Instructor:
Mark Wyse

Teaching Assistants:
Kyrie Dowling
Catherine Guevara
Ian Hsiao
Jim Limprasert
Armin Magness
Allie Pfleger
Cosmo Wang
Ronald Widjaja

http://xkcd.com/138/
Administrivia

- Lab 0 due today @ 11:59 pm
  - You will revisit these concepts later!
- hw1 due today @ 11:59 pm
- hw2 due Monday, hw3 due Wednesday @ 11:00 am
  - Autograded, unlimited tries, no late submissions
- Lab 1a released today, due next Friday (1/15)
  - Pointers in C
  - Reminder: last submission graded, individual work
National Events, Resources, and Week 1

❖ Blog post by UW President Cauce:
  ▪ [https://www.washington.edu/president/2021/01/06/misinformation-disinformation-and-the-assault-on-democracy/](https://www.washington.edu/president/2021/01/06/misinformation-disinformation-and-the-assault-on-democracy/)

❖ Be there for each other, check in with friends and classmates, give space to process

❖ Support resources:
  ▪ CSE Undergraduate Advising: ugrad-adviser@cs.washington.edu
  ▪ Hall Health and Schmitz Hall Counseling Center: [https://wellbeing.uw.edu/topic/mental-health/](https://wellbeing.uw.edu/topic/mental-health/)
  ▪ SafeCampus is the UW's central reporting office if you are concerned for yourself or a friend. They have trained specialists who will take your call and connect you with appropriate resources. They are available 24/7 at 206-685-SAFE (206-685-7233).

❖ CSE 351: all week 1 work due Sunday 1/10 @ 11:59pm
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

```
0x00 0x01 0x02 0x03
0x00 A7 00 32 00
0x04 00 01 29 F3
0x08 EE EE EE EE
0x0C FA CE CA FE
0x10 26 00 00 00
0x14 00 00 10 00
0x18 01 00 00 00
0x1C FF 00 F4 96
0x20 DE AD BE EF
0x24 00 00 00 00
```

Little-endian

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

- 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!
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”
Assignments 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

```c
int x, y;

x = 0;

y = 0x3CD02700;

x = y + 3;
```

- Get value at \texttt{y}, add 3, store in \texttt{x}

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

- Get address of \texttt{y}, “add 3”, store in \texttt{z}

```
sizeof(int) = 4
```

```c
z = 0x18 + 3(4)
```

```
0x24 <= 12
```

- Pointer arithmetic

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`
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; = 6
```
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.

- `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 0x10
- `a[1]` at 0x18
- `a[2]` at 0x20
- `a[3]` at 0x28
- `a[4]` at 0x30
- `a[5]` at 0x38
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: \texttt{int a[6];}

Indexing: \texttt{a[0] = 0x015f; a[5] = a[0];}

No bounds \texttt{a[6] = 0xBAD;}

Checking: \texttt{a[-1] = 0xBAD;}

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

\texttt{a} (array name) returns the array’s address

\&\texttt{a[i]} is the address of \texttt{a[0]} plus \texttt{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.

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x08</th>
<th>0x10</th>
<th>0x18</th>
<th>0x20</th>
<th>0x28</th>
<th>0x30</th>
<th>0x38</th>
<th>0x40</th>
<th>0x48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0A</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>AD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pointers:**
- `p = a;` (equivalent to `p = &a[0];`)
- `*p = 0xA;`
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

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

equivalent `p[1] = 0xB;`  
`*(p+1) = 0xB;`

`p = p + 2;`


**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;`

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

- `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

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

```
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
0x0  0  0  0  0  0  0  0  0  0
0x8  0  0  0  0  0  0  0  0  0
0x10 0  0  0  0  0  0  0  0  0
0x18 0  0  0  0  0  0  0  0  0
0x20 0  0  0  0  0  0  0  0  0
0x28 0  0  0  0  0  0  0  0  0
0x30 0  0  0  0  0  0  0  0  0
0x38 0  0  0  0  0  0  0  0  0
0x40 0  0  0  0  0  0  0  0  0
0x48 0  0  0  0  0  0  0  0  0
```

0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
0x0  0  0  0  0  0  0  0  0  0
0x8  0  0  0  0  0  0  0  0  0
0x10 0  0  0  0  0  0  0  0  0
0x18 0  0  0  0  0  0  0  0  0
0x20 0  0  0  0  0  0  0  0  0
0x28 0  0  0  0  0  0  0  0  0
0x30 0  0  0  0  0  0  0  0  0
0x38 0  0  0  0  0  0  0  0  0
0x40 0  0  0  0  0  0  0  0  0
0x48 0  0  0  0  0  0  0  0  0

`0xB + 1 = 0xC`
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 5?

- Vote in Ed Lessons

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

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Data (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>5</td>
</tr>
<tr>
<td>0x104</td>
<td>10</td>
</tr>
</tbody>
</table>

p = 0x100 + sizeof(int) + 4 = 0x104

5: *p = a[1] = 0x10

0x10 + 0x1 = 0x11
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

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' 'l' 'e' 'a' 's' 'e' 'v' 'o' 't' 'e' '!' '\0'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Endianness and Strings

\texttt{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

- **String literal**
  - \texttt{0x31 = 49 decimal = ASCII ‘1’}

- **IA32, x86-64 (little-endian)**
  - \texttt{0x00 31 \rightarrow 31 0x00 ‘1’}
  - \texttt{0x01 32 \rightarrow 32 0x01 ‘2’}
  - \texttt{0x02 33 \rightarrow 33 0x02 ‘3’}
  - \texttt{0x03 34 \rightarrow 34 0x03 ‘4’}
  - \texttt{0x04 35 \rightarrow 35 0x04 ‘5’}
  - \texttt{0x05 00 \rightarrow 00 0x05 ‘\0’}

- **SPARC (big-endian)**
  - \texttt{31 0x00 \rightarrow \texttt{‘1’}}
  - \texttt{32 0x01 \rightarrow \texttt{‘2’}}
  - \texttt{33 0x02 \rightarrow \texttt{‘3’}}
  - \texttt{34 0x03 \rightarrow \texttt{‘4’}}
  - \texttt{35 0x04 \rightarrow \texttt{‘5’}}
  - \texttt{00 0x05 \rightarrow \texttt{‘\0’}}
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  0x40
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)