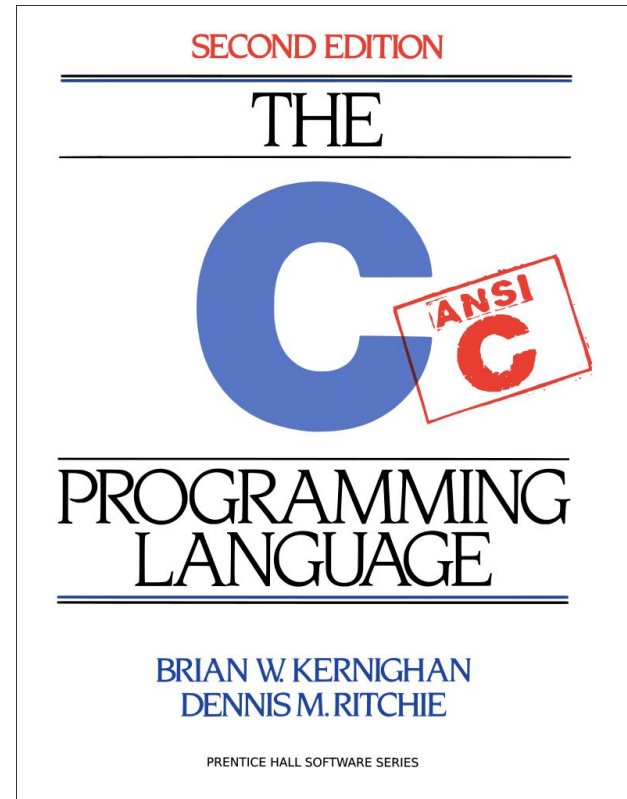


C Boot Camp

June 15, 2018

Fernando
Yinglan
Zach
Kevin



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
```

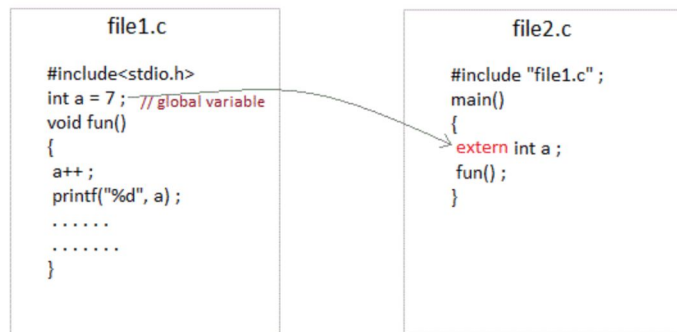
- Contains useful, self-contained C examples
- Slides relating to these examples will have the file names in the **top-right corner!**

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

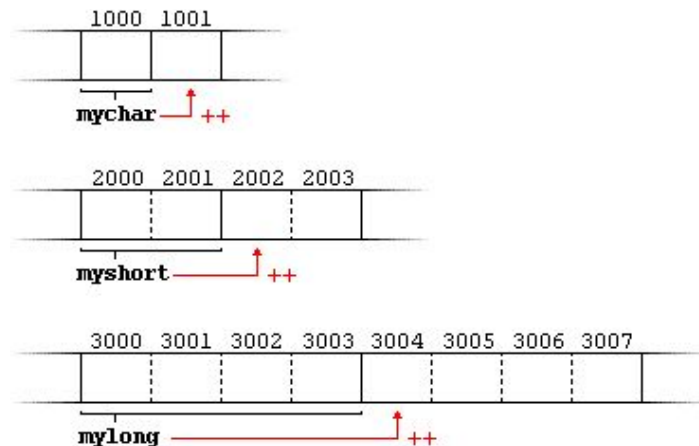
- Can convert a variable to a different type
- Rules for Casting Between Integer Types
- Integer Casting:
 - Signed \leftrightarrow Unsigned: Keep Bits - Re-Interpret
 - Small \rightarrow Large: Sign-Extend MSB, preserve value
- Cautions:
 - Cast Explicitly: `int x = (int) y` instead of `int x = y`
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

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

./pointer_arith

- 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 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 $(A + x * \text{sizeof}(\text{TYPE_OF_PTR_A}))$.
- 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

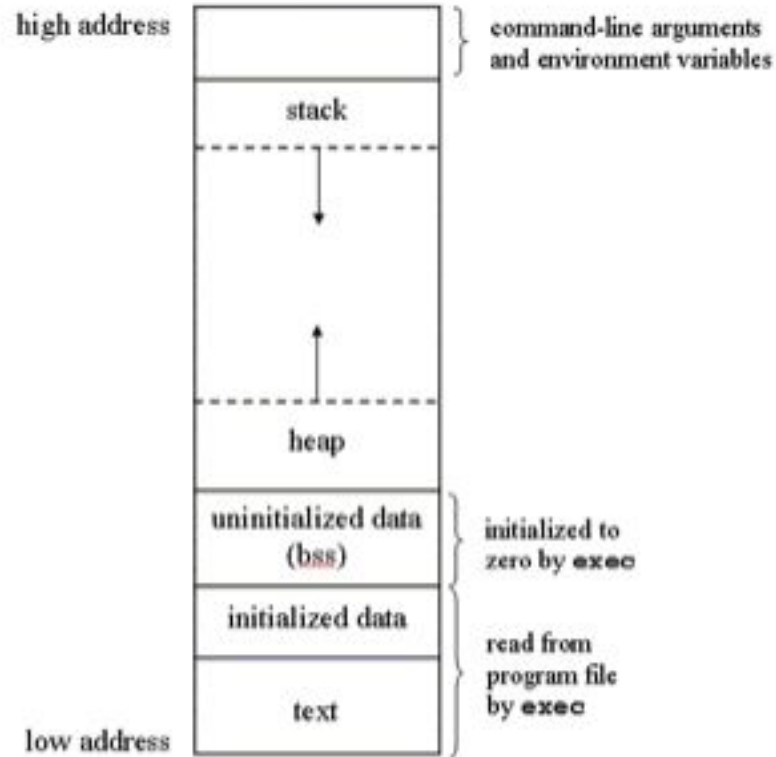
```
void swap(int* a, int* b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
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 / ni` → `step / next`
 - `break file.c:line_num`
 - `disas` → `list`
 - `print <any_var_name>` (in current frame)
 - `frame` and `backtrace` still useful!
- 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

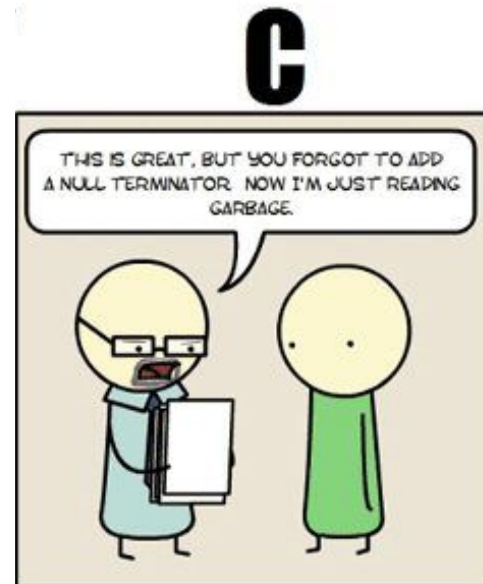
- A way to replace a name with its macro definition
 - No function call overhead, type neutral
 - Think “find and replace” like in a text editor
- Uses:
 - defining constants (INT_MAX, ARRAY_SIZE)
 - defining simple operations (MAX(a, b))
 - 122-style contracts (REQUIRES, ENSURES)
- Warnings:
 - Use 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

- Used heavily in shell/proxy labs
- Reminders:
 - 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>: Dealing with memory

- `void *memset (void *ptr, int val, size_t n);`
 - Starting at `ptr`, write `val` to each of `n` bytes of memory
 - Commonly used to initialize a value to all 0 bytes
 - Be careful if using on non-char arrays
- `void *memcpy (void *dest, void *src, size_t n);`
 - Copy `n` bytes of `src` into `dest`, returns `dest`

Whenever using these functions, a `sizeof` expression is in order, since they only deal with lengths expressed in **bytes**. For example:

```
int array[32];
memset(array, 0, sizeof(array));
memset(array, 0, 32 * sizeof(array[0]));
memset(array, 0, 32 * sizeof(int));
```

<string.h>: Copying and concatenating strings

Many of the string functions in <string.h> have “n” versions which read at most n bytes from `src`. They can help you avoid buffer overflows, but their behavior may not be intuitive.

- `char *strcpy (char *dest, char *src);`
`char *strncpy (char *dest, char *src, size_t n);`
 - Copy the string `src` into `dest`, stopping once a `'\0'` character is encountered in `src`. Returns `dest`.
 - **Warning:** `strncpy` will write at most `n` bytes to `dest`, including the `'\0'`. If `src` is more than `n-1` bytes long, `n` bytes will be written, but no `'\0'` will be appended!

<string.h>: Concatenating strings

On the other hand, `strncat` has somewhat nicer semantics than `strncpy`, since it always appends a terminating `'\0'`. This is because it assumes that `dest` is a null-terminated string.

- `char *strcat (char *dest, char *src);`
`char *strncat (char *dest, char *src, size_t n);`
- Appends the string `src` to end of the string `dest`, stopping once a `'\0'` character is encountered in `src`. Returns `dest`.
- *Make sure `dest` is large enough to contain both `dest` and `src`.*
- `strncat` will read at most `n` bytes from `src`, and will append those bytes to `dest`, followed by a terminating `'\0'`.

<string.h>: Comparing strings

- `int strcmp(char *str1, char *str2);`
`int strncmp (char *str1, char *str2, size_t n);`
- Compare `str1` and `str2` using a lexicographical ordering. Strings are compared based on the ASCII value of each character, and then based on their lengths.
- `strcmp(str1, str2) < 0` means `str1` is less than `str2`, etc.
- `strncmp` will only consider the first `n` bytes of each string, which can be useful even if you don't care about buffer overflows.

<string.h>: Miscellaneous

- `char *strstr (char *haystack, char *needle);`
 - Returns a pointer to first occurrence of `needle` in `haystack`, or `NULL` if no occurrences were found.

- `char *strtok (char *str, char *delimiters);`
 - Destructively tokenize `str` using any of the delimiter characters provided in `delimiters`.
 - Each call returns the next token. After the first call, continue calling with `str = NULL`. Returns `NULL` if there are no more tokens.

- `size_t strlen (const char *str);`
 - Returns the length of the string `str`.
 - Does not include the terminating `'\0'` character.

What's wrong?

```
char *copy_string(char *in_str) {  
    size_t len = strlen(in_str);  
    char *out_str = malloc(len * sizeof(char));  
    strcpy(out_str, in_str);  
    return out_str;  
}
```


What's wrong?

```
char *copy_string(char *in_str) {  
    size_t len = strlen(in_str);  
    char *out_str = malloc((len + 1) * sizeof(char));  
    strcpy(out_str, in_str);  
    return out_str;  
}
```

- `malloc` should be paired with `free` if possible
- **One-byte buffer overflow**

<stdlib.h>: General Purpose Functions

- `int atoi(char *str);`
 - Parse string into integral value
 - Returns 0 on failure...

- `int abs(int n);`
 - Returns absolute value of `n`
 - **See also:** `long labs(long n);`

- `void exit(int status);`
 - Terminate calling process
 - Return `status` to parent process

- `void abort(void);`
 - Aborts process abnormally

<stdlib.h>: What's a `size_t`, anyway?

- Unsigned type used by library functions to represent **memory sizes**
- `ssize_t` is its signed counterpart (often used for -1)
- Machine word size: 64 bits on Shark machines
- `int` may not be able to represent size of large arrays

`warning:` comparison between signed and unsigned integer expressions [-Wsign-compare]

```
for (int i = 0; i < strlen(str); i++) {  
    ^
```

More standard library friends

`<stdbool.h>`

- `bool`

`<stdint.h>`

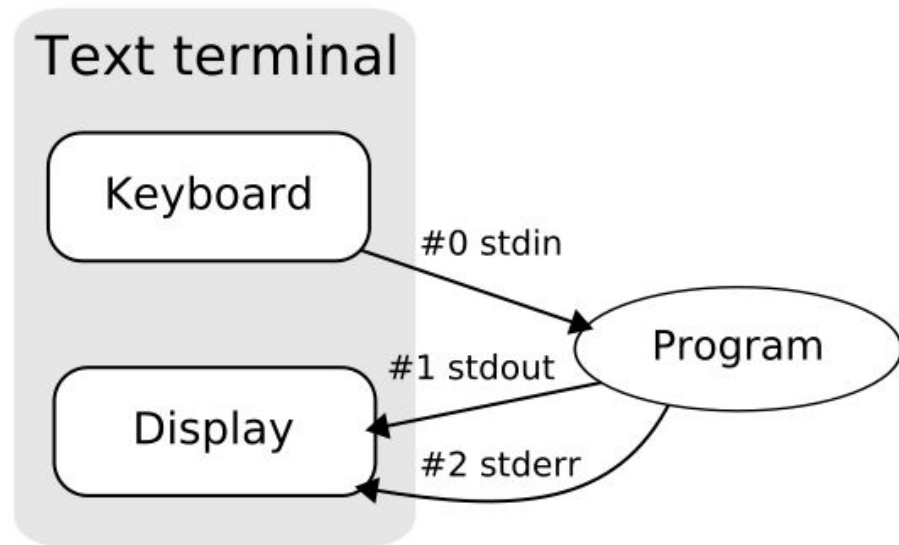
- `SIZE_MAX`, `INT_MIN`, **etc**

`<assert.h>`

- `void assert(scalar expression);`
 - Aborts program if `expression` evaluates as false
 - 122 wasn't completely useless!

<stdio.h>: C standard library I/O

- Used heavily in cache/shell/proxy labs
- Functions:
 - argument parsing
 - file handling
 - input/output
- `printf`, a fan favorite, comes from this library!



<stdio.h>: File I/O

- `FILE *fopen (char *filename, char *mode);`
 - Open the file with specified filename
 - Open with specified `mode` (read, write, append)
 - Returns file object, or `NULL` on error

- `int fclose (FILE *stream);`
 - Close the file associated with `stream`
 - Returns `EOF` on error

- `char *fgets (char *str, int num, FILE *stream);`
 - Read at most `num-1` characters from `stream` into `str`
 - Stops at newline or `EOF`; appends terminating `'\0'`
 - Returns `str`, or `NULL` on error

<stdio.h>: scanf and friends

```
int scanf (char *format, ...);  
int fscanf (FILE *stream, char *format, ...);  
int sscanf (char *str, char *format, ...);
```

- Read data from `stdin`, another file, or a string
- Additional arguments are **memory locations** to read data into
- `format` describes types of values to read
- Return number of items matched, or `EOF` on failure

<stdio.h>: printf and friends

```
int printf (char *format, ...);  
int fprintf (FILE *stream, char *format, ...);  
int sprintf (char *str, char *format, ...);  
int snprintf (char *str, size_t n, char *format, ...);
```

- Write data to `stdout`, a file, or a string buffer
- `format` describes types of argument values
- Returns number of characters that would be written by the string (unless truncated in the case of `snprintf`)

<stdio.h>: Format strings crash course

Placeholders

- **%d**: signed integer
- **%u**: unsigned integer
- **%x**: hexadecimal
- **%f**: floating-point
- **%s**: string (char *)
- **%c**: character
- **%p**: pointer address

Size specifiers

Used to change the size of an existing placeholder.

- **h**: short
- **l**: long
- **ll**: long long
- **z**: size_t

For example, consider these modified placeholders:

- **%ld** for long
- **%lf** for double
- **%zu** for size_t

What's wrong?

```
int parse_int(char *str) {  
    int n;  
    sscanf(str, "%d", n);  
    return n;  
}
```

```
void echo(void) {  
    char buf[16];  
    scanf("%s", buf);  
    printf(buf);  
}
```

What's wrong?

```
int parse_int(char *str) {
    int n;
    sscanf(str, "%d", &n);
    return n;
}
```

- Don't forget to pass pointers to `scanf`, not uninitialized values!
- At least checking return value of `scanf` tells you if parsing failed – which you can't do with `atoi`

```
void echo(void) {
    char buf[16];
    scanf("%15s", buf);
    printf("%s", buf);
}
```

- Avoid using `scanf` to read strings: buffer overflows.
- Need room for null terminator
- Never pass a non-constant string as the format string for `printf`!

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