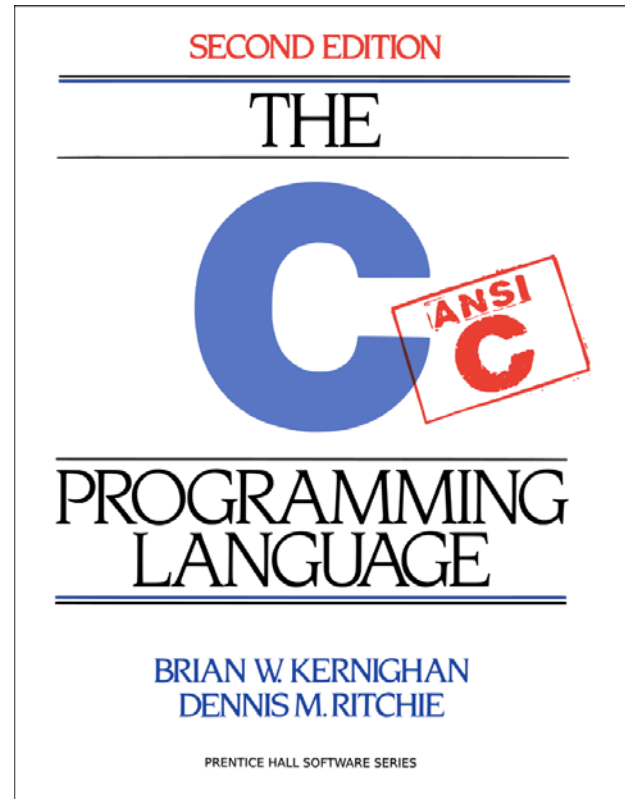# C Boot Camp

10 Oct 2015
Ben Spinelli

**Agenda**

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

# C Basics

# 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

Demo Time!

# Call by Value vs Call by Reference

- <u>Call-by-value</u>: Changes made to arguments passed to a function *aren't* reflected in the calling function
- <u>Call-by-reference</u>: 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

```
void swap(int* a, int* b) {          int x = 42;
   int temp = *a;                    int y = 54;
   *a = *b;                          swap(&x, &y);
   *b = temp;                        printf("%d\n", x); // 54
}                                    printf("%d\n", y); // 42
```

**Demo Time!**

# 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 `int` (*x86-64*)
  - `int* A`: `A+i = 0x100 + sizeof(int) * i = 0x100 + 4 * i`
  - `char* A`: `A+i = 0x100 + sizeof(char) * i = 0x100 + i`
  - `int** A`: `A + i = 0x100 + sizeof(int*) * i = 0x100 + 8 * i`
- Rule of thumb: cast pointer explicitly to avoid confusion
  - Prefer `(char*)(A) + i` vs `A + i`, even if `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

```
struct foo_s {        struct bar_s {        bar_s biz; // bar_s instance
    int a;                char ar[10];      biz.ar[0] = 'a';
    char b;               foo_s baz;        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

# Typedefs

- Creates an *alias* type name for a different type
- Useful to simplify names of complex data types

```
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

```
#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)

```
// 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
```

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

# Odds and Ends

- Prefix vs Postfix increment/decrement
  - `a++`: use `a` in the expression, then increment `a`
  - `++a`: increment `a`, then use `a` in the expression
- Switch Statements:
  - remember break statements after every case, unless you want fall through (may be desirable in some cases)
  - should probably use a default case
- Variable/function modifiers:
  - global variables: defined outside functions, seen by all files
  - static variables/functions: seen only in file it's declared in
  - Refer to K&R for other modifiers and their meanings

# 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!
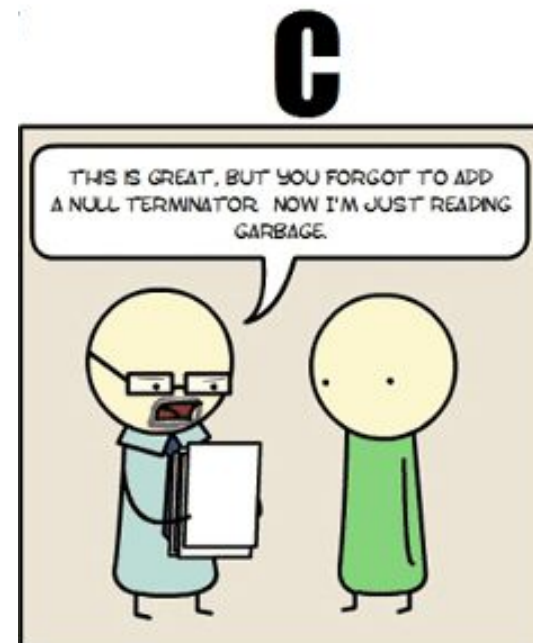
# string.h: Common String/Array Methods

- <u>Copying</u>:
  - `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
- <u>Concatenation</u>:
  - `char * strcat (char * dest, char* src)`: append copy of src to end of dest, return dest
- <u>Comparison</u>:
  - `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

# string.h: Common String/Array Methods (Continued)

- ## <u>Searching</u>:
  - `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`
- ## <u>Other</u>:
  - `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

- <u>Dynamic memory allocation:</u>
  - `malloc, calloc, free`
- <u>String conversion:</u>
  - `int atoi(char* str):` parse string into integral value (return 0 if not parsed)
- <u>System Calls:</u>
  - `void exit(int status):` terminate calling process, return `status` to parent process
  - `void abort():` aborts process abnormally
- <u>Searching/Sorting:</u>
  - provide array, array size, element size, comparator (function pointer)
  - `bsearch:` returns pointer to matching element in the array
  - `qsort:` sorts the array destructively
- <u>Integer arithmetic:</u>
  - `int abs(int n):` returns absolute value of `n`
- <u>Types</u>:
  - `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
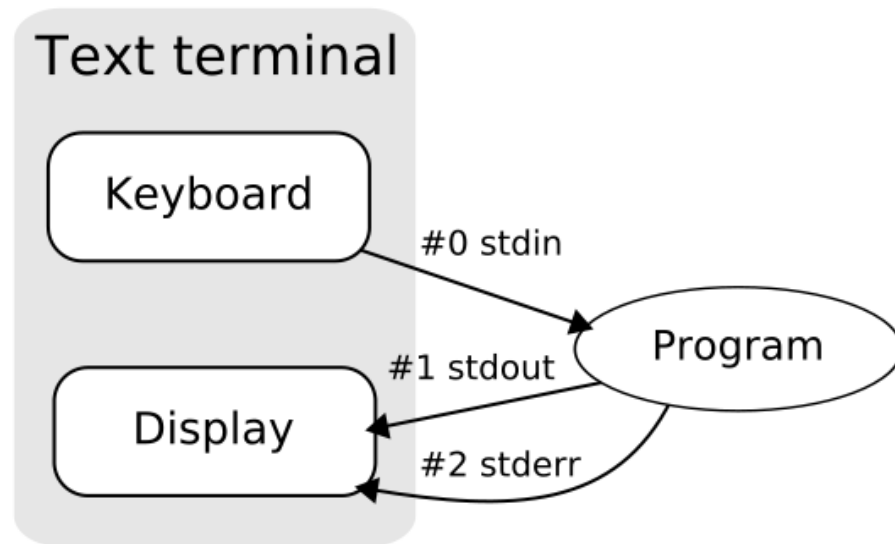
# stdio.h: Common I/O Methods

- `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

# Questions?