointer
  } foo_t;      // <------type declaration
  ```
- Actual contiguous memory
- Includes data and function pointers

Pointers

- Memory placement explicit (heap vs. stack)
- Two syntaxes (dot, a

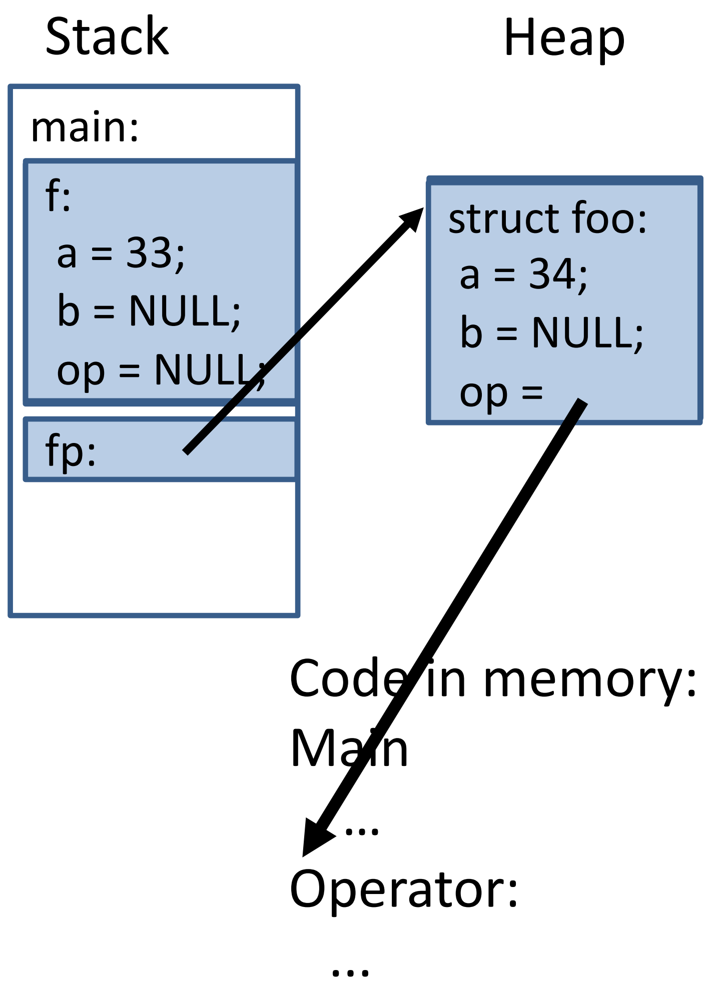
```
int main {
        struct foo f;
        struct foo *fp = &f;
        f.a = 32; // dot: access object directly
        fp->a = 33; // arrow: follow a pointer
        fp = malloc(sizeof(struct foo));
        fp->a = 34;
        …
}
```



```
struct foo {
        int a;
        void *b;
        void (*op)(int c);
}
```

Function pointer example

```
fp->op = operator;
fp->op(32); // Same as calling
            // operator(32);
```



```
struct foo {
        int a;
        void *b;
        void (*op)(int c);
}
```

More on Function Pointers

- C allows function pointers to be used as members of a struct or passed as arguments to a function
- Continuing the previous example:

```
void myOp(int c){ /*…*/ }
/*…*/
foo_t *myFoo = malloc(sizeof(foo_t));
myFoo->op = myOp; // set pointer
/*…*/
myFoo->op(5); // Actually calls myop
```

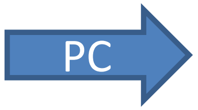
No Constructors or Destructors

- Must manually allocate and free memory - No Garbage Collection!
  - void *x = malloc(sizeof(foo_t));
    - sizeof gives you the number of bytes in a foo_t - DO NOT COUNT THEM YOURSELF!
  - free(x);
    - Memory allocator remembers the size of malloc'ed memory
- Must also manually initialize data
  - Custom function
  - memset(x, 0, sizeof(*x)) will zero it

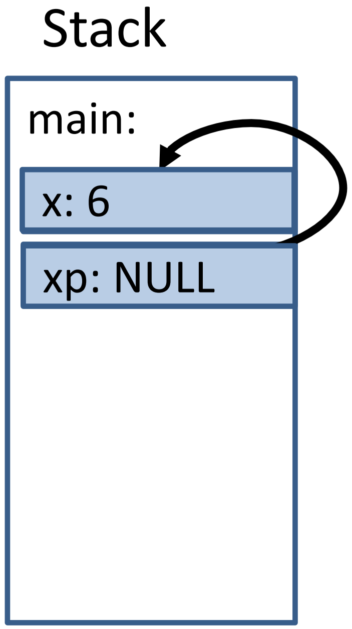
Memory References

- '.' - access a member of a struct
  - myFoo.a = 5;
- '&' - get a pointer to a variable
  - foo_t * fPointer = &myFoo;
- '->' - access a member of a struct, via a pointer to the struct
  - fPointer->a = 6;
- '*' - dereference a pointer
  - if(5 == *intPointer){…}
    - Without the *, you would be comparing 5 to the address of the int, not its value.

Int example



```
int x = 5;  // x is on the stack
int *xp = &x;
*xp = 6;
printf("%d\n", x);  // prints 6
xp  = (int *) 0;
*xp = 7; // segmentation fault
```



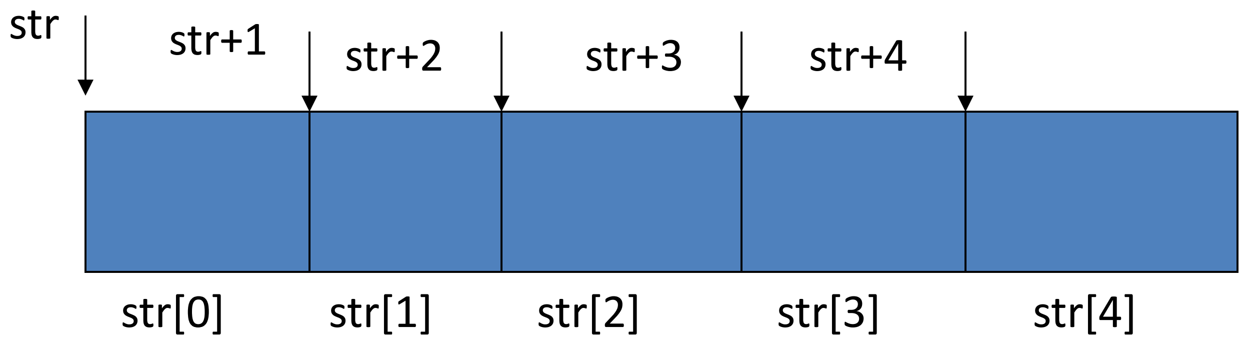
Memory References, cont.

- '[]' - refer to a member of an array

  char *str = malloc(5 * sizeof(char));

  str[0] = 'a';

  - Note: *str = 'a' is equivalent
  - str++; increments the pointer such that *str == str[1]



The Chicken or The Egg?

- Many C functions (printf, malloc, etc) are implemented in libraries
- These libraries use system calls
- System calls provided by kernel
- Thus, kernel has to “reimplement” basic C libraries
  - In some cases, such as malloc, can’t use these language features until memory management is implemented

For more help

- man pages are your friend!
  - (not a dating service)!
  - Ex: 'man malloc', or 'man 3 printf'
    - Section 3 is usually where libraries live - there is a command-line utility printf as well
- Use 'apropos *term*' to search for man entries about *term*
- *The C Programming Language* by Brian Kernighan and Dennis Ritchie is a great reference.

Lab 0 Overview

- C programming on Linux refresher
- Parser for your shell (Lab 1)

Shells

- Shell: aka the command prompt

- At a high level:

```
while (more input) {
    read a line of input                   We will give you this
    parse the line into a command          Lab 0
    if valid command: execute it           Lab 1
}
```

Detour: Environment Variables

- Nearly all shell commands are actually binary files
  - Very few commands actually implemented in the shell
  - A few built-ins that change the shell itself (exit, cd)
- Example: **`ls`** is actually in **`/bin/ls`**
  - For fun, play with **`which`**, as in **`which ls`**
- So where to look for a given command?
  - Note that we want some flexibility system-to-system
- Idea: dynamically set a variable that controls which directories to search

Environment Variables

- A set of key-value pairs
  - Passed to main() as a third argument
  - Often ignored by programmers
- Serves many different purposes
- For Lab 0, we need to look at PATH
  - By convention, a single, colon-delimited set of prefixes
- Example:

```
/usr/local/sbin:/usr/local/bin:/usr/s
bin:/usr/bin:/sbin:/bin
```

PATH in a shell

- If my PATH is

```
/usr/local/sbin:/usr/local/bin:/usr/sbin
:/usr/bin:/sbin:/bin
```

- Then, for a given command (ls), the shell will check, in order, until found:

```
/usr/local/sbin/ls
/usr/local/bin/ls
/usr/sbin/ls
/usr/bin/ls
/sbin/ls
/bin/ls
```

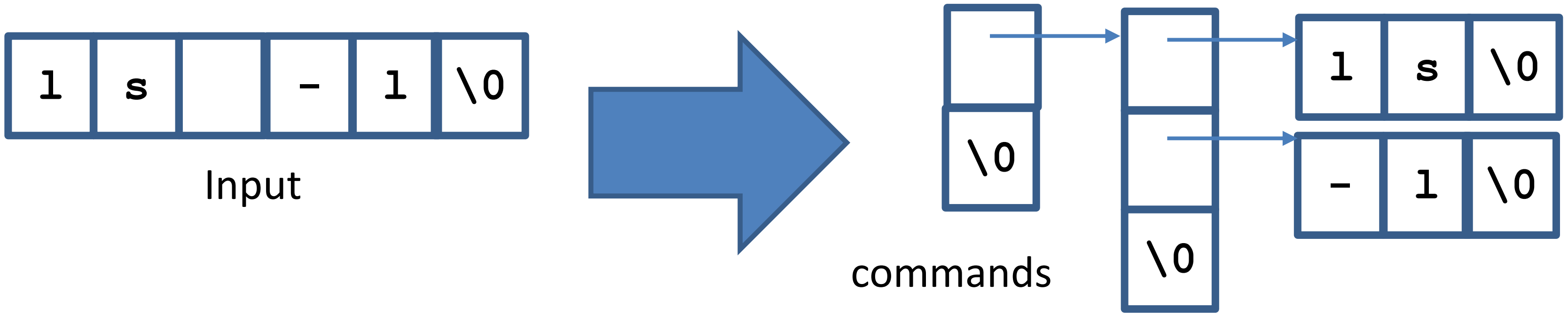
Lab 0, Exercise 1

- Your first job will be to write parsing code that takes in a colon-delimited set of prefixes, and to create a table of prefixes to try in future commands
  - See path_table in jobs.c
  - We wrote a test harness test_env.c

```
$ PATH=/foo:/bar ./test_env
===== Begin Path Table =====
Prefix 0: [/foo]
Prefix 1: [/bar]
===== End Path Table =====
```

Ex 2: Parsing commands

- A typical shell command includes a main binary (e.g., ‘ls’)
  - and 0+ whitespace-separated arguments (e.g., ‘-l’)
  - and possibly extra whitespace
- You will get this as a single character array
- Your job is to break this up into individual ‘tokens’



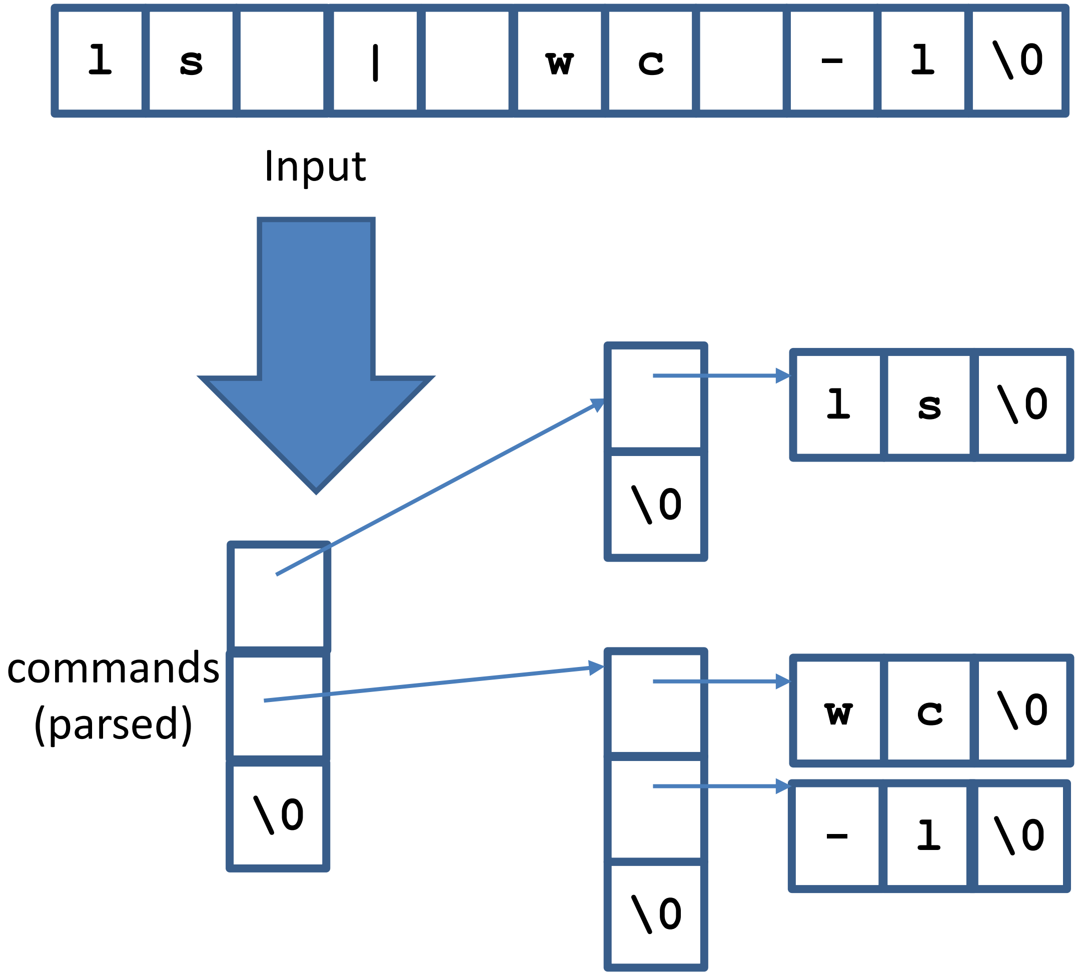
Pipelines

- A shell can compose multiple commands using pipelines
  - Key idea: standard output of one command becomes standard input of next
- Example: **`ls | wc -l`**
  - List a directory (ls) – send listing output to a wordcount utility (wc) to count how many entries in directory
- The vertical bar (|) is a special character
  - May not appear in any other valid commands
  - Does not need whitespace: **`ls|wc -l`** is valid

parse.c:parse_line()

- The workhorse for lab 0 (and 1)
- Takes in a line of input, outputs a 2-D array
- First dimension: one entry per pipeline stage
  - Simple commands just have one entry
- Second dimension: one entry per command token

How to parse a pipeline?



Other special cases

- Comments – anything past a '#' character
- File redirection - sets standard input/output to a file
  - Example: `ls > mydir.txt`
    - Saves the output of ls into a file
  - Example: `wc -l < mydir.txt`
    - Sends the contents of mydir.txt into wc as standard input
- Built-in commands (see builtin.c)
  - For now, you just need to recognize them and call the appropriate handler function

Working on Homework Assignments

- You should all have accounts on comp530fa20.cs.unc.edu
  - Use your ONYEN to log in
- You are welcome to use your own laptop, but code must work on comp530fa20 !

Checking out the starter code

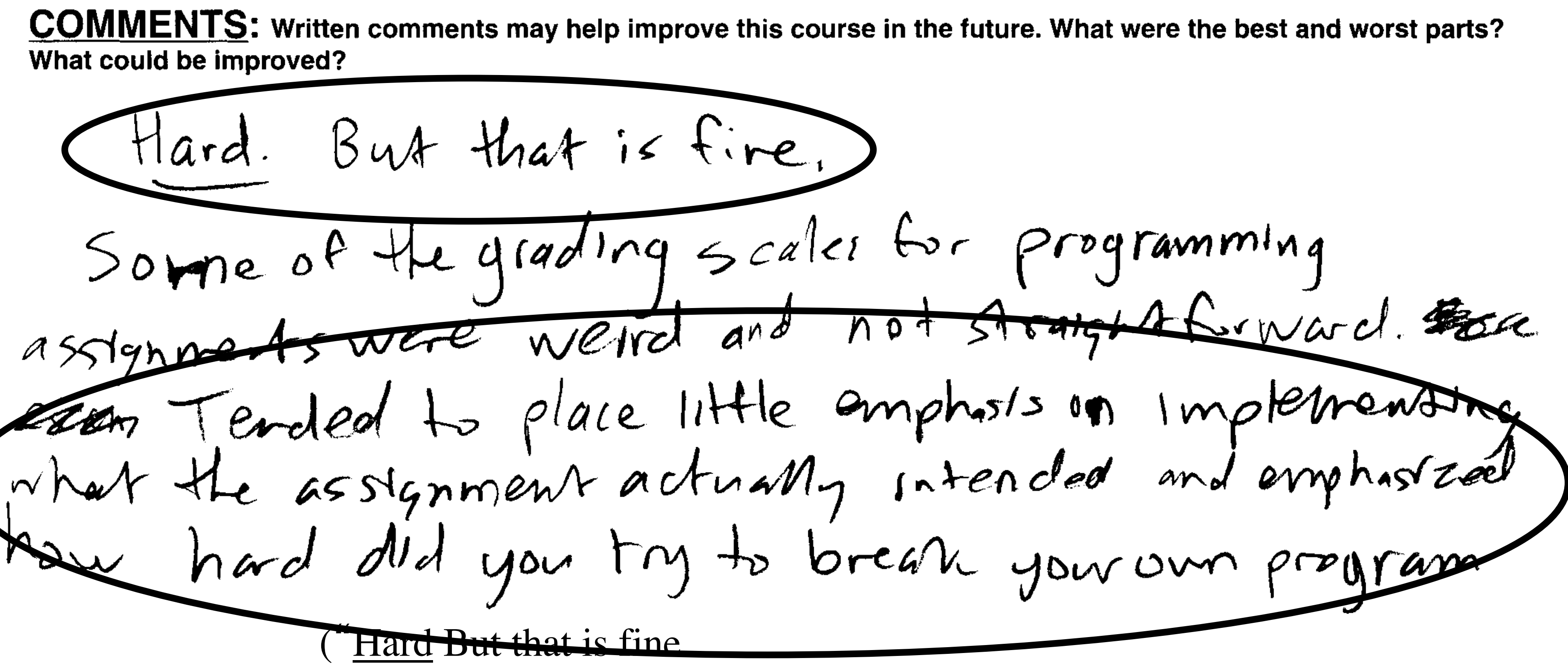
- Once you have a github account registered
  - Make sure you accept the invite:
    - Click https://github.com/comp530-f20
- Click the link in the homework to create a private repo
- Then, on your machine or classroom (substituting your team for ‘team-don’ – see the green clone button):

```
git clone git@github.com:comp530-f20/thsh-team-don.git
```

Submitting homework

- We will be using gradescope to submit and autograde the homework
  - Challenge problems and late hours done manually
  - Submit challenges separately
- Ideally, use github connection to directly submit
  - Upload ok
- Feel free to try early to catch issues with grading

Dr. Jeffay’s Experience



COMMENTS: Written comments may help improve this course in the future. What were the best and worst parts? What could be improved?

Hard. But that is fine.

Some of the grading scales for programming assignments were weird and not straightforward. ~~[illegible]~~ ~~exam~~ Tended to place little emphasis on implementing what the assignment actually intended and emphasized how hard did you try to break your own program

(“Hard But that is fine
Some of the grading scales for programming
assignments were weird and not straightforward.

- Programs that “mostly work” don’t cut it in a senior-level course!

Honor Code: Acceptable and Unacceptable Collaboration

- Working in teams on programming assignments is OK
  - But you can only collaborate with other students in the course
  - Every line of code handed in must be written exclusively by team members themselves, and
  - All collaborators must be acknowledged in writing (and part of the team)
- Use of the Internet
  - Using code from the Internet in any form is not allowed
  - Websites may be consulted for reference (*e.g*., to learn how a system call works)
  - But all such websites used or relied on must be listed as a reference in a header comment in your program
  - *Warning: Sample code found on the Internet rarely helps the student*