Memory, Data, & Addressing II
CSE 351 Spring 2021

Instructor:
Ruth Anderson

Teaching Assistants:
Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Neil Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu

http://xkcd.com/138/
Administrivia

- hw1 due tonight, Friday (4/02) @ 11:59 pm
- Lab 0 and hw2 due Monday (4/05) @ 11:59 pm
- hw3 due Wednesday (4/07) @ 11:59 pm
- Lab 1a coming soon! due next Monday (4/12)
  - Pointers in C
  - Submitted via Gradescope
  - Reminder: last submission graded, *individual* work
- Questions Doc: You must log on with your @uw google account to access!!
  - [https://tinyurl.com/CSE351-21sp-Questions](https://tinyurl.com/CSE351-21sp-Questions)
  - Then open the “TODAY's Lecture Questions” doc for 11:30/2:30
Late Days

- You are given 7 late days for the whole quarter
  - Late days can only apply to Labs & Unit Summaries
  - No benefit to having leftover late days
- Count lateness in days (even if just by a second)
  - Special: weekends count as one day
  - No submissions accepted more than two days late
- Late penalty is 20% deduction of your score per day
  - Only late work is eligible for penalties
  - Penalties applied at end of quarter to maximize your grade
- Use at own risk – don’t want to fall too far behind
  - Intended to allow for unexpected circumstances
Memory, Data, and Addressing

- Representing information as bits and bytes
  - Binary, hexadecimal, fixed-widths
- Organizing and addressing data in memory
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
  - Assignment
  - Pointers, pointer arithmetic, and arrays
- Boolean algebra and bit-level manipulations
Reading Review

- Terminology:
  - address-of operator (&), dereference operator (*), NULL
  - box-and-arrow memory diagrams
  - pointer arithmetic, arrays
  - C string, null character, string literal
Review Questions

1. \[ \text{int } x = 351; \]
   \[ \text{char } *p = &x; \]
   \[ \text{int } ar[3]; \]

   How much space does the variable \( p \) take up?
   - A. 1 byte
   - B. 2 bytes
   - C. 4 bytes
   - D. 8 bytes

2. Assume 64-bit words x86-64

   Which of the following expressions evaluate to an address?
   - A. \( x + 10 \)
   - B. \( p + 10 \)
   - C. \( &x + 10 \)
   - D. \( \ast(&p) \)
   - E. \( ar[1] \)
   - F. \( &ar[2] \)
Addresses and Pointers in C

- `&` = “address of” operator
- `*` = “value at address” or “dereference” operator

```c
int* ptr;

int x = 5;
int y = 2;

ptr = &x;

y = 1 + *ptr;
```

- Declares a variable, `ptr`, that is a pointer to (i.e. holds the address of) an `int` in memory
- Declares two variables, `x` and `y`, that hold `ints`, and initializes them to 5 and 2, respectively
- Sets `ptr` to the address of `x` (“`ptr` points to `x`”)
- “Dereference `ptr`”
- Sets `y` to “1 plus the value stored at the address held by `ptr`.” Because `ptr` points to `x`, this is equivalent to `y=1+x`;

What is `*(&y)`? => `y`
Pointer Operators

- & = “address of” operator
- * = “value at address” or “dereference” operator

Operator confusion

- The pointer operators are unary (i.e., take 1 operand)
- These operators both have binary forms
  - x & y is bitwise AND (we’ll talk about this next lecture)
  - x * y is multiplication
- * is also used as part of the data type in pointer variable declarations – this is NOT an operator in this context!
Assignment in C

- A variable is represented by a location
- **Declaration ≠ initialization** (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7 00 32 00</td>
</tr>
<tr>
<td>0x04</td>
<td>00 01 29 F3</td>
</tr>
<tr>
<td>0x08</td>
<td>EE EE EE EE</td>
</tr>
<tr>
<td>0x0C</td>
<td>FA CE CA FE</td>
</tr>
<tr>
<td>0x10</td>
<td>26 00 00 00</td>
</tr>
<tr>
<td>0x14</td>
<td>00 00 10 00</td>
</tr>
<tr>
<td>0x18</td>
<td>01 00 00 00</td>
</tr>
<tr>
<td>0x1C</td>
<td>FF 00 F4 96</td>
</tr>
<tr>
<td>0x20</td>
<td>DE AD BE EF</td>
</tr>
<tr>
<td>0x24</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Current state of memory
Assignment in C

- A variable is represented by a location
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

32-bit example (pointers are 32-bits wide)
little-endian
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location

```c
int x, y;

x = 0;
```

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

& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`

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

& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;
  x = 0;
  y = 0x3CD02700;
  x = y + 3;
- Get value at y, add 3, store in x

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

& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at y, add 3, store in x

- `int* z;`
  - z is at address 0x20

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

& = “address of”
* = “dereference”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

```
int x, y;
int* z = &y + 3;
```

- Get value at y, add 3, store in x
- Get address of y, "add 3", store in z

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>03</td>
</tr>
<tr>
<td>0x08</td>
<td>27</td>
</tr>
<tr>
<td>0x0C</td>
<td>D0</td>
</tr>
<tr>
<td>0x10</td>
<td>3C</td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td>27</td>
</tr>
<tr>
<td>0x20</td>
<td>D0</td>
</tr>
<tr>
<td>0x24</td>
<td>3C</td>
</tr>
</tbody>
</table>

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

& = "address of"
* = "dereference"
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int) = 4`

- `int* z = &y + 3;`
  - Get address of `y`, add `3 * sizeof(int)`, store in `z`
  - `&y = 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`
Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add 12, store in `z`
- `*z = y;`
  - What does this do?

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

& = “address of”
*
* = “dereference”
Assignment in C

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

Declaration: `int a[6];` // &a is 0x10

- **Element type:** 4 bytes each
- **Name:**
- **Number of elements:**

Arrays are adjacent locations in memory storing the same type of data object.

`a` (array name) returns the array’s address.

64-bit example (pointers are 64-bits wide)
Arrays in C

Declaration: `int a[6];`

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

Arrays are adjacent locations in memory storing the same type of data object

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes
Arrays in C

Declaration: `int a[6];`

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

No bounds checking:  
- `a[6] = 0xBAD;`

arrays are adjacent locations in memory storing the same type of data object

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Arrays are adjacent locations in memory storing the same type of data object

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes
Arrays in C

Declaration: `int a[6];`

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

No bounds checking:  
- `a[6] = 0xBAD;`

Pointer equivalent:  
- `int* p;`
- `p = &a[0];`
- `*p = 0xA;`

Arrays are adjacent locations in memory storing the same type of data object.

- `a` (array name) returns the array’s address.
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.
Arrays in C

Declaration: `int a[6];`

Indexing: `a[0] = 0x015f;`  
`a[5] = a[0];`

No bounds `a[6] = 0xBAD;`
checking: `a[-1] = 0xBAD;`

Pointers: `int* p;`  
`p = &a[0];`  
`*p = 0xA;`  
`p[1] = 0xB;`  
`*(p+1) = 0xB;`

Arranges are adjacent locations in memory storing the same type of data object

a (array name) returns the array’s address

`&a[i]` is the address of `a[0]` plus `i` times the element size in bytes

Array indexing = address arithmetic (both scaled by the size of the type)

Equivalent:

```
p[0] = a;  
p = &a[0];  
*p = 0xA;  
a[2] = 0x10 + 2  
0x10 + 2 -> 0x18  
a + 2*size of(int) = 0x18
```
Arrays in C

Declaration: `int a[6];`

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

No bounds:  
- `a[6] = 0xBAD;`
checking:  
- `a[-1] = 0xBAD;`

Pointers:  
- `int* p;`
- `p = a;`
- `a[0]`
- `p = &a[0];`
- `a[2]`
- `*p = 0xA;`
- `a[4]`

Array indexing = address arithmetic (both scaled by the size of the type)

Equivalent:  
- `p[1] = 0xB;`
- `*(p+1) = 0xB;`
- `p = p + 2;`

Arrays are adjacent locations in memory storing the same type of data object

- `a` (array name) returns the array’s address
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes
**Question:** The variable values after Line 3 executes are shown on the right. What are they after Line 5?

- No voting

```c
void main() {
    int a[] = {0x5, 0x10};
    int* p = a;
    p = p + 1;
    *p = *p + 1;
}
```

<table>
<thead>
<tr>
<th></th>
<th>Data (hex)</th>
<th>Address (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>5</td>
<td>0x100</td>
</tr>
<tr>
<td>a[1]</td>
<td>10</td>
<td>0x104</td>
</tr>
<tr>
<td>p</td>
<td>100</td>
<td>0x104</td>
</tr>
</tbody>
</table>

(A) 0x101  0x5  0x11
(B) 0x104  0x5  0x11
(C) 0x101  0x6  0x10
(D) 0x104  0x6  0x10
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

<table>
<thead>
<tr>
<th>ASCII Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
</tr>
<tr>
<td>3</td>
<td>#</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>%</td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
</tr>
<tr>
<td>7</td>
<td>'</td>
</tr>
<tr>
<td>8</td>
<td>(</td>
</tr>
<tr>
<td>9</td>
<td>)</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>,</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>.</td>
</tr>
<tr>
<td>15</td>
<td>/</td>
</tr>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>48</td>
<td>@</td>
</tr>
<tr>
<td>64</td>
<td>P</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
</tbody>
</table>

ASCII: American Standard Code for Information Interchange
Representing strings

- C-style string stored as an array of bytes (**char***)
  - No “String” keyword, unlike Java
  - Elements are one-byte ASCII codes for each character
  - Last character followed by a 0 byte (\'\0\')
    (a.k.a. "null terminator")

<table>
<thead>
<tr>
<th>Decimal</th>
<th>80</th>
<th>108</th>
<th>101</th>
<th>97</th>
<th>115</th>
<th>101</th>
<th>32</th>
<th>118</th>
<th>111</th>
<th>116</th>
<th>101</th>
<th>33</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>0x50</td>
<td>0x6C</td>
<td>0x65</td>
<td>0x61</td>
<td>0x73</td>
<td>0x65</td>
<td>0x20</td>
<td>0x76</td>
<td>0x6F</td>
<td>0x74</td>
<td>0x65</td>
<td>0x21</td>
<td>0x00</td>
</tr>
<tr>
<td>Text</td>
<td>'P'</td>
<td>'l'</td>
<td>'e'</td>
<td>'a'</td>
<td>'s'</td>
<td>'e'</td>
<td></td>
<td>'v'</td>
<td>'o'</td>
<td>'t'</td>
<td>'e'</td>
<td>'!'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

string literal: "Please vote!" uses 13 bytes
(doule quotes)
Endianness and Strings

`char s[6] = "12345";

- 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

C (char = 1 byte)
Examining Data Representations

- Code to print byte representation of data
  - Treat any data type as a *byte array* by casting its address 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\t0x%.2hhX\n", start+i, *(start+i));
    printf("\n");
}
```

- `printf` directives:
  - `%p` Print pointer
  - `\t` Tab
  - `%2hhX` Print value as char (hh) in hex (X), padding to 2 digits ( . 2)
  - `\n` New line
Examining Data Representations

❖ Code to print byte representation of data
  - Treat any data type as a *byte array* by *casting* its address 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\t0x%.2hhX\n", start+i, *(start+i));
    printf("\n");
}

void show_int(int x) {
    show_bytes( (char *) &x, sizeof(int));
}
```
show_bytes Execution Example

```c
int x = 123456; // 0x00 01 E2 40
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 = 123456;
0x7fffb245549c  0x40
0x7fffb245549d  0xE2
0x7fffb245549e  0x01
0x7fffb245549f  0x00
```
Summary

- Assignment in C results in value being put in memory location
- Pointer is a C representation of a data address
  - \& = “address of” operator
  - \* = “value at address” or “dereference” operator
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory
  - Be careful when using – particularly when casting variables
- Arrays are adjacent locations in memory storing the same type of data object
  - Strings are null-terminated arrays of characters (ASCII)