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/
Administrivia

- Lab 0 due today @ 11:59 pm
  - You will revisit these 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 \(\rightarrow\) int
  - B. p + 10 \(\rightarrow\) char *
  - C. &x + 10 \(\rightarrow\) int *
  - D. *(&p) \(\rightarrow\) char *
  - E. ar[1] \(\rightarrow\) int
  - F. &ar[2] \(\rightarrow\) int *
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

```
int x, y;
```

![Current state of memory](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”

little endian!

least significant byte
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 \( x \)

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 \textit{location}
  - RHS must evaluate to a \textit{value} (could be an address)
  - Store RHS value at LHS location

- \texttt{int x, y;}
- \texttt{x = 0;}
- \texttt{y = 0x3CD02700;}
- \texttt{x = y + 3;}
  - Get value at \texttt{y}, add 3, store in \texttt{x}
- \texttt{int* z = \&y + 3;}  \texttt{// expect 0x1b}
  - Get address of \texttt{y}, “add 3”, store in \texttt{z}

\textbf{32-bit example (pointers are 32-bits wide)}

\& = “address of”
* = “dereference”

\textbf{Pointer arithmetic}

get this instead (scale by sizeof(int)=4)
Assignment in C

- \texttt{int x, y;}
- \texttt{x = 0;}
- \texttt{y = 0x3CD02700;}
- \texttt{x = y + 3;}
  - Get value at \texttt{y}, add 3, store in \texttt{x}
- \texttt{int* z = \&y + 3;}
  - Get address of \texttt{y}, add \texttt{12}, store in \texttt{z}
- \texttt{*z = y;}

\textbf{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”`
Addresses and Pointers in C

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

1. \( \text{int* } \text{ptr;} \)
2. \( \text{int } x = 5; \)
3. \( \text{int } y = 2; \)
4. \( \text{ptr} = \&x; \)
5. \( y = 1 + (*\text{ptr}); \)
Arrays in C

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

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

![Diagram of array memory layout](image-url)
Arrays in C

Declaration: `int a[6];`

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

No bounds `a[6] = 0xBAD;`

checking: `a[-1] = 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 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;`
- `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;`
- equivalent:
  - `p = a;`
  - `p = &a[0];`
  - `*p = 0xA;`

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

Pointer arithmetic:
- `0x10 + 1 -> 0x14`
- `p = p + 2;`
- `0x10 + 2 -> 0x18`

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: \textbf{int} \ a[6];

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

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

Pointers: \ \textbf{int}* \ p;
\ equivalent \ \begin{cases} \ p = \ a; \\ \ p = \ &a[0]; \\ \ *p = 0xA; \end{cases}
\ equivalent \ \begin{cases} \ p[1] = 0xB; \\ \ *(p+1) = 0xB; \end{cases}
\ store \ at \ 0x18 \ \begin{cases} \ *p = \ a[1] + 1; \end{cases}

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?

- Vote in Ed Lessons

```c
1 void main() {
2    int a[] = {0x5, 0x10};
3    int* p = a;
4    p = p + 1;
5    *p = *p + 1;
6 }
```

(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**)  
  - No “String” keyword, unlike Java  
  - Elements are one-byte ASCII codes for each character

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
<td>1</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>3</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>5</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>6</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>8</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>9</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>51</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>6</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>7</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>9</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>10</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>''</td>
</tr>
</tbody>
</table>

string literal: "Please vote!" uses 13 bytes (double quotes)
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

- **IA32, x86-64 (little-endian)**
  - Little-endian order: `s[0]` is 0x0031 = 49 decimal = ASCII ‘1’

- **SPARC (big-endian)**
  - Big-endian order: `s[0]` is 0x0031 = 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");  // format string
}

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)