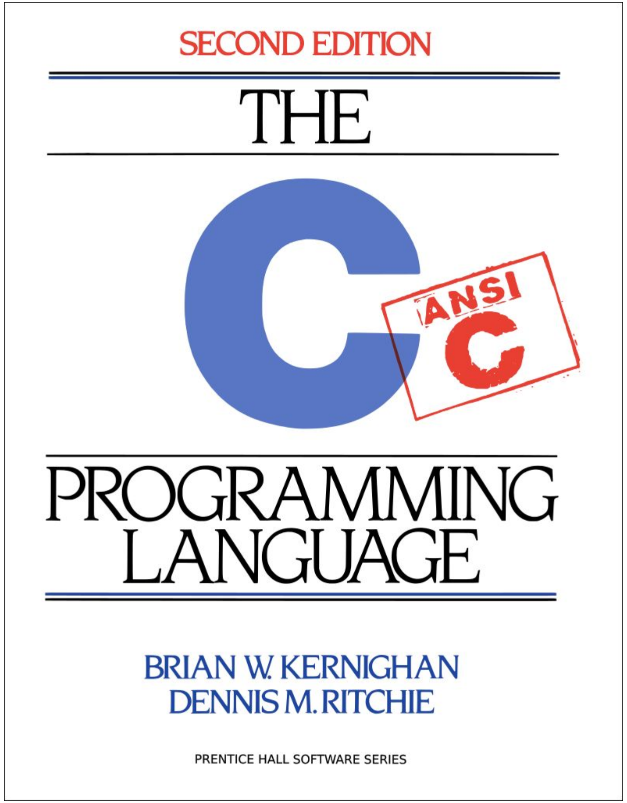
C Boot Camp

Oct 6th, 2017

Jack Biggs
Raghav Goyal
Nikhil Jog



Agenda

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

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

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
  - GrabBag (Macros, typedefs, header guards/files, etc)

Pointers

- Stores address of a value in memory
  - e.g. `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 `./passing_args`

- 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
```

Pointer Arithmetic `./pointer_arith`

- 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`
  - `int*   A: A+i = 0x100 + sizeof(int)   * i = 0x100 + 4 * i`
  - `char*  A: A+i = 0x100 + sizeof(char)  * i = 0x100 + 1 * i`
  - `int**  A: A+i = 0x100 + sizeof(int*)  * i = 0x100 + 8 * i`

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

Structs ./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 {
   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 convert a variable to a different type
- Integer Casting:
  - Signed <-> Unsigned: Keep Bits - Re-Interpret
  - Small -> Large: Sign-Extend MSB
- 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

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*)`

Memory Management Rules

`mem_mgmt.c`

`./mem_valgrind.sh`

- `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

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

./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!

```
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

./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

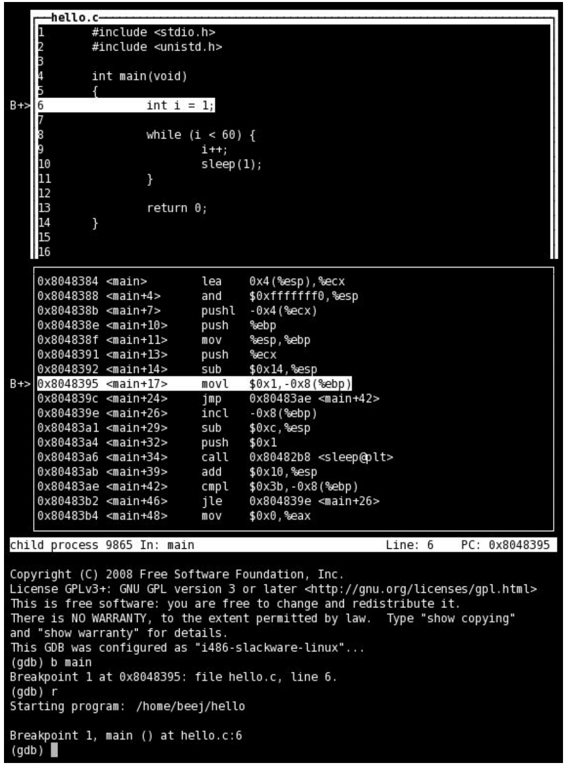
Debugging

GDB, Valgrind

GDB

- No longer stepping through assembly!
  - Si / Ni -> Step / Next
  - Line / Function Breaks
  - Disas -> List
  - Print <var_name>

- Use gdbtui (layout src)
  - Nice display for viewing source/executing commands



```
hello.c
    1       #include <stdio.h>
    2       #include <unistd.h>
    3
    4       int main(void)
    5       {
B+> 6               int i = 1;
    7
    8               while (i < 60) {
    9                       i++;
    10                      sleep(1);
    11              }
    12
    13              return 0;
    14      }
    15
    16
```

```
    0x8048384 <main>        lea     0x4(%esp),%ecx
    0x8048388 <main+4>      and     $0xfffffff0,%esp
    0x804838b <main+7>      pushl   -0x4(%ecx)
    0x804838e <main+10>     push    %ebp
    0x804838f <main+11>     mov     %esp,%ebp
    0x8048391 <main+13>     push    %ecx
    0x8048392 <main+14>     sub     $0x14,%esp
B+> 0x8048395 <main+17>     movl    $0x1,-0x8(%ebp)
    0x804839c <main+24>     jmp     0x80483ae <main+42>
    0x804839e <main+26>     incl    -0x8(%ebp)
    0x80483a1 <main+29>     sub     $0xc,%esp
    0x80483a4 <main+32>     push    $0x1
    0x80483a6 <main+34>     call    0x80482b8 <sleep@plt>
    0x80483ab <main+39>     add     $0x10,%esp
    0x80483ae <main+42>     cmpl    $0x3b,-0x8(%ebp)
    0x80483b2 <main+46>     jle     0x804839e <main+26>
    0x80483b4 <main+48>     mov     $0x0,%eax
```

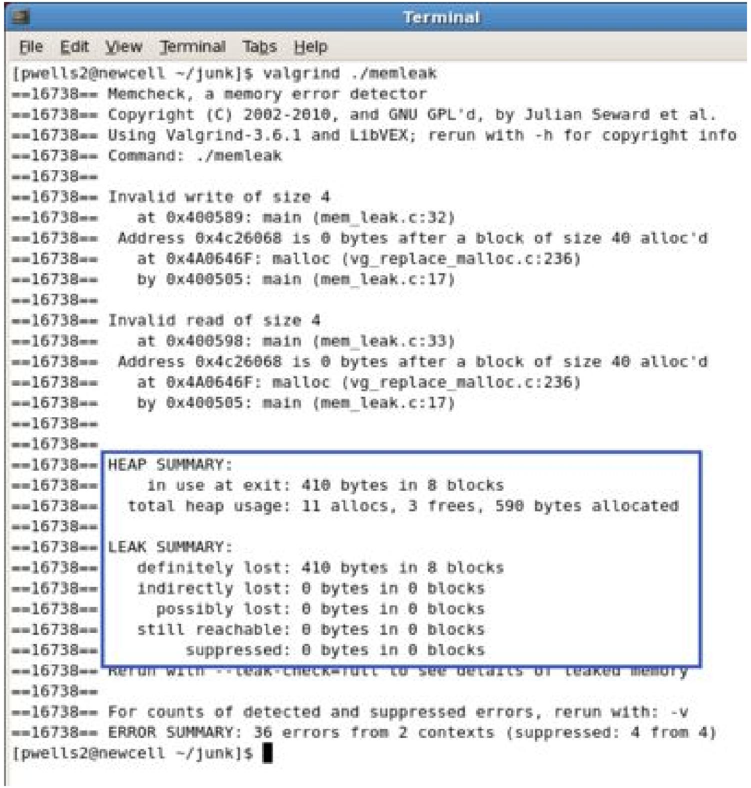
```
child process 9865 In: main                              Line: 6    PC: 0x8048395

Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
(gdb) b main
Breakpoint 1 at 0x8048395: file hello.c, line 6.
(gdb) r
Starting program:  /home/beej/hello

Breakpoint 1, main () at hello.c:6
(gdb) 
```

Valgrind

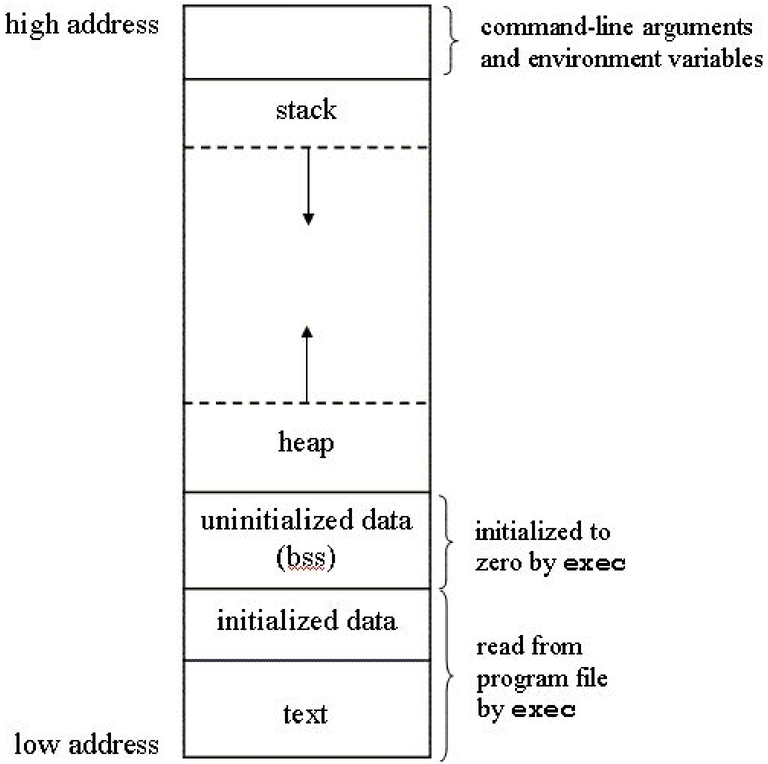
- Find memory errors, detect memory leaks
- Common errors:
  - Illegal read/write errors
  - Use of uninitialized values
  - Illegal frees
  - Overlapping source/destination addresses
- Typical solutions
  - Did you allocate enough memory?
  - Did you accidentally free stack variables/something twice?
  - 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



```
[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==
==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== 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]$
```

Appendix

C Program Memory Layout



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!

C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.

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

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

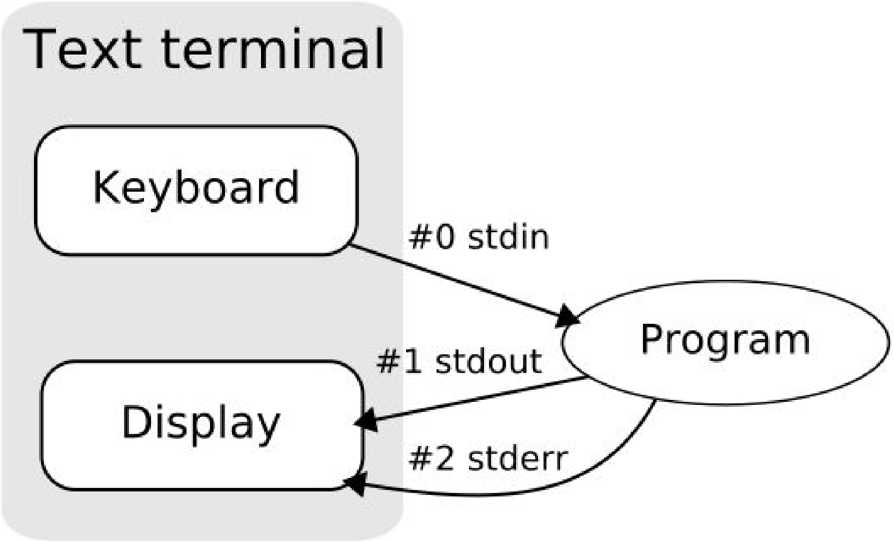
- Searching:
  - `char* strstr (char * str1, char * str2)`: return pointer to *first* occurrence of str2 in str1, else NULL
  - `char* strtok (char * str, char * delimiters)`: tokenize `str` according to delimiter characters provided in `delimiters`, return the next token per successive stroke call, 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!

```
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)