61C/C Review Slides

Presented by Alan Ton, Akshat Gokhale, and Taj Shaik

Based off of Nick Riasanovsky's Presentation from Fall 2019

Disclaimer

- These slides are not a comprehensive overview of everything you need from 61C to succeed in this class
  - We only have 2 hours!
  - Other concepts not mentioned will likely be brought up when relevant
- This class has A LOT of C coding
  - If you choose to take this class you are committing to that workload
  - Almost everyone taking this class is rusty in C but this review alone with not make you comfortable enough for this class
- We will go through questions for this review, you can find the code in your student VM

Outline

- For each topic we will do a brief conceptual review and then do some practice problems
- Topics List:
  - C Basics (< 15 minutes): types, truthiness, sizeof
  - C Pointers (< 30 minutes): pointers, arrays, strings
  - C Memory (< 15 minutes)
  - C Data Structures (< 15 minutes): structs, typedef, linked lists
  - Useful Functions (< 15 minutes): File I/O, libc
  - Some More Advanced C (< 15 minutes)
  - x86 and RISC-V Review (Any Leftover Time)

Truthiness in C

- In C, the only false values are things that evaluate to 0
  - 0, NULL, false (w/ #include <stdbool.h>)
- All other values evaluate to True
- It's not uncommon in C to see while (1) { … }
  - This is an infinity loop in C and common practice

Types

Types in C

- C is a weakly/statically typed language
- All types are numerical or a composition of other types:
  - Ex. int, char, struct, union, typedef, pointer
- Every variable in C has a type:
  - Ex: int i;
- All variables in C are just bytes under the hood with some length
  - Ex: int32_t is a 4 byte integer interpreted as a signed 2's complement number
  - Ex: uint32_t is the unsigned version of the above
- There is no built-in bool
  - Must import stdbool.h with `#include <stdbool.h>`

Sizeof

- In C, many types don’t have defined sizes
    - Ex: how many bits in an int are defined on a per system basis
- In C, we use sizeof to determine the size
    - Ex: sizeof (int) == # bytes in an integer
- sizeof is important to use for memory allocation
    - Malloc calls should include sizeof somewhere

Pointers

Memory Manipulation.

Pointers in C

- Contiguous memory is represented in C pointers
    - Pointers are a less generic abstraction for addresses
- Pointers are denoted with a `*`
    - `int *` means a pointer to an integer
    - `char **` means a pointer to a pointer to a character
- NULL is used as the value for an invalid pointer
- Just passing an address doesn't make it valid
    - Using memory that is not legal/in scope leads to program crashes (segfaults)

How to use pointers

Operators:

1. address-of (&): returns memory address of the variable
2. dereference (*): returns value pointed to by a pointer

int my_value = 8;
int *my_pointer = _______________ ;
int **my_db_pointer = ____________ ;
	int *my_pointer = 0x5C3EFF;
	int my_value = ______________ ;
	
	int **my_db_pointer = 0x7FFE0A;
	int my_value = _______________ ;

How to use pointers

Operators:

1. address-of (&): returns memory address of the variable
2. dereference (*): returns value pointed to by a pointer

int my_value = 8;
int *my_pointer = &my_value;
int **my_db_pointer = &my_pointer;
	int *my_pointer = 0x5C3EFF;
	int my_value = ________________ ;
	
	int **my_db_pointer = 0x7FFE0A;
	int my_value = _________________ ;

How to use pointers

Operators:

1. address-of (&): returns memory address of the variable
2. dereference (*): returns value pointed to by a pointer

int my_value = 8;
int *my_pointer = &my_value;
int **my_db_pointer = &my_pointer;
	int *my_pointer = 0x5C3EFF;
	int my_value = *my_pointer;
	
	int **my_db_pointer = 0x7FFE0A;
	int my_value = ________________ ;

How to use pointers

Operators:

1. address-of (&): returns memory address of the variable
2. dereference (*): returns value pointed to by a pointer

int my_value = 8;
int *my_pointer = &my_value;
int **my_db_pointer = &my_pointer;
	int *my_pointer = 0x5C3EFF;
	int my_value = *my_pointer;
	
	int **my_db_pointer = 0x7FFE0A;
	int my_value = **my_db_pointer;

Why do we use pointers?

Use cases:

1. Arrays
2. Strings
3. Writable Function Parameters
4. Functions (as arguments)

Arrays in C

- Arrays are a contiguous region of memory of fixed size
- Referenced by a pointer to their first element
- Access elements via pointer arithmetic (`a[i] == *(a + i)`)
- Arrays don't have an end marker
  - programmer's responsibility to keep track of the size of an array
  - exception: strings in C (char arrays) are terminated by a '\0'

Ex:

`int lottery_numbers[3] = {62, 55, 30};` (pointer w/ space for 3 integers)

`int x = lottery_numbers[0];` (equivalent to: `*(lottery_numbers + 0)`)

`int y = lottery_numbers[2];` (equivalent to: `*(lottery_numbers + 2)`)

Pointer Arithmetic

`int lottery_numbers[3] = {62, 55, 30};` (pointer w/ space for 3 integers)

`int x = lottery_numbers[0];` (equivalent to: `*(lottery_numbers + 0)`)

`int y = lottery_numbers[2];` (equivalent to: `*(lottery_numbers + 2)`)

Q: `*(lottery_ numbers + 2)` or `*(lottery_numbers + 2*sizeof(int))`?

Pointer Arithmetic

`int lottery_numbers[3] = {62, 55, 30};` (pointer w/ space for 3 integers)

`int x = lottery_numbers[0];` (equivalent to: `*(lottery_numbers + 0)`)

`int y = lottery_numbers[2];` (equivalent to: `*(lottery_numbers + 2)`)

Q: `*(lottery_ numbers + 2)` or ~~`*(lottery_numbers + 2*sizeof(int))`~~?

A: Compiler knows to multiply 2 by the sizeof(int)!!

Pointers to Arrays

`int lottery_numbers[3] = {62, 55, 30};` (pointer w/ space for 3 integers)

Q: Which of the following gives a pointer to the `lottery_numbers` array?

```
void *ptr_one = lottery_numbers;
void *ptr_two = &lottery_numbers;
void *ptr_tre = &lottery_numbers[0];
```

Pointers to Arrays

`int lottery_numbers[3] = {62, 55, 30};` (pointer w/ space for 3 integers)

Q: Which of the following gives a pointer to the `lottery_numbers` array?

```
void *ptr_one = lottery_numbers;
void *ptr_two = &lottery_numbers;
void *ptr_tre = &lottery_numbers[0];
```

A: All of them are correct and equivalent to each other!

C Basics

The following code has undefined behavior. Identify all of the following bugs in the code.

C Basics

basics.c

```c
// Print out all the elements of
// the array,
// each element on a newline
void print_array (int* arr) {
    while (*arr != NULL) {
        printf ("%d\n", *arr);
        arr += sizeof (int);
    }
}
```

```c
int main () {
    // Array intended to consist of
    // 1, 2, 0, 5
    int a[] = {1, 2, 0, 5, NULL};
    // Should print:
    // 1
    // 2
    // 0
    // 5
    print_array (a);
}
```

C Basics

basics.c

```c
// Print out all the elements of
// the array,
// each element on a newline
void print_array
(int* arr, size_t size) {
    int *endpoint = arr + size;
    while (arr < endpoint) {
        printf ("%d\n", *arr);
        arr += 1;
    }
}
```

```c
int main () {
    // Array intended to consist of
    // 1, 2, 0, 5
    // No need for NULL
    int a[] = {1, 2, 0, 5};
    // Should print:
    // 1
    // 2
    // 0
    // 5
    print_array (a, 4);
}
```

Strings in C

- A string in C is just an array of characters (type is char *)
  - A proper string always ends with a null terminator '\0'
- C Library functions assume proper strings (e.g. strlen, strcpy, strcmp)
  - very unsafe assumption, see buffer overflow
- Functions that have length as a parameter are safer
  - ex) strncpy, strncat, etc.

Writeable Function Parameters

- When a function is called, its parameters are *copied* onto its stack
- Changes to these values will be lost when the function returns

- Solution:
    - pass in a pointer as a parameter
    - function receives copy of pointer
    - function uses this pointer to access/edit the contents it points to
    - the changes will persist after the function returns

Function Pointers

- C can also pass around functions to provide more generic functionality
- This is done through function pointers, which have a really gross syntax

Ex:

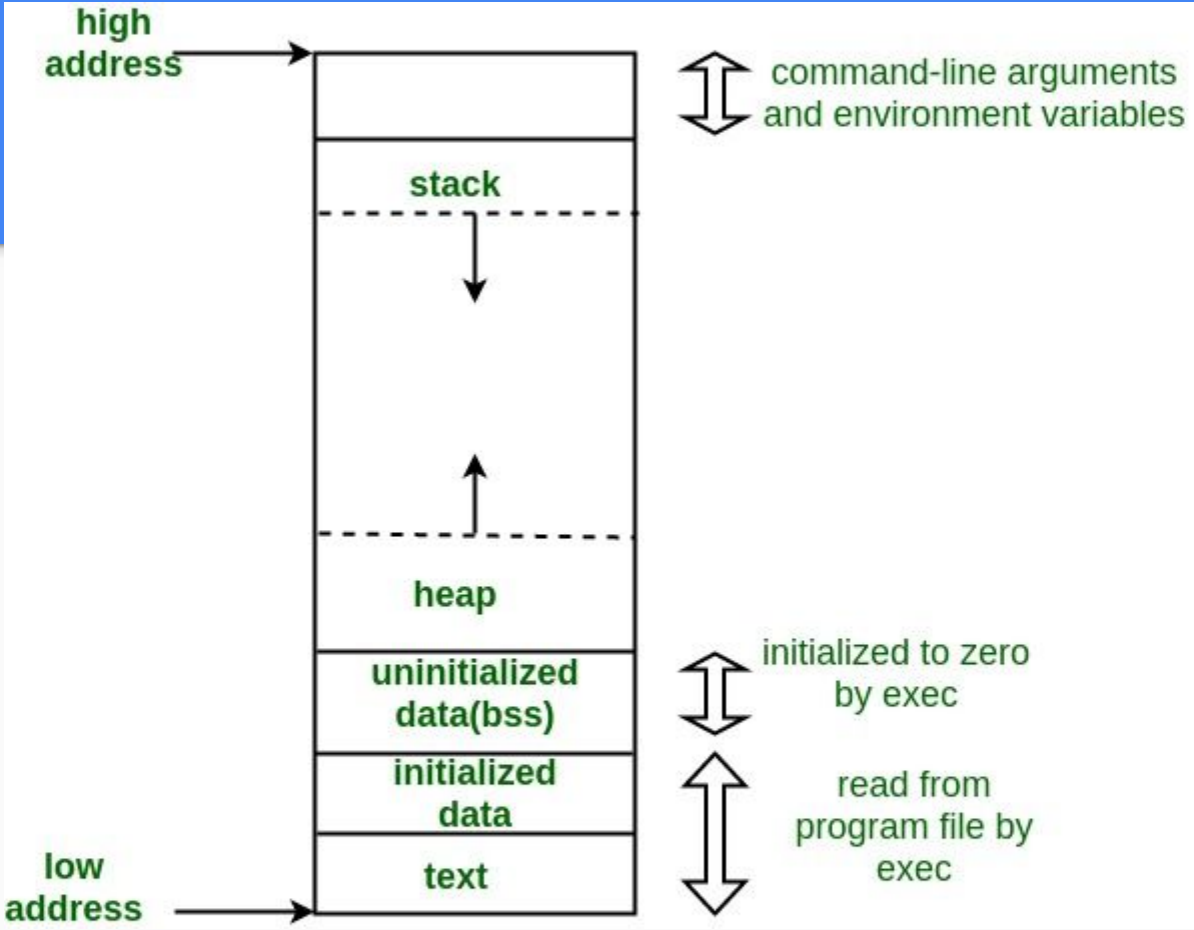
// Pointer to function f void (*f_ptr)(int) = &f	// Function f void f(int a) { … }

This syntax declares the variable f_ptr, sets its type as pointer to a function with a return value of void and parameter int and then initializes it to point to f.

C Memory Layout

Pointers require an understanding of memory regions.

Memory Layout



Memory Layout (explained)

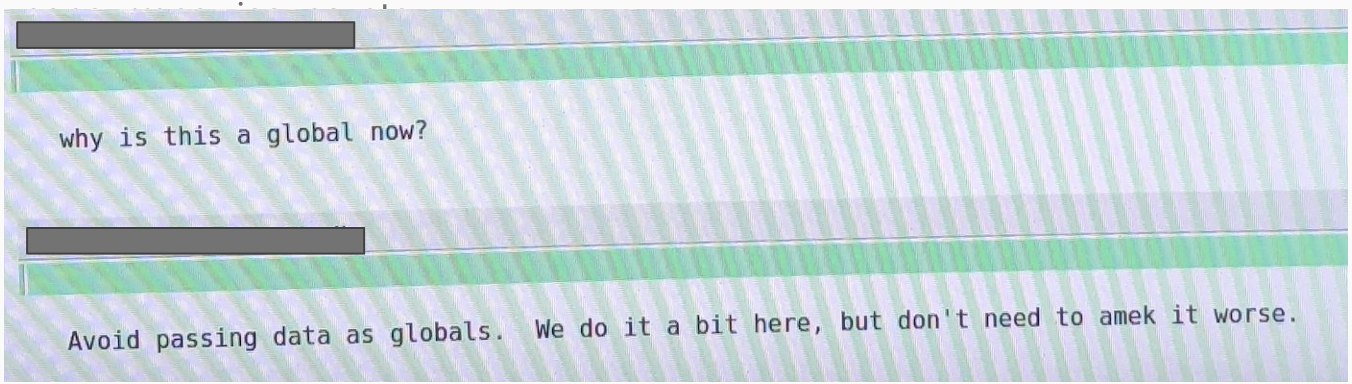
- Text: Actual instructions of the Program
- Data (+ BSS): Statically allocated data (global variables, Strings, constants)
- Stack: Local variables for each function call
- Heap: Dynamic memory that persists beyond function calls (malloc)

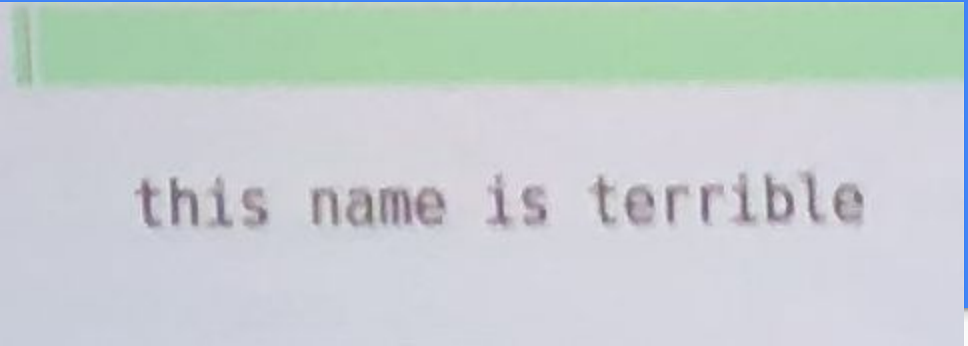
C Memory (Global)

- Global variables can be accessed by all functions and exist throughout the duration of a program's lifetime
- Convenient, but dangerous
  - no access control, namespace pollution, testing/confinement issues (hard to unit test), concurrency issues, etc.
  - bad practice in large-scale software engineering projects

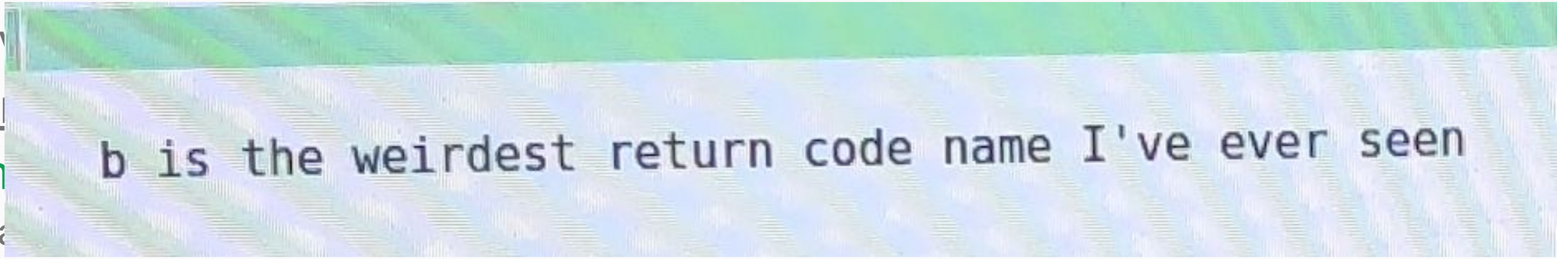
C Memory (Global)

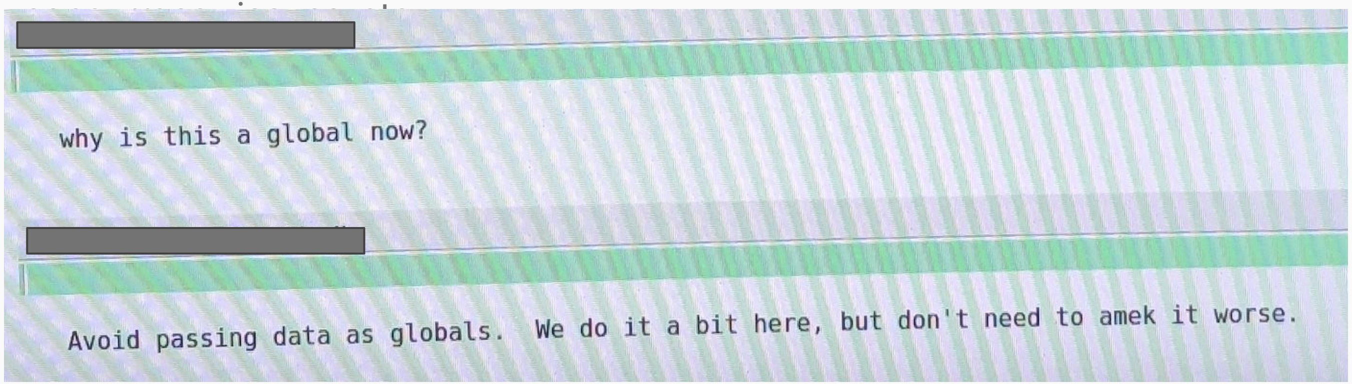
- Global variables can be <u>accessed by all functions</u> and <u>exist throughout the duration of a program's lifetime</u>
- Convenient, but dangerous
  - no access control, namespace pollution, testing/confinement issues (hard to unit test),





- Global v
  duratio
- Conven
  - no a





C Memory (Stack)

- Stores local arguments and function parameters in a stack frame
- When a function is invoked, a stack frame is pushed to the stack
- When a function returns, its stack frame is removed from the stack

If you need variables to persist across function calls,

do not store them on the stack!!

Instead, make them global variables OR store them on the heap.

C Memory (Heap)

- Heap data is requested with the alloc series of functions

- `void* malloc (size_t nbytes)`
  - Return a pointer to n bytes of data
- `void* calloc (size_t elemsize, size_t nelems)`
  - Return a pointer to elemsize * nelems bytes of data
- `void* calloc (void *ptr, size_t nbytes)`
  - Return a pointer to n bytes of data resizing the existing ptr (possibly moving it)
- Memory is returned with `free (void *ptr)`

C Memory (Heap)

- Heap data is requested with the alloc series of functions

WARNING: calloc provides null-terminators (because it zeros out all allocated memory); however, malloc **does not** provide null-terminators.

If you want to malloc space for a string, be sure to request one additional byte for the null-terminator.

- Memory is returned with `free (void *ptr)`

C Memory

In the following program we have provided 5 print statements. State which print statements will not always succeed and why. Assume all necessary includes.

C Memory

`memory.c`

```c
char global[] = {'h', 'e', 'l', 'l', '0'};
int main () {
        printf ("%s\n", f1());
        printf ("%s\n", f2());
        printf ("%s\n", f3());
        printf ("%s\n", f4());
        printf ("%s\n", f5());
}
```

```c
char *f5 () {
        char *arr = calloc (strlen ("hello") + 1
                                            , sizeof (char));
        for (int i = 0; i < 5; i++) {
                arr[i] = "hello"[i];
        }
        return arr;
}
```

```c
char *f1 () {
        return "hello";
}
```

```c
char *f3 () {
        char hello[] = "hello";
        return hello;
}
```

```c
char *f2 () {
        return global;
}
```

```c
char *f4 () {
        return malloc (strlen ("hello") + 1);
}
```

C Memory

memory.c

```c
char global[] = {'h', 'e', 'l', 'l', '0'};
int main () {
        printf ("%s\n", f1());
        printf ("%s\n", f2());
        printf ("%s\n", f3());
        printf ("%s\n", f4());
        printf ("%s\n", f5());
}
```

```c
char *f5 () {
        char *arr = calloc (strlen ("hello") + 1
                                    , sizeof (char));
        for (int i = 0; i < 5; i++) {
                arr[i] = "hello"[i];
        }
        return arr; // calloc adds null terminator
}
```

```c
char *f1 () {
        // string literals stored in data segment
        return "hello";
}
```

```c
char *f3 () {
        char hello[] = "hello";
        return hello; // Cannot return stack array
}
```

```c
char *f2 () {
        return global; // No null terminator
}
```

```c
char *f4 () {
        return malloc (strlen ("hello") + 1); // No null term
}
```

Data Structures in C

structs, typedefs.

structs

- structs allow us to create groups of different types (arrays, int, char)
- struct syntax
    - `struct <name> {}`
- instantiate structs using the {}
- Use dot-notation to access the attributes of a struct
- Use arrow-notation to access attributes of struct pointer
- structs DO NOT have methods

Memory Layout of a struct

```
struct ListNode {
    int val; // val = 0x12345678
    struct ListNode* next; // next = 0xdeadbeefdeadbeef
}
```

- Naively, the layout of a struct looks like this
  - 0x7fffffffe350: 0x12345678 0xdeadbeef
  - 0x7fffffffe354: 0xdeadbeef 0x00000000
- Actually, the layout of a struct looks like this
  - 0x7fffffffe350: 0x12345678 0x00007fff
  - 0x7fffffffe354: 0xdeadbeef 0xdeadbeef

Ex: Declaring a Struct

```
// struct syntax: struct <name>
struct ListNode {
    // insert attributes in {}
    char* value;
    struct ListNode* next; // can reference pointers
}
```

Ex: Working w/ Structs

```
int main() {
    struct ListNode x = {"hello world" ,0}; // Using {}
    x.value = "CS162";                      // Use dot notation
    ListNode *y = &x;          // Code pointer to struct
    y->value = "operating systems"; // Use arrow notation
}
```

Typedef

- typedef creates a new type that has the exact same structure as a data type
- Syntax for typedef
  - typedef <data type name> <new data type name>
- Commonly used to create new types from structs

Ex: Using typedef

```
struct ListNode {
    char* value;
    struct ListNode* next; // can reference pointers
};

typedef struct ListNode LinkNode;
/* if typedef not included, struct would be declared as
struct ListNode */
```

`types.h`

`off_t`: **signed integer**, used for file sizes
`pid_t`: **signed integer**, used for process IDs
`pthread_t`: **unsigned integer**, used to identify a thread
`size_t`: **unsigned integer**, used for sizes of objects

Working with C Data Structures! `data-structures.c`

Complete the example function that removes all elements from our list of strings that contain str as their value. This must be done in place, producing a valid pointer to the first node at the memory address passed into the function. Assume all nodes have been malloced and must be freed.

Working with C Data Structures! `data-structures.c`

```
void remove_nodes (LinkNode **node_addr, char *str) {
    while (*node_addr != NULL) { // Iterate through list
        if (!strcmp((*node_addr)->value, str)) { // Is a match
            // Your code here
        } else {
            // Your code here
        }
}
```

Working with C Data Structures! (sol'n)

data-structures.c

```
void remove_nodes (LinkNode **node_addr, char *str) {
      while (*node_addr != NULL) { // Iterate through list
      if (!strcmp((*node_addr)->value, str)) { // 0 is a match
          LinkNode *to_free = *node_addr;
          *node_addr = to_free->next; // Change the pointer
          free (to_free);
      } else {
          node_addr = &(*node_addr)->next; // Make next changes
      }
   }
```

libc

string manipulation, file i/o

Useful Functions to Know

- strlen - returns the length of a string (not including the null terminator)
- strcpy - copies the characters from src string to dest string
- strcmp - compares two strings lexicographically; returns an integer
- fprintf - print formatted strings to a specified file
- fopen - opens a FILE *
- fclose - close a FILE *
- fread - read contents from a FILE *
- fwrite - write contents to a FILE *

File I/O Example

```
int write_char(char *infile) {
    FILE *f = fopen(infile, "r+");
    if (f != NULL) {
      char buf[1];
      size_t chars_read = fread(buf, 1, 1, f);
      printf("The character is: %c\n", buf[0]);
      size_t chars_written = fwrite(buf, 1, 1, f);
      return fclose(f);
    }
    return 1;
}
```

```
(void *ptr, size_t size,
size_t nmemb, FILE *stream)
```

strcpy Implementation

```
char *strcpy(char *dest, const char *src){
    if (dest == NULL) {
        return NULL;
    }
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    return dest;
}
```

strcpy Implementation (sol'n)

```
char *strcpy(char *dest, const char *src) {
    if (dest == NULL) {
        return NULL;
    }
    char *ptr = dest;
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
    return ptr;
}
```

strlen Implementation

```
long int strlen_staff(const char *src) {

    /* INSERT CODE HERE */

}
```

strlen Implementation (sol'n)

```
long int strlen_staff(const char *src) {
    long int count = 0;
    while(*src != '\0') {
        count++;
        src += 1;
    }
    return count;
}
```

fprintf formatting

- Prints a formatted string to the specified `FILE *`
- For each conversion specification (%[char]), provide an argument to be printed
  - `fprintf(file1, "Line %d: %s", 1, "Segmentation Fault")`
- Some common specifications: %d [decimal], %u [unsigned decimal], %c [character], %s [string], %f [double]
- `printf(...) = fprintf(stdout, ...)`

Advanced C

type casting, preprocessor guards.

Type Casting

- Types can be converted between with casting, either implicit or explicit
  - Ex: unsigned int i = -1;
  - Ex: long s = (unsigned int) -1;
- C does not have the concept of generics the way other languages do
- Instead C uses void* and char* to generalize pointer types and modify pointers at the per byte level
- It's easy to cast to fix compilation errors but break your program as a result

Preprocessor Guards

- Many files often import the same .h file to get the same definitions
- To prevent against multiple includes we use include guards
    - If we don't have this we risk an error for multiple definitions of the same data type.

For file foo.h

```
#ifndef FOO_H
# define FOO_H
…
#endif
```

Global/Static Variables

- Real C programs are split across many files
- The keyword `static` is used to state a variable can only be used in the current file
  - `static int x = 7;`

- The keyword `extern` can be used to declare a variable, but not define it
  - in other words: declare that this variable exists, but is defined in another file

  - `int x;` // declaration and definition (memory allocated for one int that holds default value 0)
  - `extern int y;` // declaration, but no definition (i.e. memory isn't allocated for it in this file)

Global, global, global

Assume file advanced1.c contains an int global variable, `global` that we want to modify in advanced2.c. How can we modify advanced2.c to allow us to modify `global.` What should we do if many different files need to access `global?`

Global, global, global (sol'n)

Place a declaration at the top of advanced2.c that reads

extern int global;

If practice we should put this in advanced1.h file in case the definition changes or if many files want to use.

```
#ifndef ADVANCED1_H
#define ADVANCED1_H
extern int global; ….
#endif
```

Preprocessor Directives

- the preprocessor is invoked before compilation
- takes action on any statement that starts with a `#` (these are called preprocessor directives)
- Examples:

a. `#define` *`identifier`* *`replacement`*
  - replaces any occurence of *`identifier`* with *`replacement`*
  - ex. `#define MAX_WORD_LEN 64`

b. `#ifdef` *`identifier`*

  ... `code block` ...

  `#endif`
  - compiles the code block only if *`identifier`* has been `#define'`d
  - can define the `identifier` when compiling

    (ex. `gcc -Didentifier file.c -o a.out`)

c. `#include <header>` and `#include "file"`
  - replaces entire `#include` statement with content of `header` or `file`
  - make sure to not introduce circular dependencies

x86

Expectations: Read and understand, don’t need to write it.

x86 ISA Registers

- pintOS is a 32-bit x86 machine (the student VM uses an x86-64 processor)

eax: store return value of a function	**esi**: often used as a pointer to “source” data
ebx: general-purpose; sometimes used to store constants	**edi**: often used as a pointer to “destination” data
ecx: general-purpose	**esp**: points to top of the stack
edx: general-purpose	**ebp**: stores location of stack at the beginning of a function
	eip: points to address of currently executing instruction (like PC in RISC-V)

x86 ISA Registers

- pintOS is a 32-bit x86 machine (the student VM uses an x86-64 processor)

eax: store return value of a function	**esi**: often used as a pointer to "source" data
ebx: general-purpose; sometimes used to store constants	**edi**: often used as a pointer to "destination" data
ecx: general-purpose	**esp**: points to top of the stack
edx: general-purpose	**ebp**: stores location of stack at the beginning of a function
	eip: points to address of currently executing instruction (like PC in RISC-V)

x86 ISA Syntax

1. Registers preceded by a percent sign
   - ex. %eax for the register eax
2. Immediates preceded by a (ty) dollar $ign
   - ex. $4 for the constant 4
3. *Most* instructions use parentheses to dereference memory addresses
   - ex. (%eax) reads from the memory address in eax
   - Add a constant offset by prefixing the parenthesis
   - ex. 8(%eax) reads from the memory address eax + 8
4. Source operands precede destination operands

x86 ISA Continued

<u>Suffixes</u>
b: 8 bits, 1 byte
w: 16 bits, 2 bytes, 1 word
l: 32 bits, 4 bytes, 2 words, 1 longword

1. addw %ax, %bx
2. addl %eax, %ebx
3. addl (%eax), %ebx
4. addl 12(%eax), %ebx
5. subl $12, %esp

x86 ISA Continued

mov: dst = src
(http://drwho.virtadpt.net/files/mov.pdf)
and: dst = src & dst
or: dst = src | dst
xor: dst = src ^ dst

1. movl %eax, %ebx
2. movl $4, %ecx
3. movl 4, $ecx
4. movl %edx, -8(%ecx)

4.3 Clearing a Register

Write an instruction that clears register `eax` (i.e. stores 0 in `eax`).

4.3 Clearing a Register

Write an instruction that clears register `eax` (i.e. stores 0 in `eax`).

```
xorl %eax, %eax
```

```
subl %eax, %eax
```

```
movl $0, %eax
```

Calling a Function

Caller Steps

1. Push args to the stack in reverse order
    - `pushl %eax, pushl %ebx, …`
    - `pushal`
    - stack *must* be "stack-aligned" (i.e. %esp must be a multiple of 16)
2. Push the return address to the stack and jump to the function
    - `call $0x1234`
    - where 0x1234 is the address of the first instruction in the function
3. When function returns, the return address is gone but the args are still on the stack
    - `popal`

Calling a Function

Callee Steps

1. Push ebp onto the stack, and store current esp into ebp
2. Compute the return value and store it in eax
3. Restore esp to its value before the function started
    - `leave` is equivalent to:

      ```
      movl %ebp, %esp
      popl %ebp
      ```
4. Pop the return address off the stack and jump to it
    - `ret`

4.6 Reading Disassembly

- pushl %eax is equivalent to:

```
subl $4, %esp
movl %eax, (%esp)
```

`call $0x1234`: pushes return address to stack and jumps to specified address (0x1234)

```
callee:
1       pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
2       movl    8(%ebp), %edx
        movl    12(%ebp), %eax
3       addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        addl    $1, %eax
4       leave
        ret
```

leave is equivalent to:

```
movl %ebp, %esp
popl %ebp
```

```
caller:
5       pushl   %ebp
        movl    %esp, %ebp
6       pushl   $4
        pushl   $3
7       call    callee
        addl    $8, %esp
        movl    %eax, global
8       nop
        leave
        ret
```

`ret`: pops a longword (4 bytes) off of the stack (typically a return address) and jumps to it

4.6 Reading Disassembly

prologue: save esp in ebp; make space for local variables

```
callee:
1       pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
2       movl    8(%ebp), %edx
        movl    12(%ebp), %eax
3       addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        addl    $1, %eax
4       leave
        ret
```

read args from stack->register

callee function

epilogue: restore esp, pop off return addy & jump to it

```
caller:
5       pushl   %ebp
        movl    %esp, %ebp
6       pushl   $4
        pushl   $3
7       call    callee
        addl    $8, %esp
        movl    %eax, global
8       nop
        leave
        ret
```

prologue

push args to stack (rev order)

call the function

clean up stack

store result

epilogue: restore esp, pop off return addy & jump to it

4.7 x86 Calling Convention

```
void helper(char* str, int len) {
 char word[len];
 strncpy(word, str, len);
 printf("%s", word);
 return;
}

int main(int argc, char *argv[]) {
 char* str = "Hello World!";
 helper(str, 13);
}
```

4.7 x86 Calling Convention

```
void helper(char* str, int len) {
 char word[len];
 strncpy(word, str, len);
 printf("%s", word);
 return;
}

int main(int argc, char *argv[]) {
 char* str = "Hello World!";
 helper(str, 13);
}
```

13
str
return address
saved ebp
'\0'
'!'
'd'
'l'
'r'
'o'
'W'
' '
'o'
'l'
'l'
'e'
'H'

RISC-V

Brief RISC-V/Assembly Review

- You WILL NOT need to write/work with any RISC-V for this class
- You WILL need to write very little x86 assembly (which we will teach you)
- You WILL need to be able to read some x86 assembly
- To ease that transition we are going to review how some higher level functionality relates to RISC-V
- x86 is covered in detail in discussion THIS WEEK!

Registers, Immediates, and Memory

- Registers are pieces of hardware that store 32-bit values we are using
  - Most instructions rely on registers
  - Some Registers are special
    - Ex: Stack Pointer holds stack bound)
- Immediates are compile-time constants
  - Ex: Offset from an address for accessing a struct elem
- Memory values are for what can't fit in registers or needs to shared across threads
  - May need to move to a register to use the data

Instruction Execution

- The next instruction executed is based on a register holding the address of an instruction
  - Called the PC (Program Counter) in RISC-V
- Most instructions increment the program counter past the instruction just executed
  - In RISC-V this was always 4 bytes
  - In x86 this may be a variable amount (variable length instructions)

Control Flow

- More complicated control flow is done with conditional jumps (branches) and unconditional jumps (jump instructions)
- These instructions directly change the value of the PC
- This is how we enter/leave a function

Function Calling

- When calling a function we must:
  - Load arguments
  - Change the PC
  - Store how to return from the function

Load Arguments

- Before entering a function we need to evaluate and load all arguments
  - In RISC-V this is done with the argument registers (a0-a7) with spillover on the stack
  - In x86 each argument is placed on the stack

Changing the PC

- To change the PC we jump to a different address
  - For most functions this is a jump to a label/constant value
  - Other situations involve jumping to register values holding a function address
    - These are how function pointers in C are executed
- The common instructions in RISC-V are jal (jump to a label) and jalr (jump to a register)
- x86 jumps to functions with CALL instruction

Storing Return Address

- In RISC-V return address are stored in a special register (RA), which are modified by the And Link portion of jal and jalr
  - If that function calls a function then the previous RA is pushed to the stack
- x86 stores this return address directly on the stack
  - Programmer directly “pushes” the value onto the stack

Executing a Function

- When a function executes a frame is allocated by manipulating the stack
    - This consists of modifying the stack pointer (bottom of the stack) and the frame pointer (top of the stack)
- There are also some registers that if changed need to be restored
- When a function finishes all local variables are now out of scope and so we need to restore the stack to the previous values

Function Returns

- To return from a function we return control flow to the return address
    - RISC-V this is JR RA
    - x86 pops off the return address from the stack and calls RET

Quick Conceptual Check

Assume you always save the current return address but forget to save the previous return address. What types of programs can you no longer run?

Quick Conceptual Check

Assume you always save the current return address but forget to save the previous return address. What types of programs can you no longer run?

Any program with more than 2 open frame will not execute (i.e. main calls a function which call a function)