Instructor:
Justin Hsia

Teaching Assistants:
Andrew Hu
Antonio Castelli
Cosmo Wang
Diya Joy
Ivy Yu
Kaelin Laundry
Maurice Montag
Melissa Birchfield
Millicent Li
Suraj Jagadeesh

http://xkcd.com/138/
Administrivia

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

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

- Lab 1a released today, due next Monday (10/7)
  - 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
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
   b) 64 bits is the size of an integer
   c) 64 bits is the width of a register

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?
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
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;
```

* & is also used with variable declarations

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`”)

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)`?
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>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7</td>
<td>00</td>
<td>32</td>
<td>00</td>
</tr>
<tr>
<td>0x04</td>
<td>00</td>
<td>01</td>
<td>29</td>
<td>F3</td>
</tr>
<tr>
<td>0x08</td>
<td>EE</td>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>0x0C</td>
<td>FA</td>
<td>CE</td>
<td>CA</td>
<td>FE</td>
</tr>
<tr>
<td>0x10</td>
<td>26</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x14</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td>FF</td>
<td>00</td>
<td>F4</td>
<td>96</td>
</tr>
<tr>
<td>0x20</td>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
</tr>
<tr>
<td>0x24</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
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)

```
0x00 0x01 0x02 0x03
0x04 01 29 F3
0x08 0x0C 0x10 0x14
0x18 0x1C 0x20 0x24
```

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;`
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”
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

- \textbf{int } x, y;

- \textbf{x} = 0;

- \textbf{y} = 0x3CD02700;

- \textbf{x} = \textbf{y} + 3;
  - Get value at \textbf{y}, add 3, store in \textbf{x}

- \textbf{int* } z = \&\textbf{y} + 3;
  - Get address of \textbf{y}, “add 3”, store in \textbf{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`

```c
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`
- `*z = y;`
  - Get value of `y`, put in address stored in `z`
Arrays in C

Declaration: `int a[6];`

- **element type**
- **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):

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

```
a[0] 0x00
a[1] 0x08
a[2] 0x10
a[3] 0x18
a[4] 0x20
```

```
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7
0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
```
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:  
- `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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>5F 01 00 00</td>
</tr>
<tr>
<td>0x18</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>AD 0B 00 00</td>
</tr>
<tr>
<td>0x28</td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
</tbody>
</table>
Arrays in C

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

Indexing: \begin{align*} a[0] &= 0x015f; \\
         a[5] &= a[0]; \end{align*}

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

Pointers: \begin{align*} \textbf{int}* \quad p; \\
    &\quad \text{equivalent} \\
    &\quad \quad \begin{cases} p = a; & a[0] \quad 0x00 \\
                     p = \&a[0]; & a[2] \quad 0x10 \\
                     *p = 0xA; & a[4] \quad 0x18 \end{cases} \end{align*}

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

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

\&\textbf{a}[i] is the address of \textbf{a}[0] plus \textbf{i} times the element size in bytes
Arrays in C

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

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

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

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

Pointers: \texttt{int* } p;
\begin{equation}
\begin{aligned}
\text{equivalent} & \quad \{ p = a; \quad a[0] \\
\text{equivalent} & \quad \{ p = \&a[0]; \quad a[2] \\
\text{equivalent} & \quad \{ *p = 0xA; \quad a[4] \\
\end{aligned}
\end{equation}

array indexing = address arithmetic (both scaled by the size of the type)
\begin{equation}
\begin{aligned}
\text{equivalent} & \quad \{ p[1] = 0xB; \quad *(p+1) = 0xB; \\
\text{equivalent} & \quad \{ p = p + 2; \\
\end{aligned}
\end{equation}
# Arrays in C

## Declaration: `int a[6];`

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

## No bounds
- `a[6] = 0xBADE;`

## checking:
- `a[-1] = 0xBADE;`

## Pointers:
- `int* p;`
  - Equivalent:
    - `p = a;`
    - `p = &a[0];`
    - `*p = 0xA;`
    - `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

---

![Memory Map](image)
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

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

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

(A) 101 10 5 10 then 101 11 5 11
(B) 104 10 5 10 then 104 11 5 11
(C) 100 6 6 10 then 101 6 6 10
(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>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>”</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>’</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>;</td>
</tr>
<tr>
<td>59</td>
<td>:</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>o</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>
<tr>
<td>126</td>
<td>~</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>68</th>
<th>111</th>
<th>110</th>
<th>97</th>
<th>108</th>
<th>100</th>
<th>32</th>
<th>84</th>
<th>114</th>
<th>117</th>
<th>109</th>
<th>112</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>0x44</td>
<td>0x6F</td>
<td>0x6E</td>
<td>0x61</td>
<td>0x6C</td>
<td>0x64</td>
<td>0x20</td>
<td>0x54</td>
<td>0x72</td>
<td>0x75</td>
<td>0x6D</td>
<td>0x70</td>
<td>0x00</td>
</tr>
<tr>
<td>Text</td>
<td>D o n a l d</td>
<td>T r u m p</td>
<td>\0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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";
```

- **String literal**
  - 0x31 = 49 decimal = ASCII ‘1’

### 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

#### Examples

<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 31</td>
</tr>
<tr>
<td>0x01 32</td>
<td>0x01 32</td>
</tr>
<tr>
<td>0x02 33</td>
<td>0x02 33</td>
</tr>
<tr>
<td>0x03 34</td>
<td>0x03 34</td>
</tr>
<tr>
<td>0x04 35</td>
<td>0x04 35</td>
</tr>
<tr>
<td>0x05 00</td>
<td>0x05 00</td>
</tr>
</tbody>
</table>

C (char = 1 byte)
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");
}
```

**printf directives:**
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `
` 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

```c
int x = 12345;
0x7fffb7f71dbc 0x39
0x7fffb7f71dbd 0x30
0x7fffb7f71dbe 0x00
0x7fffb7f71dbf 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)