C Boot Camp

February 21st, 2016
Jenna MacCarley
Agenda

- C Basics
- C Libraries
- Debugging Tools
- Version Control
- Compilation
- Demo
C Basics

- The *minimum* you must know to do well in this class
  - You have seen these concepts before
  - Make sure you remember them.

- Summary:
  - Pointers/Arrays/Structs/Casting
  - Memory Management
  - Function pointers/Generic Types
  - Strings
  - GrabBag (Macros, typedefs, header guards/files, etc)
Pointers

- Stores address of a value in memory
  - eg `int*`, `char*`, `int**`, etc
  - Access the value by dereferencing (`*a`); can be used to read value or write value to given address
  - Dereferencing `NULL` causes a runtime error

- Pointer to type `a` references a block of `sizeof(a)` bytes

- Get the address of a value in memory with the `&` operator

- Can alias pointers to same address
Call by Value vs Call by Reference

- **Call-by-value**: Changes made to arguments passed to a function aren’t reflected in the calling function
- **Call-by-reference**: Changes made to arguments passed to a function are reflected in the calling function
- C is a *call-by-value* language
- To cause changes to values outside the function, use pointers
  - Do *not* assign the pointer to a different value (that won’t be reflected!)
  - Instead, *dereference the pointer* and assign a value to that address

```c
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int x = 42;
int y = 54;
swap(&x, &y);
printf(“%d
”, x); // 54
printf(“%d
”, y); // 42
```
**Pointer Arithmetic**

- Can add/subtract from an address to get a new address
  - Only perform when absolutely necessary (i.e., `malloc`)
  - Result depends on the pointer type

- \( A+i, \) where \( A \) is a pointer = 0x100, \( i \) is an \( \text{int} \) (x86-64)
  - \( \text{int}^* \): \( A+i = 0x100 + \text{sizeof} \text{(int)} * i = 0x100 + 4 * i \)
  - \( \text{char}^* \): \( A+i = 0x100 + \text{sizeof} \text{(char)} * i = 0x100 + i \)
  - \( \text{int}^{**} \): \( A + i = 0x100 + \text{sizeof} \text{(int*)} * i = 0x100 + 8 * i \)

- Rule of thumb: cast pointer explicitly to avoid confusion
  - Prefer (\( \text{char}^* \))(A) + i vs A + i, even if \( \text{char}^* \) A
  - Absolutely do this in macros (i.e., `malloc`)
Structs

- Collection of values placed under one name in a single block of memory
  - Can put structs, arrays in other structs
- Given a struct *instance*, access the fields using the ‘.’ operator
- Given a struct *pointer*, access the fields using the ‘->’ operator

```c
struct foo_s {
    int a;
    char b;
};

struct bar_s {
    char ar[10];
    foo_s baz;
};

bar_s biz; // bar_s instance
biz.ar[0] = 'a';
biz.baz.a = 42;

bar_s* boz = &biz; // bar_s ptr
boz->baz.b = 'b';
```
Arrays/Strings

- Arrays: fixed-size collection of elements of the same type
  - Can allocate on the stack or on the heap
  - `int A[10]; // A is array of 10 int’s on the stack`
  - `int* A = calloc(10, sizeof(int)); // A is array of 10 int’s on the heap`

- Strings: Null-character (‘\0’) terminated character arrays
  - Null-character tells us where the string ends
  - All standard C library functions on strings assume null-termination.
Casting

- Can cast a variable to a different type
- Integer Type Casting:
  - signed <-> unsigned: change interpretation of most significant bit
  - smaller signed -> larger signed: sign-extend (duplicate the sign bit)
  - smaller unsigned -> larger unsigned: zero-extend (duplicate 0)
- Cautions:
  - cast explicitly, out of practice. C will cast operations involving different types implicitly, often leading to errors
  - never cast to a smaller type; will truncate (lose) data
  - never cast a pointer to a larger type and dereference it, this accesses memory with undefined contents
C Program Memory Layout

- **Text**
  - read from program file by exec

- **Initialized data**
  - initialized to zero by exec

- **Uninitialized data (bss)**

- **Heap**

- **Stack**

- **High address**

- **Low address**

- **Command-line arguments and environment variables**
Malloc, Free, Calloc

- Handle dynamic memory allocation on HEAP
  - `void* malloc (size_t size)`: 
    - allocate block of memory of size bytes 
    - does not initialize memory
  - `void* calloc (size_t num, size_t size)`: 
    - allocate block of memory for array of num elements, each size bytes long 
    - initializes memory to zero values
  - `void free(void* ptr)`: 
    - frees memory block, previously allocated by malloc, calloc, realloc, pointed by ptr 
    - use exactly once for each pointer you allocate

- size argument: 
  - should be computed using the `sizeof` operator
  - `sizeof` takes a type and gives you its size
  - e.g., `sizeof(int), sizeof(int*)`
Memory Management Rules

- Malloc what you free, free what you malloc
  - client should free memory allocated by client code
  - library should free memory allocated by library code

- Number mallocs = Number frees
  - Number mallocs > Number Frees: definitely a memory leak
  - Number mallocs < Number Frees: definitely a double free

- Free a malloced block exactly once
  - Should not dereference a freed memory block
Stack vs Heap vs Data

- Local variables and function arguments are placed on the stack
  - deallocated after the variable leaves scope
  - do not return a pointer to a stack-allocated variable!
  - do not reference the address of a variable outside its scope!
- Memory blocks allocated by calls to malloc/calloc are placed on the heap
- Globals, constants are placed in data section
- Example:
  // a is a pointer on the stack to a memory block on the heap
  int* a = malloc(sizeof(int));
Typedefs

- Creates an alias type name for a different type
- Useful to simplify names of complex data types
- Be careful when typedef-ing away pointers!

```c
struct list_node {
    int x;
};

typedef int pixel;
typedef struct list_node* node;
typedef int (*cmp)(int e1, int e2);

pixel x; // int type
node foo; // struct list_node* type
cmp int_cmp; // int (*cmp)(int e1, int e2) type
```
Macros

- Fragment of code given a name; replace occurrence of name with contents of macro
  - No function call overhead, type neutral

- Uses:
  - defining constants (INT_MAX, ARRAY_SIZE)
  - defining simple operations (MAX(a, b))
  - 122-style contracts (REQUIRES, ENSURES)

- Warnings:
  - Use parentheses around arguments/expressions, to avoid problems after substitution
  - Do not pass expressions with side effects as arguments to macros

```c
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)
#define WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```
Generic Types

- void* type is C’s provision for generic types
  - Raw pointer to some memory location (unknown type)
  - Can’t dereference a void* (what is type void?)
  - Must cast void* to another type in order to dereference it
- Can cast back and forth between void* and other pointer types

// stack implementation:

typedef void* elem;

stack stack_new();
void push(stack S, elem e);
elem pop(stack S);

// stack usage:

int x = 42; int y = 54;
stack S = stack_new();
push(S, &x);
push(S, &y);
int a = *(int*)pop(S);
int b = *(int*)pop(S);
Header Files

- Includes C declarations and macro definitions to be shared across multiple files
  - Only include function prototypes/macros; no implementation code!
- Usage: `#include <header.h>`
  - `#include <lib>` for standard libraries (eg `#include <string.h>`)  
  - `#include “file”` for your source files (eg `#include “header.h”`)  
  - Never include .c files (bad practice)

```c
// list.h
struct list_node {
    int data;
    struct list_node* next;
};
typedef struct list_node* node;
node new_list();
void add_node(int e, node l);

// list.c
#include “list.h”
node new_list() {
    // implementation
}
void add_node(int e, node l) {
    // implementation
}

// stacks.h
#include “list.h”
struct stack_head {
    node top;
    node bottom;
};
typedef struct stack_head* stack;
stack new_stack();
void push(int e, stack S);
```
Header Guards

- Double-inclusion problem: include same header file twice

//grandfather.h
#include "grandfather.h"

//father.h
#include "father.h"

//child.h
#include "father.h"
#include "grandfather.h"

Error: child.h includes grandfather.h twice

- Solution: header guard ensures single inclusion

//grandfather.h
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
#endif

//father.h
#ifndef FATHER_H
#define FATHER_H
#endif

//child.h
#include "father.h"
#include "grandfather.h"

Okay: child.h only includes grandfather.h once
Variable Declarations & Qualifiers

- **Global Variables:**
  - Defined outside functions, seen by all files
  - Use “extern” keyword to use a global variable defined in another file

- **Const Variables:**
  - For variables that won’t change
  - Data stored in read-only data section

- **Static Variables/Functions:**
  - For globals, usable/viewable only from within the current file
  - For locals, keeps value between invocations
  - USE SPARINGLY

- **Volatile Variables:**
  - Compiler will not make assumptions about current value, useful for asynchronous reads/writes, i.e. interrupts
  - “volatile” == “subject to change at any time”
C Libraries
string.h: Common String/Array Methods

- One the most useful libraries available to you
- Used heavily in shell/proxy labs
- Important usage details regarding arguments:
  - prefixes: `str` -> strings, `mem` -> arbitrary memory blocks.
  - ensure that all strings are ‘/0’ terminated!
  - ensure that `dest` is large enough to store `src`!
  - ensure that `src` actually contains `n` bytes!
  - ensure that `src/dest` don’t overlap!
string.h: Common String/Array Methods

- **Copying:**
  - `void* memcpy(void* dest, void* src, size_t n)` : copy n bytes of src into dest, return dest
  - `char* strcpy(char* dest, char* src)` : copy src string into dest, return dest

- **Concatenation:**
  - `char * strcat(char * dest, char* src)` : append copy of src to end of dest, return dest

- **Comparison:**
  - `int strcmp(char * str1, char * str2)` : compare str1, str2 by character (based on ASCII value of each character, then string length), return comparison result
    - `str1 < str2: -1`
    - `str1 == str2: 0`
    - `str1 > str2: 1`
Searching:
- `char* strstr (char * str1, char * str2)`: return pointer to *first* occurrence of `str2` in `str1`, else NULL
- `char* strtok (char * str, char * delimiters)`: tokenize `str` according to delimiter characters provided in `delimiters`, return the next token per successive stroke call, using `str = NULL`

Other:
- `size_t strlen ( const char * str )`: returns length of the string (up to, but not including the ‘\0’ character)
- `void * memset (void* ptr, int val, size_t n )`: set first `n` bytes of memory block addressed by `ptr` to `val` (use this for setting bytes only; don’t use to set int arrays or anything else!)
stdlib.h: General Purpose Functions

- **Dynamic memory allocation:**
  - `malloc`, `calloc`, `free`

- **String conversion:**
  - `int atoi(char* str)`: parse string into integral value (return 0 if not parsed)

- **System Calls:**
  - `void exit(int status)`: terminate calling process, return `status` to parent process
  - `void abort()`: aborts process abnormally

- **Searching/Sorting:**
  - Provide array, array size, element size, comparator (function pointer)
  - `bsearch`: returns pointer to matching element in the array
  - `qsort`: sorts the array destructively

- **Integer arithmetic:**
  - `int abs(int n)`: returns absolute value of `n`

- **Types:**
  - `size_t`: unsigned integral type (store size of any object)
stdio.h

- Another really useful library.
- Used heavily in cache/shell/proxy labs
- Used for:
  - argument parsing
  - file handling
  - input/output
**stdio.h: Common I/O Methods**

- **FILE* fopen (char* filename, char* mode):** open the file with specified filename in specified mode (read, write, append, etc), associate it with stream identified by returned file pointer.
- **int fscanf (FILE* stream, char* format, ...):** read data from the stream, store it according to the parameter format at the memory locations pointed at by additional arguments.
- **int fclose (FILE* stream):** close the file associated with the stream.
- **int fprintf (FILE* stream, char* format, ...):** write the C string pointed at by format to the stream, using any additional arguments to fill in format specifiers.
Getopt

- Need to include `getopt.h` and `unistd.h` to use
- Used to parse command-line arguments.
- Typically called in a loop to retrieve arguments
- Switch statement used to handle options
  - colon indicates required argument
  - `optarg` is set to value of option argument
- Returns -1 when no more arguments present
- Very useful for Cache lab!

```c
int main(int argc, char** argv){
    int opt, x;
    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, "x:"))){
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            default:
                printf("wrong argument\n");
                break;
        }
    }
}
```
Note about Library Functions

- These functions can return error codes
  - malloc could fail
  - a file couldn’t be opened
  - a string may be incorrectly parsed

- Remember to check for the error cases and handle the errors accordingly
  - may have to terminate the program (eg malloc fails)
  - may be able to recover (user entered bad input)
Debugging

GDB, Valgrind
GDB

- No longer stepping through assembly!
  - Use the step/next commands
  - break on line numbers, functions
  - Use list to display code at line-numbers and functions
  - Use print with variables

- Use gdbtui
  - Nice display for viewing source/executing commands
Valgrind

- Find memory errors, detect memory leaks

Common errors:
- Illegal read/write errors
- Use of uninitialized values
- Illegal frees
- Overlapping source/destination addresses

Typical solutions:
- Did you allocate enough memory?
- Did you accidentally free stack variables/something twice?
- Did you initialize all your variables?
- Did use something that you just free’d?

- --leak-check=full
  - Memcheck gives details for each definitely/possibly lost memory block (where it was allocated)
Compilation

GCC, Make Files
GCC

- Used to compile C/C++ projects
  - List the files that will be compiled to form an executable
  - Specify options via flags
- Important Flags:
  - -g: produce debug information (important; used by GDB/valgrind)
  - -Werror: treat all warnings as errors (this is our default)
  - -Wall/-Wextra: enable all construction warnings
  - -pedantic: indicate all mandatory diagnostics listed in C-standard
  - -O0/-O1/-O2: optimization levels
  - -o <filename>: name output binary file ‘filename’
- Example:
  - gcc -g -Werror -Wall -Wextra -pedantic foo.c bar.c -o baz
Make Files

- Command-line compilation becomes inefficient when compiling many files together
- Solution: use make-files
  - Single operation to compile files together
  - Only recompiles updated files

```bash
# Makefile for the malloc lab driver
#
CC = gcc
CFLAGS = -Wall -Wextra -Werror -O2 -g -DDRIVER -std=gnu99

OBJS = mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o

all: mdriver
mdriver: $(OBJS)
    $(CC) $(CFLAGS) -o mdriver $(OBJS)

mdriver.o: mdriver.c fsecs.h fcyc.h clock.h memlib.h config.h mm.h
memlib.o: memlib.c memlib.h
mm.o: mm.c mm.h memlib.h
fsecs.o: fsecs.c fsecs.h config.h
fcyc.o: fcyc.c fcyc.h
ftimer.o: ftimer.c ftimer.h config.h
clock.o: clock.c clock.h

clean:
    rm -f *~ *.o mdriver
```
Make File Rules

- Comments start with a ‘#’, Commands start with a TAB.
- Common Make File Format:
  - target: source(s)
  - TAB: command
  - TAB: command
- Macros: similar to C-macros, find and replace:
  - CC = gcc
  - CCOPT = -g -DDEBUG -DPRINT
  - foo.o: foo.c foo.h
    - $(CC) $(CCOPT) -c foo.c
- See
  - http://www.andrew.cmu.edu/course/15-123-kesden/index/lecture_index.html
  for more details
Questions?