Memory, Data, & Addressing II

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/334/

Administrivia
- Lab 0 due today @ 11:59pm
  - You will be revisiting this program throughout this class!
- Homework 1 due Wednesday
  - Reminder: autograded, 20 tries, no late submissions
- Lab 1 released today
  - Prelim due Jan. 15
  - Due Jan. 19

Memory, Data, and Addressing
- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

Addresses and Pointers in C
- A pointer is a variable that holds an address
- Pointers are declared similarly to other variables in C
  - Type (e.g., int *)
  - Name (e.g., ptr)
  - Declaration, Initialization, Assignment
- Type is specified using one (or more) * after some type T
  - int *ptr;
  - struct Scores *s;
  - double **dPtr;
- Operators
  - & = "address of" operator
  - * = "dereference" operator, or "value at address"

Assignment in C
- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds "garbage")
- Left-Hand Side = Right-Hand Side
  - = operator
  - LHS is a memory location
  - RHS is a value (could be an address)
Assignment in C

❖ int x, y;
❖ x = 0;
❖ y = 0x3CD02700;
❖ Get value at y, add 3, store in x
❖ int *z;
❖ z is at address 0x20
❖ &x = 0x00000004
❖ y = *z + 1;
Assignment in C

- `int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- `int *z = &x;
  - *z = 0x00000004
- y = *z + 1;
  - y = 0x3CD02704

*Pointer arithmetic can be dangerous!*

Get address of x, y;

x = 0;

What does this do?

*y = 0x3CD02700;

Get value at y;

x, y;

Can easily lead to bad memory accesses

Be careful with data types and casting

"add 3", store in z

12 + 3*(4) = 24

"address of"

"dereference"

32-bit example (pointers are 32-bits wide)

- `int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- `int *z = &y + 3;
  - z is at address 0x20

Pointer arithmetic is scaled by the size of the pointer's target data type

- In this example, `sizeof(int) = 4`
- `int* z = &y + 3;
  - Get address of y, add 3*sizeof(int), store in z
  - 4y = 0x18 = 1*16^1 + 8*16^0 = 24
  - 24 + 3*4 = 36 = 2*16^1 + 4*16^0 = 0x24

`Pointer arithmetic can be dangerous!`

Can easily lead to bad memory accesses

Be careful with data types and casting

"address of"

"dereference"

32-bit example (pointers are 32-bits wide)
### Arrays in C

#### Declaration:

- `int a[6];`

#### Indexing:

- `a[0] = 0x015f;`
- `a[5] = a[0];`

#### No bounds:

- `a[6] = 0xBAD;`
- `a[-1] = 0xBAD;`

#### Pointers:

- `int* p;`  
- `p = a;`  
- `p += 3;`  
- `*p = 0x015f;`  
- `p = &a[0];`  
- `p[5] = 0x015f;`  
- `*(p+1) = 0x015f;`  
- `p = p + 2;`

#### Arrays are adjacent locations in memory

- Storing the same type of data object
- The address of `a[1]` is the address of `a[0]` plus `i` times the element size in bytes

### Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
- `int* z = &y + 3;`
- `Get value at y, add 3, store in x`
- `Get address of y, add 12, store in z`
- `Get value of y, put in address stored in z`
Arrays in C

Declaring: `int a[6];`

Accessing: `a[0] = 0x123456;`

Null-terminator: `char *p = a[0];`

Endianness and Strings

C (char = 1 byte)

String literal: `char s[6] = "123456";`

Endianness:
- Little-endian (IA32, ARM64)
- Big-endian (SPARC)

Representing strings
- C-style string stored as an array of bytes (char*)
  - Elements are one-byte ASCII codes for each character
  - No "String" keyword, unlike Java

Endianness:
- Byte ordering (endianness) is not an issue for 1-byte values
  - The whole array does not constitute a single value
  - Individual elements are values; chars are single bytes
Examining Data Representations

❖ Code to print byte representation of data
  ▪ Any data type can be treated as a byte array by casting it to char
  ▪ C has unchecked casts !!! DANGER !!

```c
void show_bytes(char* start, int len) {
  int i;
  for (i = 0; i < len; i++)
    printf("%p \t \0x%.2x\n", start+i, *(start+i));
  printf("\n");
}
```

printf directives:
  \%p  Print pointer
  \t  Tab
  \%x  Print value as hex
  \n  New line

show_bytes Execution Example

```c
int x = 12345; // 0x00003039
printf("int x = %d;\n", x);
show_int(x); // show_bytes(char * \&x, sizeof(int));
```

❖ Result (Linux x86-64):
  ▪ Note: The addresses will change on each run (try it!), but fall in same general range

```c
int x = 12345;
0x7fffb7f71dbc 0x39
0x7fffb7f71dbd 0x30
0x7fffb7f71dbe 0x00
0x7fffb7f71dbf 0x00
```