Memory, Data, & Addressing II
CSE 351 Autumn 2020

Instructor:
Justin Hsia

Teaching Assistants:
Aman Mohammed
Ami Oka
Callum Walker
Cosmo Wang
Hang Do
Jim Limprasert
Joy Dang
Julia Wang
Kaelin Laundry
Kyrie Dowling
Mariam Mayanja
Shawn Stanley
Yan Zhe Ong

http://xkcd.com/138/
Adminstrivia

- Lab 0 due today @ 11:59 pm
  - You will revisit this concepts from program!

- hw2 due Wednesday, hw3 due Friday @ 11:00 am
  - Autograded, unlimited tries, no late submissions

- Lab 1a released today, due next Monday (10/12)
  - Pointers in C
  - Reminder: last submission graded, individual work
Late Days

- You are given **5 late day tokens** for the whole quarter
  - Tokens can only apply to Labs
  - No benefit to having leftover tokens
- 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 labs are 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
Reading Review

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

- Questions from the Reading?
Review Questions

- `int x = 351;
  char *p = &x;
  int ar[3];`

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

- Which of the following expressions evaluate to an address?
  - A. `x + 10`
  - B. `p + 10`
  - C. `&x + 10`
  - D. `*(&p)`
  - E. `ar[1]`
  - F. `&ar[2]`
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>

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

```
0x00 0x01 0x02 0x03
0x00                X
0x04 0x01 29 F3
0x08
0x0C
0x10
0x14
0x18 0x00 00 00
0x1C
0x20
0x24
```
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;`
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”

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;
y = 0x3CD02700;
x = y + 3;
```

- Get value at \( y \), add 3, store in \( x \)
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

```c
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
```

- Get value at `y`, add 3, store in `x`

```c
int* z = &y + 3;
```

- Get address of `y`, "add 3", store in `z`

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`
- `*z = y;`

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
- *z = y;
  - Get value of y, put in address stored in z

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

& = “address of”
* = “dereference”

The target of a pointer is also a location
Addresses and Pointers in C

- Draw out a box-and-arrow diagram for the result of the following C code:

```c
int* ptr;

int x = 5;

int y = 2;

ptr = &x;

y = 1 + *ptr;
```
Arrays in C

Declaration: `int a[6];`

- **element type**: int
- **name**: a
- **number of elements**: 6

Arrays are adjacent locations in memory storing the same type of data object. The array `a` (array name) returns the array’s address.

64-bit example (pointers are 64-bits wide)

```
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
```

- `a[0]` at `0x00`
- `a[1]` at `0x08`
- `a[2]` at `0x10`
- `a[3]` at `0x18`
- `a[4]` at `0x20`
- `a[5]` at `0x28`
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;`
- `a[-1] = 0xBAD;`

The address of `a[0]` is `0x0000015F`, and the address of `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.
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;`

Equivalent:  
            
            `p = a;`  
            `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

The diagram shows the memory layout of the array where each element is 4 bytes and the pointers are used to access the array elements.
Arrays in C

Declaration: \texttt{int a[6];}

Indexing: \begin{align*}
\texttt{a[0]} &= 0\times015f; \\
\texttt{a[5]} &= \texttt{a[0]};
\end{align*}

No bounds \begin{align*}
\texttt{a[6]} &= 0\timesBAD; \\
\texttt{checking: a[-1]} &= 0\timesBAD;
\end{align*}

Pointers: \begin{align*}
\texttt{int* p;}
\end{align*}

\begin{align*}
\text{equivalent} & \begin{cases}
\texttt{p} = \texttt{a}; & \texttt{a[0]} \\
\texttt{p} = \&\texttt{a[0];} & \texttt{a[2]} \\
\texttt{*p} = 0\timesA; & \texttt{a[4]}
\end{cases}
\end{align*}

\begin{align*}
\text{array indexing = address arithmetic (both scaled by the size of the type)} & \begin{cases}
\texttt{p[1]} = 0\timesB; & \texttt{p[1]} \\
\texttt{* (p+1)} = 0\timesB; & \texttt{*(p+1)}
\end{cases} \\
\text{equivalent} & \begin{cases}
\texttt{p} = \texttt{p + 2;}
\end{cases}
\end{align*}

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

\begin{align*}
\texttt{a} \text{ (array name)} & \text{ returns the array’s address} \\
\&\texttt{a[i]} & \text{ is the address of } \texttt{a[0]} \text{ plus } i \text{ times the element size in bytes}
\end{align*}
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;`

**equivalent**  
```c
p = a;  
p = &a[0];  
*p = 0xA;
```

Array indexing = address arithmetic  
(both scaled by the size of the type)  
**equivalent**  
```c
p[1] = 0xB;  
*(p+1) = 0xB;  
p = p + 2;
```

\*p = a[1] + 1;

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

\[\text{a (array name)}\] returns the array’s address

\[\&\text{a[i]}\] is the address of \[\text{a[0]}\] 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?

- Vote in Ed Lessons

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

<table>
<thead>
<tr>
<th>p</th>
<th>a[0]</th>
<th>a[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>0x101</td>
<td>0x5</td>
</tr>
<tr>
<td>(B)</td>
<td>0x104</td>
<td>0x5</td>
</tr>
<tr>
<td>(C)</td>
<td>0x101</td>
<td>0x6</td>
</tr>
<tr>
<td>(D)</td>
<td>0x104</td>
<td>0x6</td>
</tr>
</tbody>
</table>
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

| 32 | space | 48 | 0 |
| 33 | !     | 49 | 1 |
| 34 | "     | 50 | 2 |
| 35 | #     | 51 | 3 |
| 36 | $     | 52 | 4 |
| 37 | %     | 53 | 5 |
| 38 | &     | 54 | 6 |
| 39 | '     | 55 | 7 |
| 40 | (     | 56 | 8 |
| 41 | )     | 57 | 9 |
| 42 | *     | 58 | : |
| 43 | +     | 59 | ; |
| 44 | ,     | 60 | < |
| 45 | -     | 61 | = |
| 46 | .     | 62 | > |
| 47 | /     | 63 | ? |
| 64 | @     | 65 | A |
| 66 | B     | 67 | C |
| 68 | D     | 69 | E |
| 70 | F     | 71 | G |
| 72 | H     | 73 | I |
| 74 | J     | 75 | K |
| 76 | L     | 77 | M |
| 78 | N     | 79 | O |
| 80 | P     | 81 | Q |
| 82 | R     | 83 | S |
| 84 | T     | 85 | U |
| 86 | V     | 87 | W |
| 88 | X     | 89 | Y |
| 90 | Z     | 91 | [ |
| 92 | \    | 93 | ] |
| 94 | ^     | 95 | _ |
| 96 | `     | 97 | a |
| 98 | b     | 99 | c |
| 100| d     | 101| e |
| 102| f     | 103| g |
| 104| h     | 105| l |
| 106| j     | 107| k |
| 108| l     | 109| m |
| 110| n     | 111| o |
| 112| p     | 113| q |
| 114| r     | 115| s |
| 116| t     | 117| u |
| 118| v     | 119| w |
| 120| x     | 121| y |
| 122| z     | 123| { |
| 124| |      | 125| } |
| 126| ~     | 127| del |

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

```c
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

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>0x00 0x00 '1'</td>
</tr>
<tr>
<td>0x01 32</td>
<td>0x01 0x01 '2'</td>
</tr>
<tr>
<td>0x02 33</td>
<td>0x02 0x02 '3'</td>
</tr>
<tr>
<td>0x03 34</td>
<td>0x03 0x03 '4'</td>
</tr>
<tr>
<td>0x04 35</td>
<td>0x04 0x04 '5'</td>
</tr>
<tr>
<td>0x05 00</td>
<td>0x05 0x05 '\0'</td>
</tr>
</tbody>
</table>

0x31 = 49 decimal = ASCII ‘1’
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;
0x7ffffb245549c  0x40
0x7ffffb245549d  0xE2
0x7ffffb245549e  0x01
0x7ffffb245549f  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)