L5: Pointers

I/O in C

- Stdin, stdout, stderr standard stream for I/O - defined in stdio.h
- Fprintf - write to specified streams
- Sprintf - writes to a specified character array in memory

Pointers

- Stores an address in memory
- This may be the address of another variable

L6: Dynamic Structures

Structs

- Bundle data of different types together
- Similar to classes in Python and Java
- However they don't have:
    - No methods
    - Different declaration syntax
    - Arrow operator for accessing through a pointer

The Stack

When a function call occurs and new stack frame is added to the bottom of the stack
Stack frame contains:

- Arguments
- Space for local variables
- Call linkage information

Freeing Memory

Any allocated space must be freed when no longer required
Linked lists are good

L7: Working with Files

- Everything is a file in c
- File is a sequence of bytes
- File permission:
    - Levels:
        - User
        - Group (collection of users)
        - Oterh
    - Types:
        - Read
        - Write
        - Execute

L8: C Projects and Make

.c File

- Anything that causes the compiler to generate code should go into
- Function bodies
- Global variable declarations

.h File

- Anything intended as a message to the compiler
- Function prototypes
- Struct definitions
- Typedefs
- #defines

Make

```
# standard variables used by built-in rules
CFLAGS = -Wall -g
RCS = slen.c

# other variables.

OBJS = slen.o recstrlen.o
PROGRAM = slen

# $@ is a built-in variable that expands to the target
# name
$(PROGRAM): $(OBJS)
    gcc -o $@ $(OBJS)

clean:
    rm -f $(PROGRAM) $(OBJS)
```

L9: Debugging with gdb

Errors in C

- Common in all languages
    - Typos
    - Incorrect algorithms
- Buffer overruns
- Misuse of pointers
- Segmentation Fault

Segmentation Fault

- Happens when you program tries to access memory that has not been allocated to it
- Is a signal that the OS sends to your program, causing it to exit
- OS manages memory, as it manages all hardware; it won't let your program read from or write to memory segments it doesn't own

Handling Errors

- Check return values from functions such as *malloc()* and *fopen()*
- Check that pointers are not null
- Always initialize pointer to null on creation if you're not giving them a valid target immediately

Source Level Debuggers

- Compile program using *-g* flag
- *gdb ./myprog*
- Then *run*
- When the program crashes type where
- Show the line where the program crashed
    - *where*

If the program takes command-line arguments, you can't put them on the *gdb* command line

- Solution : *run (command-line-args)*

Example:

```
gcc -Wall -g -o myprog myprog.c
gdb ./myprog
run abcde 1234
```

Stack Frames

What if the line of code that crashed wasn't in the function with the error, but in something that it called?

Commands that will help

- *bt* : shows you a stack trace
- *up* : moves you up from the stack frame
- *down* : moves you down a stack frame

Setting Breakpoints

Tells gdb to stop running the code when it gets to a certain point

How to set it?

- *break* command
- Break at link 5 of current file : *break 5*
- Break at line 5 of main.c : *break main.c: 5*
- Break at start of myfunc() in utilities.c: *break utilities.c:myfunc*
- To delete it: *clear*

How to run it?

Type *run*