Instructor:
Justin Hsia

Teaching Assistants:
Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan

http://xkcd.com/138/
Administrivia

- Lab 0 due today @ 11:59 pm
  - You will be revisiting this program throughout this class!

- Homework 1 due Wednesday
  - Reminder: autograded, 20 tries, no late submissions

- Lab 1a released today, due next Monday (10/8)
  - Pointers in C
  - Reminder: last submission graded, individual work
Late Days

- All submissions due at 11:59 pm
  - Count lateness in days (even if just by a second)
    - Special: weekends count as one day
  - No submissions accepted more than two days late

- You are given 5 late day tokens for the whole quarter
  - Tokens can only apply to Labs (not HW)
  - No benefit to having leftover tokens

- 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
Review Questions

1) If the word size of a machine is 64-bits, which of the following is usually true? (pick all that apply)
   a) 64 bits is the size of a pointer  \( \boxed{T} \)
   b) 64 bits is the size of an integer  \( \boxed{F} \)  \( (32 \text{ bits} = 4 \text{ bytes}) \)
   c) 64 bits is the width of a register  \( \boxed{T} \)

2) (True/False) By looking at the bits stored in memory, I can tell if a particular 4-bytes is being used to represent an integer, floating point number, or instruction.

3) If the size of a pointer on a machine is 6 bits, the address space is how many bytes?

\[
\begin{align*}
\text{represent} & \quad 2^6 \text{ things} \\
\rightarrow & \quad 2^6 \text{ addresses} \\
\rightarrow & \quad 2^6 \text{ bytes of data} = 64 \text{ B}
\end{align*}
\]
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

- `&` = “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)`? It returns the value stored in `y` (equivalent to just using `y`).

* is also used with variable declarations
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
A7    00    32    00
00    01    29    F3
EE    EE    EE    EE
EE    EE    EE    EE
FA    CE    CA    FE
26    00    00    00
00    00    10    00
00    00    00    00
FF    00    F4    96
DE    AD    BE    EF
00    00    00    00
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>00</td>
<td>01</td>
<td>29</td>
<td>F3</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```c
int x, y;

x = 0;
y = 0x3CD02700;
```

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

& = "address of"
*
= "dereference"

little endian!

least significant byte

0x00 0x01 0x02 0x03

0x00 0x00 0x00 0x00
0x04 0x08 0x0C 0x10
0x14 0x18 0x1C 0x20
0x24

X

Y
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

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

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

& = “address of”
* = “dereference”

Pointer arithmetic
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 \( z \)
- \( \star z = y; \)
  - Get value of \( y \), put in address stored in \( z \)
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
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
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;`  
            `p[1] = 0xB;`  
            `*(p+1) = 0xB;`  
            `p = p + 2;`  

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

equivalent `p[1] = 0xB;`  
            `*(p+1) = 0xB;`  
pointer arithmetic:  
                   `0x10 + 1 \rightarrow 0x19`  
                   `p = p + 2;`  
                   `0x10 + 2 \rightarrow 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

\[
p[i] \leftrightarrow *(p + i)
\]
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;`
- `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

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;`

Store at `0x18`:
- `*p = 0xB + 1 = 0xC;`

(no pointer arithmetic)
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?
- Vote at http://PollEv.com/justinh

```
void main() {
    int a[] = {5,10};
    int* p = a;
    p = p + 1; // sizeof(int) = 4
    *p = *p + 1;
}
```

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

---

1. **(A)** 101 10 5 10 then 101 11 5 11
2. **(B)** 104 10 5 10 then 104 11 5 11
3. **(C)** 100 6 6 10 then 101 6 6 10
4. **(D)** 100 6 6 10 then 104 6 6 10
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>32</th>
<th>space</th>
<th>48</th>
<th>0</th>
<th>64</th>
<th>@</th>
<th>80</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>!</td>
<td>49</td>
<td>1</td>
<td>65</td>
<td>A</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
<td>50</td>
<td>2</td>
<td>66</td>
<td>B</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
<td>51</td>
<td>3</td>
<td>67</td>
<td>C</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
<td>52</td>
<td>4</td>
<td>68</td>
<td>D</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
<td>53</td>
<td>5</td>
<td>69</td>
<td>E</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
<td>54</td>
<td>6</td>
<td>70</td>
<td>F</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
<td>55</td>
<td>7</td>
<td>71</td>
<td>G</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>56</td>
<td>8</td>
<td>72</td>
<td>H</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
<td>57</td>
<td>9</td>
<td>73</td>
<td>I</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
<td>58</td>
<td>:</td>
<td>74</td>
<td>J</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
<td>59</td>
<td>;</td>
<td>75</td>
<td>K</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>60</td>
<td>&lt;</td>
<td>76</td>
<td>L</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
<td>61</td>
<td>=</td>
<td>77</td>
<td>M</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
<td>62</td>
<td>&gt;</td>
<td>78</td>
<td>N</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
<td>63</td>
<td>?</td>
<td>79</td>
<td>O</td>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>124</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>

ASCII: American Standard Code for Information Interchange
Null-Terminated Strings

- **Example:** "Donald Trump" stored as a 13-byte array

  ![Hexadecimal and decimal representation of Donald Trump's name](image)

- Last character followed by a 0 byte ( '\0' )
  (a.k.a. "null terminator")
  - Must take into account when allocating space in memory
  - Note that '0' ≠ '\0' (i.e. character 0 has non-zero value)

- How do we compute the length of a string?
  - Traverse array until null terminator encountered
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

---

**C (char = 1 byte)**

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

- `s[0] = 0x31 = 49 decimal = ASCII ‘1’`
- `s[5] = 0x00 = \0`
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	0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p`  Print pointer
- `	`  Tab
- `%x`  Print value as hex
- `\n`  New line
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\t0x%.2x\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 = 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

```plaintext
int x = 12345;
0x7ffffff7f71dbc 0x39
0x7ffffff7f71dbd 0x30
0x7ffffff7f71dbe 0x00
0x7ffffff7f71dbf 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)