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  
   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
- 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

```
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""")
- 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

```
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
```
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)

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></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></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>

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!
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”

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
</tr>
<tr>
<td>0x08</td>
<td>0x08</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x10</td>
<td>0x10</td>
</tr>
<tr>
<td>0x14</td>
<td>0x14</td>
</tr>
<tr>
<td>0x18</td>
<td>0x18</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x1C</td>
</tr>
<tr>
<td>0x20</td>
<td>0x20</td>
</tr>
<tr>
<td>0x24</td>
<td>0x24</td>
</tr>
</tbody>
</table>

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

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

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

\& = “address of”
\* = “dereference”
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 name `a` returns the array's address. In a 64-bit example, pointers are 64-bits wide.
Arrays in C

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

Indexing: \begin{itemize}
\item \texttt{a[0] = 0x015f;}
\item \texttt{a[5] = a[0];}
\end{itemize}

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;`

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];`
- `a[6] = 0xBAD;`
- `a[-1] = 0xBAD;`

No bounds checking:  
- `a[6] = 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>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td>0A</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0A</td>
<td>00</td>
</tr>
<tr>
<td>0A</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```
# 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

---

- **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: \texttt{int} \ a[6];

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

No bounds \quad \texttt{a[6]} = 0xBAD;
checking: \quad \texttt{a[-1]} = 0xBAD;

Pointers: \quad \texttt{int}\^* \ p; \quad \texttt{p} = \texttt{a}; \quad \texttt{p} = \&\texttt{a[0]}; \quad \*\texttt{p} = 0xA;

\begin{itemize}
  \item \texttt{p[1]} = 0xB;
  \item \*(\texttt{p+1}) = 0xB;
  \item \texttt{p} = \texttt{p} + 2;
  \item \*\texttt{p} = \texttt{a[1]} + 1;
\end{itemize}
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?


<table>
<thead>
<tr>
<th>p</th>
<th>*p</th>
<th>a[0]</th>
<th>a[1]</th>
<th>p</th>
<th>*p</th>
<th>a[0]</th>
<th>a[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>101</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>then</td>
<td>101</td>
</tr>
<tr>
<td>(B)</td>
<td>104</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>then</td>
<td>104</td>
</tr>
<tr>
<td>(C)</td>
<td>100</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>then</td>
<td>101</td>
</tr>
<tr>
<td>(D)</td>
<td>100</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>then</td>
<td>104</td>
</tr>
</tbody>
</table>
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>&quot;</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>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 T r u m p \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>

- 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
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 \t 0x%.2x \n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p` Print pointer
- `\t` 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

```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)