Midterm Review

Dr. Jack Lange

What makes a file executable?

- In UNIX: “Everything is a file”
- For a file to execute it needs to be marked as special
  - Someone needs to grant it permission to execute
  - Every file has a set of permissions that define who can do what with it
- File permissions
  - Set of bits that flag specific permissions for certain users
  - <type (1 flag)><user (3 flags)><group (3 flags)><everyone (3 flags)>
    - E.g. drwxr-xr-x
    - this is a directory that the owner can modify, but everyone else can only read
  - Type
    - ./- = regular file
    - d = directory
  - r = reads allowed
  - w = writes allowed
  - x = execution allowed, or directory can be accessed

What is memory?

- **Memory is a giant array of bytes (8 bits)**
- Each byte is referenced via an index/offset
  - The base index/offset always starts at 0
  - The index of a byte is its "address"
    - Byte with address 9 == memory[9]
  - Hardware interacts with memory in the same way
- How big is the array?
  - Determined by the machines **address size**.
  - Not the amount of memory installed in the system

Variables

- Variables name a **region** of memory, not a single byte location
- Variables have types other than raw bytes
  - char, int, short, float, etc…
- A type defines what the value of a variable **means**, and what value it holds
  - Types can take up more than one byte
    - More than a single entry in the memory array
  - The number of bytes required to represent a variable type is chosen by the compiler
    - There are some loose rules, but this is a source of pain
- `int sizeof();`
  - Built in compiler function that returns the size of a variable's type

Standard primitive types (x86_64)

```
char           1 byte  -128 to 127
unsigned char  1 byte  0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           8 bytes -9223372036854775808 to
                        9223372036854775807
unsigned long  8 bytes 0 to 18446744073709551615
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
pointer        8 bytes
```

Important: These sizes are not always true!!!
It depends on the compiler and underlying architecture

Type encodings

- Remember: C interacts directly with memory
  - So all data is stored directly as raw bits (not objects)
- `char y = 10;`
  `==>> 1 byte of value 0001010`
- Signed vs unsigned
  - Most primitive types can be signed (+/-) or unsigned (+)
  - Unsigned: Raw binary value
  - Signed: 2's complement
    - "Flip the bits and add 1"
    - 1 in the highest order bit location means value is negative

Getting Input for a C program

- 1st mechanism: Command line arguments
  - List of words you typed into command line
  - Available as an array of strings named **`argv`**
    - `char * argv[]`
  - `argv[0]` is always the name of the executable you are running
- Example: cmdargs.c

Interactive Input

- 2nd Mechanism: Console Input
  - Example: **hello-1.c**
- scanf()
  - `int scanf(const char *format, ...);`
  - Waits for input from the console and parses it
    - According to a format string that specifies how to parse input
  - Copies resulting values into memory locations
    - Uses location of variables in memory (& operator)
  - Returns the number of successful matches
    - Used to detect errors (0 == No input parsed)

Interactive Input - Streams

- 3rd mechanism: FIFO Streams
  - **FIFO** = First In, First Out (in order processing)
- Console streams
  - stdin – standard input
  - stdout – standard output
  - stderr – standard error
- Other streams
  - Files, Devices, Network Sockets
  - We'll get to block interfaces later

Easy File I/O

- The UI shell can be used for file I/O
  - Files and streams are the same in C
    - Unix: *Everything is a file*
  - …so we can mix and match easily
- Shell Redirection
  - Redirect console streams
    1. Redirect *stdout* to a file ('>' and '>>')
       - `ls > files.out` -- creates new file "files.out"
       - `ls >> all_files.out` -- appends to file "all_files.out"
    2. Read *stdin* from a file ('<')
       - `wc < files.out`
    3. Pipe *stdout* of one program to *stdin* of another **('|')**
       - `ls -l | sort`
       - `ls | grep .c | wc`

File I/O in programs

- Command line and streams handled by system
- Files you have to manage on your own
  - `FILE * file = fopen(name, <mode>);`
    - On error NULL is returned
  - `void fclose(FILE * file)`
- Text files
  - Once open, a text file **CAN** be treated like a console
  - `fprintf, fscanf` – variants that can work on files

ASCII Files

- ASCII: Encoding that maps bytes to text characters
  - Translates characters from human readable to machine readable
  - ”7” is a human readable string representation of the number 7
  - 7 is a raw binary number stored in memory
- Examples

```
FILE * file;
int i = 7

file = fopen(”foo”, ”r+”);
fprintf(file, "%d\n", i);

Memory: 0x00000007
File: 0x37, 0x0a
```

```
FILE * file;
int i = -7;

file = fopen(”foo”, ”r+”);
fprintf(file, "%d\n", i);

Memory: 0xfffffff9
File: 0x2d, 0x37, 0x0a
```

ASCII Files

- ASCII: Encoding that maps bytes to text characters
  - Translates characters from human readable to machine readable
  - ”7” is a human readable string representation of the number 7
  - 7 is a raw binary number stored in memory

- Examples

```
FILE * file;
int i = 7

file = fopen(”foo”, ”r+”);
fprintf(file, "%d\n", i);
```

```
Memory: 0x00000007
File: 0x37, 0x0a
```

```
FILE * file;
int i = -7;

file = fopen(”foo”, ”r+”);
fprintf(file, "%d\n", i);
```

```
Memory: 0xfffffff9
File: 0x2d, 0x37, 0x0a
```

Pointers

- A pointers is an alias, a renaming of a variable
  - Similar to a file shortcut/alias on a desktop

- `int * pi; -> '*' indicates a pointer`

- A pointer includes the type of data it is pointing to
  - In this case an int
  - Can be any C data type and a special **`void`** type

Pointers Examples

```
int   x =  4;
int * y = &x;
int   z = *y;
printf("z = %d\n", z);
➔ z = 4
```

- **Examples:**
  - **dereference.c**
    - **Getting around pass by value**
  - **ptrSwap.c**
    - **Swapping two numbers in another function**
  - **dangerous.c**
    - **Be very careful about dereferencing**
    - **This is where a LOT of bugs happen**
    - **Pointers are very powerful, but can be very dangerous**

Arrays

- Arrays are a natural data type in C
  - Memory is an array
  - Just a sequence of values back to back in memory
- Raw values of a single data type
  - This means that arrays cannot change size
  - Arrays must be initialized with the correct size
- Initialization
  - `int a[] = {1, 2, 3, 4, 5, 6};`
  - `int c[6];`
- Accessing arrays
  - `int x = c[2];`
  - The compiler knows where in memory `c` starts
  - … so it just goes to the correct offset in memory

Arrays and pointer arguments

- `int * x;` and `int x[];` are the same thing!
  - `int foo(int * x) == int foo(int x[]);`
- x[1] is a base/offset calculation
  - Address of array 'x' + offset of index '1'
- Offsets
  - C is smart about offset calculations
    - Scales offset by the size of the ***type***

Strings

- What are strings in C?
  - Already saw them defined as (`char *`)'s.
  - … so they are just arrays of characters (bytes)…
  - … with a NULL terminator ('\0') at the end
- String format
  - Valid strings must be NULL terminated
  - Assumed by the standard library string functions
    - Many functions that look like: `str*(…);`
- Syntax:
  - `"c"` is a string (2 bytes)
  - `'c'` is a single character (1 byte)

Reading a string safely

- You **MUST** ensure that you only read a certain length
  - Limit the input you accept to the size of the target buffer

```
char * fgets(char * s, int size, FILE * stream);
```

Reads a **line** of text from the console. Returns after a newline, **UNLESS** the string exceeds 'size' bytes

```
int sscanf(char * str, char * fmt, …);
```

Same functionality as **scanf**, but reads input from string 'str' and not the console

- Together fgets and sscanf can safely parse input from the console
  - First, read a line into a char array with fgets, then parse it with sscanf

Better string functions

- Earlier functions were **VERY DANGEROUS**
  - Did not have any bounds restrictions
- Better to use explicit bounds versions
  - `size_t strnlen(char * s, size_t maxlen);`
  - `char * strncpy(char * dst, char * src, size_t n);`
  - `int strncmp(char * s1, char * s2, size_t n);`
  - `char * strncat(char * s1, char * s2, size_t n);`
- All(?) str*() functions have strn*() versions
  - But you still need to be careful
  - strn*() functions do not guarantee the output will be NULL terminated

Better string functions

- Earlier functions were **VERY DANGEROUS**
  - Did not have any bounds restrictions
- Better to use explicit bounds versions
  - `size_t strnlen(char * s, size_t maxlen);`
  - `char * strncpy(char * dst, char * src, size_t n);`
  - `int strncmp(char * s1, char * s2, size_t n);`
  - `char * strncat(char * s1, char * s2, size_t n);`
- All(?) str*() functions have strn*() versions
  - But you still need to be careful
  - strn*() functions do not guarantee the output will be NULL terminated

Allocating stack memory

- Allocating stack space means increasing the size of the stack
  - Add more memory to the end of the stack
  - **DOES NOT INITIALIZE THE CONTENTS**
- The Stack Pointer (**`esp`**)
  - A special **register** that contains a memory address
    - The keyword is "Pointer"
  - Marks the current end of the stack
    - Address of the last byte of data on the stack
  - Memory is allocated by adding/subtracting from **`esp`**
- **Example: dangerous2.c**

Dynamic Memory

- Sometimes the stack is not good enough
  - You want to share data across functions
  - You want data to be persistent across function calls
  - The data size is too large for the stack
  - You don't know how much memory you will need at compile time
    - E.g. reading a file into memory (how big is the file?)
- A new way to get memory
  - Like new/delete from Java
  - Allocates memory from the **HEAP**
- **`void * malloc(size_t size);`**
  - **Allocates 'size' bytes of memory and returns the address**
- **`void free(void * ptr);`**
  - **Frees the memory allocated at the address 'ptr'**

The Heap

- A region of memory that is managed by the programmer
  - Programmer requests ***X*** bytes of memory
- Heap memory is not scoped
  - Memory is reserved until its released/free()'d
- Heap memory has no "name"
  - `int x;` --> compiler names 4 bytes of memory
    - When ***x*** goes out of scope its memory is automatically released (How?)
  - The compiler does not name a heap allocation
    - Not automatically released when it goes out of scope

Accessing heap memory

- Compiler does not associate a variable with heap allocations
  - *Q: How do we access it?* **A: Pointers**
- Review: what are pointers?
  - A variable that stores an address
  - Pointer variables are ***scoped***
    - Compiler manages the memory for the pointer ***variable***
- Pointers are used as ***handles*** for heap memory

realloc and calloc

- `void * realloc(void *ptr, size_t size);`
  - Similar to malloc, but changes the size of an already allocated buffer
  - First argument is a pointer to the buffer to change

```
realloc(null, 5); == malloc(5);

char * x = malloc(5);
x = realloc(x, 10);
```
is the same as
```
x = malloc(10);
```

- `void * calloc(size_t nmemb, size_t size);`
  - Allocates an array of *nmemb* elements of size *size*
  - **INITIALIZES BUFFER MEMORY TO '\0'**
  - I prefer to use this over malloc

```
char * x = calloc(5, sizeof(char));
```
is the same as
```
x = malloc(5 * sizeof(char));
memset(x, 0, 5 * sizeof(char));
```

Pointer arguments

- Used to pass values as well as return values from a function
  - Sometimes both at the same time
- Another use of '**' is to return a pointer from a function

```
ssize_t getline(char ** linep, size_t * linecapp, FILE * stream);
```

Similar to **fgets**, but can dynamically allocate memory from the heap

```
{
    size_t len = 0;
    char * str = NULL;
    len = getline(&str, NULL, stdin);
}
{
    size_t len = 5;
    char * str = malloc(5);
    len = getline(&str, &len, stdin);
}
```

Searching (2)

- **char * strnstr(char * s1, char * s2, size_t n);**
  - Similar to strchr but searches for a string (*s2*)
  - Also has bounds limit via (*n*)
  - Returns location of first occurrence of string *s2* in *s1*
- **char * strtok(char * str, const char * delim);**
  - Returns the next string after a token
  - *delim* is a list of tokens, as a string
  - Modifies the contents of *str*
- **char * strsep(char ** stringp, char * delim);**
  - Replacement for strtok, but not as portable
  - Modifies value of *stringp* (input and output argument)

These functions are very complex, with lots of rules for return values and edge cases
Use with caution

Structs

- C structs are like classes in Java
  - Declares a new type
  - `struct foo;` means there is a type 'struct foo'
- Java classes encapsulates code + data
- C structs include only data
  - No fancy OOP techniques (inheritance, subclasses, etc)
  - But in C : code == data (more on this later)
- Typedef
  - `typedef <original type> <new type>;`
  - `typedef unsigned int uint32_t;`
  - `typedef struct foo foo_t;`

Assigning values to structs

- structs are like any over data type
  - Automatically or dynamically allocated

Stack Allocation:

- Fields accessed with '.'

```
int main() {
    struct foo my_foo;

    memset(&my_foo, 0, sizeof(struct foo));

    my_foo.x    = 4;
    my_foo.y    = 10;
    my_foo.name = "George";
}
```

Assigning values to structs

- structs are like any over data type
  - Automatically or dynamically allocated

Heap Allocation:

- Fields accessed with '->'

```
int main() {
    struct foo * my_foo = calloc(1, sizeof(struct foo));

    my_foo->x       = 4;
    my_foo->name    = "George";
    (*my_foo).y     = 10;

    free(foo);
}
```

Structs in memory

- What do structs look like in memory?
  - Raw values of member variables

```
struct foo {
    int x;
    int y;
    char * name
};
```

```
int main() {
    struct foo   my_foo_1;
    struct foo   my_foo_2;

    my_foo_1 = my_foo_2;
}
```

- Copying Structs
  - C provides shallow copies of struct values
    - `memcpy(&my_foo_1, &my_foo_2, sizeof(struct foo));`
  - Copies raw bytes of struct data
    - We know the how many bytes via `sizeof();`

More complex data structures

- **KEY POINT:**
  - structs can contain pointers to the same struct type

```
struct list_node {
    int x;
    struct list_node * next;
}
```

- This is not self referential
  - Rather it points to another variable structure elsewhere in memory
- This allows us to construct collections
  - lists, trees, graphs, etc

Preprocessing

- Compiling C programs is a multistage operation
  - Before code is actually compiled it is "pre-processed"
- C Pre-Processor
  - Basically just a simple syntax for text manipulation
  - Produces the final text that is used by the compiler
- We've seen an example already
  - **`#include <stdio.h>`**
  - Replace the line with the raw contents of the specified file

#define Example

- Usually used for commonly used constants
  - E.g. buffers usually have a standard size
  - Allows changing the constant in one location
    - Don't need to track down every usage
    - Avoids a lot of bugs

```
#define BUF_SIZE 100

int main() {
    char buf[BUF_SIZE];
    memset(buf, 0, BUF_SIZE);
    fgets(buf, BUF_SIZE, stdin);
}
```

Conditional Macros

- Enable/Disable blocks of code before compilation

```
#ifdef MACRO
…
#endif
```

AND

```
#ifndef MACRO
…
#endif
```

- #ifdef
  - Useful for turning on/off debugging features
  - Useful for compile time configuration
- Example:

```
#ifdef ENABLE_DEBUGGING
   printf(lots of state values);
#endif

> gcc -DENABLE_DEBUGGING test.c -o test
```

Function Macros

- Text replacement that understands arguments
  - Looks like a function definition
  - Only matches text that looks like a function : "()"

```
#define FOO(x) (x++)
#define MAX(x, y) (x > y) ? x : y;
```

- Only text replacement
  - The function body can be **ANY** text
  - Syntax is not checked until actual compilation
  - Can be ***extremely*** confusing and convoluted
    - **BEWARE COMPLEX MACROS**

Typecasting

- C provides basic type checking
  - If you try `int x = "foo";` the compiler will catch it
- However, you can easily override this
  - Example: **typecase1.c**

```
char * str = "foo";
int    x   = *(int *)str;
```

- The universal type : **void *** *(Example: **typecast2.c**)*
  - A raw memory address, can point to anything
  - Any pointer can be implicitly cast to a 'void *'

```
int foo(void * x);

void * ptr = NULL;
char * str = "hello";
int x = 4;

foo(ptr);
foo(str);   // This is OK
foo(&x);    // This is also OK
```

Function Pointers

- Every part of the program is stored in memory
  - Code, data, everything
  - C allows you direct access to memory
  - So we can access the code itself
  - *Or, we can execute data as code*

- **Function pointers:** *Functions as a data type*
  - We can assign a function to a variable
    - Just a memory address where the code is located
  - Then "call" the variable
    - Force the CPU to jump to that location in memory

Function Pointer Example

```
int bar(int x, int y) {return x + y);

int (*foo)(int x, int y);

foo = bar;
foo = &bar; // Same thing
foo(1, 2);
```

- Looks similar to regular pointers
  - But there is a subtle difference

```
int * x;     // x is a pointer to an int
int (*x)(); /* x is a pointer to a function
             *   that returns an int
             */
```