Memory, Data, & Addressing II
CSE 351 Spring 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/138/

32-bit addresses
Administrivia

- Questions doc for today: [https://tinyurl.com/CSE351-6-26](https://tinyurl.com/CSE351-6-26)
- Assignments Overview
  - Lab 0 due Tonight (6/26) – 11:59pm
  - hw2 due Monday (6/29) – 10:30am
  - hw3 due Wednesday (7/1) – 10:30am
  - Lab 1a released today, due a week from Monday (7/6)
    - Suggested Due Date is 7/3 to give time for lab1b (due 7/10)
    - Pointers in C
    - Reminder: last submission graded, individual work

- Study group survey results released today!
  - Can still fill out the survey if interested in finding a group
Late Days

❖ You are given **7 late days** for the whole quarter
  ▪ Late days can only apply to Labs & Unit Summaries
  ▪ No benefit to having leftover late days

❖ Count lateness in *days* (even if just by a second)
  ▪ **Special**: weekends count as *one day*
  ▪ No submissions accepted more than two days late

❖ The late penalty for using more than 7 late days is a
  20% deduction of your score per excess day
  ▪ Only late work is 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
Where We Left Off: Byte Ordering

❖ Big-endian (SPARC, z/Architecture)
  ▪ Least significant byte has highest address

❖ Little-endian (x86, x86-64)
  ▪ Least significant byte has lowest address

❖ Bi-endian (ARM, PowerPC)
  ▪ Endianness can be specified as big or little

❖ Example: 4-byte data 0xa1b2c3d4 at address 0x100

\[
\begin{array}{cccc}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
\text{Big-Endian} & & a1 & b2 & c3 & d4 \\
\text{Little-Endian} & d4 & c3 & b2 & a1 \\
\end{array}
\]
Byte Ordering Examples

```c
int x = 12345;
// or x = 0x3039;
```

```c
long int y = 12345;
// or y = 0x3039;
```

(A `long int` is the size of a word)
Polling Question

- We store the value 0x01 02 03 04 as a word at address 0x100 in a big-endian, 64-bit machine
- What is the byte of data stored at address 0x104?
  - Vote at http://pollev.com/pbjones

A. 0x04
B. 0x40
C. 0x01
D. 0x10
E. We’re lost...
Endianness

❖ Endianness only applies to memory storage

❖ Often programmer can ignore endianness because it is handled for you
  ▪ Bytes wired into correct place when reading or storing from memory (hardware)
  ▪ Compiler and assembler generate correct behavior (software)

❖ Endianness still shows up:
  ▪ Logical issues: accessing different amount of data than how you stored it (e.g. store int, access byte as a char)
  ▪ Need to know exact values to debug memory errors
  ▪ Manual translation to and from machine code (in 351)
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;
```

- 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)]
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”
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?
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`
  - The target of a pointer is also a location
- `*z = y;`
  - Get value of `y`, put in address stored in `z`
Arrays in C

Declaration: `int a[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)
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 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
Arrays in C

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

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

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

Pointers: \texttt{int* p;}

\begin{align*}
\text{equivalent} & \quad p = a; \\
& \quad p = \&a[0]; \\
& \quad *p = 0xA;
\end{align*}

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

\begin{align*}
\text{equivalent} & \quad p[1] = 0xB; \\
& \quad *\left(p + 1\right) = 0xB; \\
& \quad p = p + 2;
\end{align*}

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;`

  equivalent
  ```
  p = a;
  p = &a[0];
  *p = 0xA;
  ```

  equivalent
  ```
  p[1] = 0xB;
  *(p+1) = 0xB;
  p = p + 2;
  ```

  equivalent
  ```
  *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
**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/pbjones](http://pollev.com/pbjones)

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

<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>then</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>(B)</td>
<td>104</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>then</td>
<td>104</td>
<td>11</td>
</tr>
<tr>
<td>(C)</td>
<td>100</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>then</td>
<td>101</td>
<td>6</td>
</tr>
<tr>
<td>(D)</td>
<td>100</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>then</td>
<td>104</td>
<td>6</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>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>
<tr>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>
Null-Terminated Strings

- **Example:** “Ice Creamery" stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>0x49</td>
<td>Ice</td>
</tr>
<tr>
<td>99</td>
<td>0x63</td>
<td>c</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>e</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>C</td>
</tr>
<tr>
<td>67</td>
<td>0x43</td>
<td>reamery</td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td>y</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>0</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>0x6d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>0x79</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x00</td>
<td>\0</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

```
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)

- **IA32, x86-64 (little-endian)**
  - Little-endian byte ordering

- **SPARC (big-endian)**
  - Big-endian byte ordering

<table>
<thead>
<tr>
<th>IA32</th>
<th>SPARC</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>

- 0x31 = 49 decimal = ASCII ‘1’
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
- `\t` Tab
- `%x` Print value as hex
- `\n` New line
Examing 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

```
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)