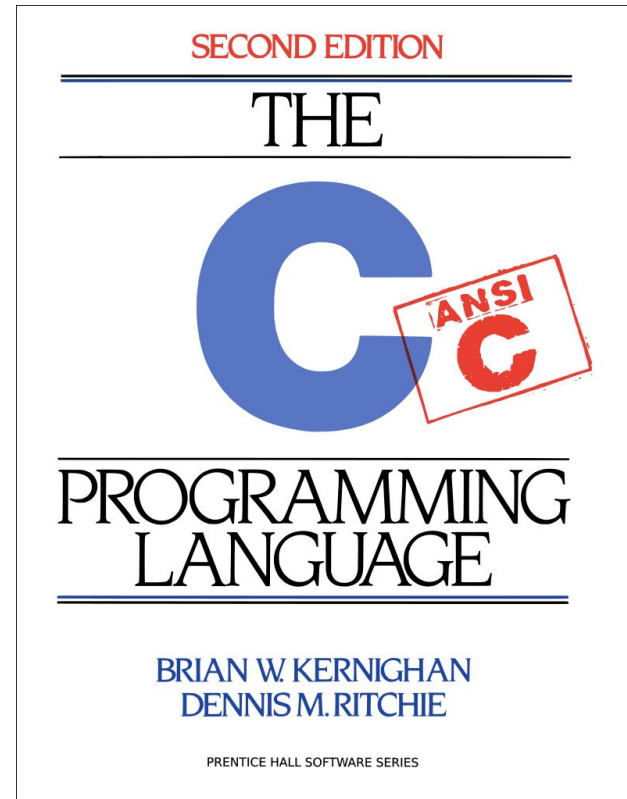


# C Boot Camp

Oct. 1, 2017

Aarohi  
Fernando  
Jiayi  
Satoru  
Stephen



## Agenda

- C Basics
- Debugging Tools / Demo
- Appendix
  - C Standard Library
    - getopt
    - stdio.h
    - stdlib.h
    - string.h



## C Basics Handout

```
ssh <andrewid>@shark.ics.cs.cmu.edu
```

```
cd ~/private
```

```
wget http://cs.cmu.edu/~213/activities/cbootcamp.tar.gz
```

```
tar xvpf cbootcamp.tar.gz
```

```
cd cbootcamp
```

```
make
```

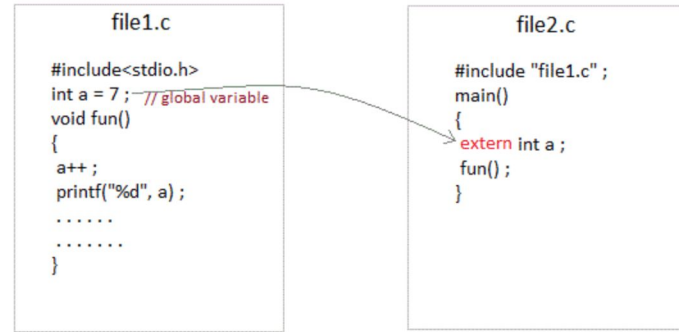
- Contains useful, self-contained C examples
- Slides relating to these examples will have the file names in the **top-right corner!**

# 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

# 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:**
  - For locals, keeps value between invocations
  - USE SPARINGLY
  - Note: static has a different meaning when referring to functions (not visible outside of object file)



global variable from one file can be used in other using **extern** keyword.

```

#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}

```

Output:

# Casting

- Can convert a variable to a different type
- Integer Casting:
  - Signed <-> Unsigned: Keep Bits - Re-Interpret
  - Small -> Large: Sign-Extend MSB, preserve value
- Cautions:
  - Cast Explicitly: `int x = (int) y` instead of `int x = y`
  - Casting Down: Truncates data
  - Cast Up: Upcasting and dereferencing a pointer causes undefined memory access
- Rules for Casting Between Integer Types

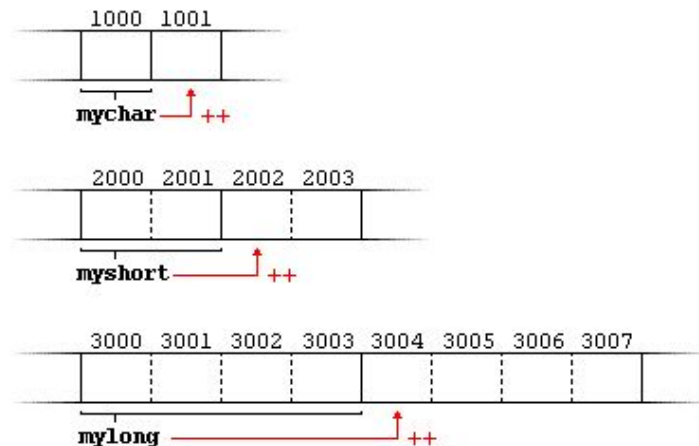
# Pointers

- Stores address of a value in memory
  - e.g. `int*`, `char*`, `int**`, etc
  - Access the value by dereferencing (e.g. `*a`).  
Can be used to read or write a value to given address
  - Dereferencing `NULL` causes undefined behavior (usually a segfault)



# Pointers

- Pointer to type `A` references a block of `sizeof(A)` bytes
- Get the address of a value in memory with the `&` operator
- Pointers can be *aliased*, or pointed to same address





# Pointer Arithmetic

./pointers

- Can add/subtract from an address to get a new address
  - Only perform when absolutely necessary (i.e., malloclab)
  - Result depends on the pointer type
- $A+i$ , where  $A$  is a pointer =  $0x100$ ,  $i$  is an int
  - $\text{int}^* A: A+i = 0x100 + \text{sizeof}(\text{int}) * i = 0x100 + 4 * i$
  - $\text{char}^* A: A+i = 0x100 + \text{sizeof}(\text{char}) * i = 0x100 + 1 * i$
  - $\text{int}^{**} A: A+i = 0x100 + \text{sizeof}(\text{int}^*) * i = 0x100 + 8 * i$
- Rule of thumb: **explicitly** cast pointer to avoid confusion
  - Prefer  $(\text{char}^*) (A) + i$  to  $(A + i)$ , even if  $A$  has type  $\text{char}^*$

## Pointer Arithmetic

`./pointers`

- The 'pointers' program demonstrates how values of different sizes can be written to and read back from the memory.
- The examples are to show you how the `~type~` of the pointer affects arithmetic done on the pointer.
- When adding `x` to a pointer `A` (i.e. `A + x`), the result is really `(A + x * sizeof(TYPE_OF_PTR_A))`.
- Run the 'pointers' program  

```
$ ./pointers
```

# 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

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

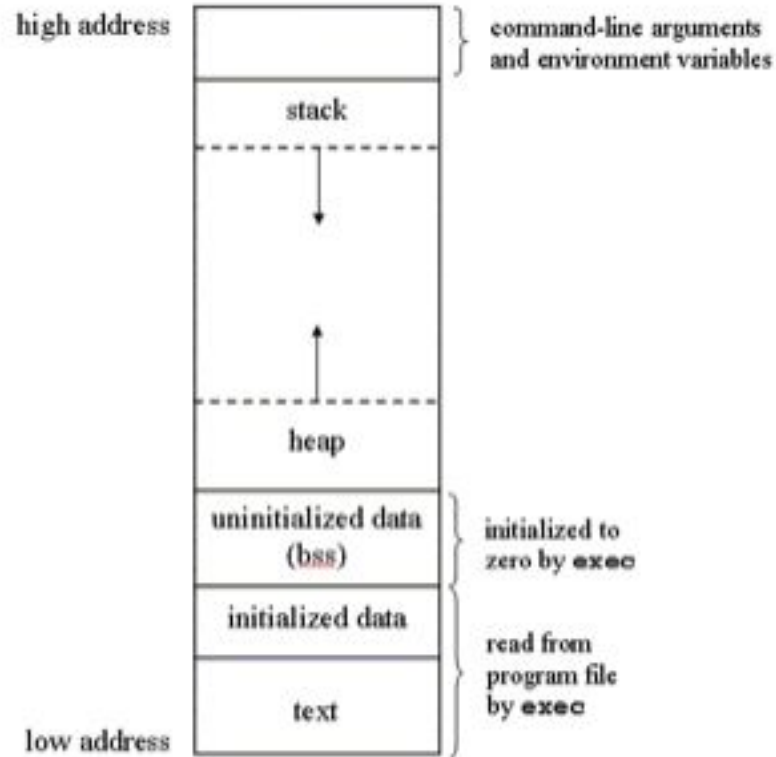
```
int x = 42;  
int y = 54;  
swap(&x, &y);  
printf("%d\n", x); // 54  
printf("%d\n", y); // 42
```

# 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.



# C Program Memory Layout



# 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));`

# 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
- `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*)`



mem\_mgmt.c

./mem\_valgrind.sh

# 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 malloc'ed block exactly once
  - Should not dereference a freed memory block
- Only malloc when necessary
  - Persistent, variable sized data structures
  - Concurrent accesses (we'll get there later in the semester)

# 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)

```

Terminal
File Edit View Terminal Tabs Help
[pwells2@newcell ~/junk]$ valgrind ./memleak
==16738== Memcheck, a memory error detector
==16738== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==16738== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==16738== Command: ./memleak
==16738==
==16738== Invalid write of size 4
==16738==   at 0x400589: main (mem_leak.c:32)
==16738==   Address 0x4c26068 is 0 bytes after a block of size 40 alloc'd
==16738==   at 0x4A0646F: malloc (vg_replace_malloc.c:236)
==16738==   by 0x400505: main (mem_leak.c:17)
==16738==
==16738== Invalid read of size 4
==16738==   at 0x400598: main (mem_leak.c:33)
==16738==   Address 0x4c26068 is 0 bytes after a block of size 40 alloc'd
==16738==   at 0x4A0646F: malloc (vg_replace_malloc.c:236)
==16738==   by 0x400505: main (mem_leak.c:17)
==16738==
==16738== HEAP SUMMARY:
==16738==   in use at exit: 410 bytes in 8 blocks
==16738==   total heap usage: 11 allocs, 3 frees, 590 bytes allocated
==16738==
==16738== LEAK SUMMARY:
==16738==   definitely lost: 410 bytes in 8 blocks
==16738==   indirectly lost: 0 bytes in 0 blocks
==16738==   possibly lost: 0 bytes in 0 blocks
==16738==   still reachable: 0 bytes in 0 blocks
==16738==   suppressed: 0 bytes in 0 blocks
==16738==
==16738== Rerun with --leak-check=full to see details of leaked memory
==16738==
==16738== For counts of detected and suppressed errors, rerun with: -v
==16738== ERROR SUMMARY: 36 errors from 2 contexts (suppressed: 4 from 4)
[pwells2@newcell ~/junk]$

```

# Debugging

GDB

# GDB

- No longer stepping through assembly!  
Some GDB commands are different:
  - `si / si` → `step / next`
  - `break file.c:line_num`
  - `disas` → `list`
  - `print <any_var_name>` (in current frame)
- Use TUI mode (layout src)
  - Nice display for viewing source/executing commands
  - Buggy, so only use TUI mode to step through lines (no continue / finish)

## Additional Topics

- Headers files and header guards
- Macros
- Appendix (C libraries)

# Header Files

- Includes C declarations and macro definitions to be shared across multiple files
  - Only include function prototypes/macros; implementation code goes in .c file!
- 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           //child.h
                           #include "grandfather.h"       #include "father.h"
                                                           #include "grandfather.h"
```

Error: child.h includes grandfather.h twice

- Solution: header guard ensures single inclusion

```
//grandfather.h           //father.h           //child.h
#define GRANDFATHER_H     #ifndef FATHER_H     #include "father.h"
                           #define FATHER_H              #include "grandfather.h"

#endif                     #endif
```

Okay: child.h only includes grandfather.h once

# Macros

./macros

- A way to replace a name with its macro definition
  - No function call overhead, type neutral
  - Think “find and replace” like in a text editor
- 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

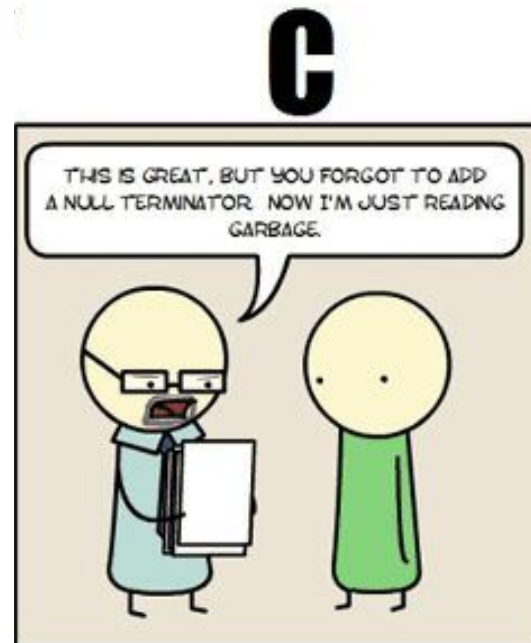
```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)
#define WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```



# 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. Make sure dest is large enough to contain src.`

## ■ Concatenation:

- `char *strncat (char *dest, char *src, size_t n) : append copy of src to end of dest reading at most n bytes, return dest`
- `char *strcat (char *dest, char *src) works for arbitrary length strings, but has the safety issues you've seen in attacklab`

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

- Comparison:

- `int strncmp (char *str1, char *str2, size_t n):` compare at most `n` bytes of `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`
- `int strcmp(char *str1, char *str2):` compare `str1` to `str2`. *Make sure each string is long enough to be safely compared.*

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

### ■ 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 one token for each `strtok` call, using `str = NULL`

### ■ Other:

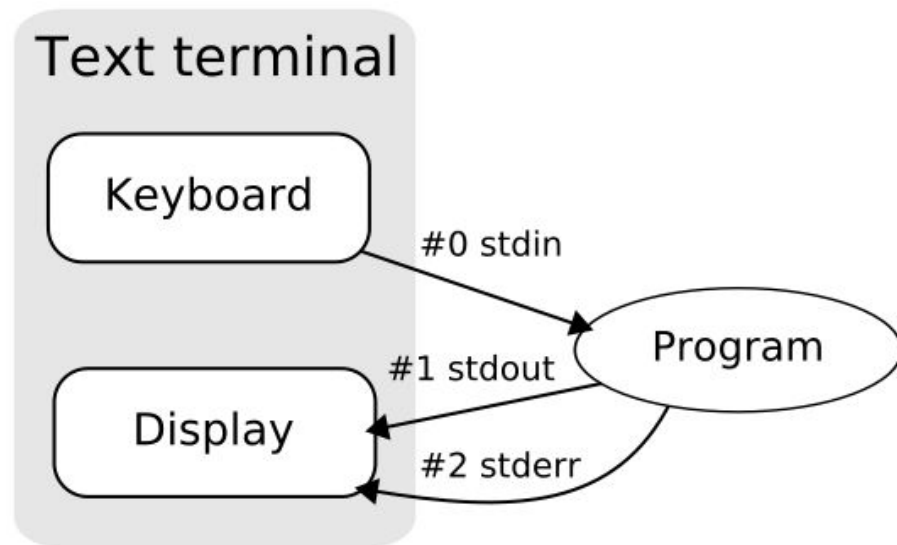
- `size_t strlen (const char *str)`: returns length of the string (up to, but not including the `'\0'` character)
- `void *memset (void *ptr, int val, size_t n)`: set first `n` bytes of memory block addressed by `ptr` to `val`  
For *setting bytes only*. Don't use it to set or initialize `int` arrays, for example.

# 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
- `printf`, a fan favorite, comes from this library!



## 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 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.
- `fgets`



# Getopt

- Need to include `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
- See recitation 6 slides for more examples

```
int main(int argc, char **argv)
{
    int opt, x;
    /* looping over arguments */
    while((opt=getopt(argc,argv,"x:"))>0){
        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
  - `int x;`  
`if ((x = malloc(sizeof(int))) == NULL)`  
`printf("Malloc failed!!!\n");`
  - 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)

# Style

- Documentation
  - file header, function header, comments
- Variable Names & Magic Numbers
  - `new_cache_size` is good, not `new_cacheSize` or `size`
  - Use `#define CACHESIZE 128`
- Modularity
  - helper functions
- Error Checking
  - `malloc`, library functions...
- Memory & File Handling
  - free memory, close files
- Check [style guide](#) for detailed information