Memory, Data, & Addressing II
CSE 351 Winter 2017

http://xkcd.com/371/
Administrivia

- Lab 0 due tomorrow @ 5pm
  - Credit/no credit – we’ll talk about topics in depth later
- Lab 1 released later today @ 5pm
- Survey results:
  - More detail how computers work, learn C, get a CE/CS major 😊
  - People from most continents!
Review

- An **address** is a location in memory
- A **pointer** is a data object that holds an address
  - Address can point to *any* data
- Pointer stored at **0x48** points to address **0x38**
  - Pointer to a pointer!
- Is the data stored at **0x08** a pointer?

64-bit example (pointers are 64-bits wide)
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 sets 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 memory 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
FA CE CA FE
26  00  00  00
  00  00  10  00
  01  00  00  00
FF  00  F4  96
DE AD BE EF
  00  00  00  00
```

0x00 0x04 X
0x08 0x0C
0x10 0x14
0x18 0x1C Y
0x20 0x24
Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *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 memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory 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 $x$

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 memory *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`
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

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

- Get value at \( y \), add 3, store in \( x \)
- Get address of \( y \), “add 3”, store in \( z \)

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

\& = “address of”
* = “dereference”

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>y</td>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>z</td>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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**: `int` (integer)
- **Name**: `a`
- **Number of elements**: 6

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

*a* is a name for 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` is a name for the array’s address.
- The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

![Array Memory Layout](image_url)
### Arrays in C

**Declaration:**

```c
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` is a name for the array’s address.

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

---

**Example:**

- The address of `a[0]` is AD 0B 00 00.
- The address of `a[4]` is AD 0B 00 00.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x08</td>
<td>0x08</td>
</tr>
<tr>
<td>0x10</td>
<td>0x10</td>
</tr>
<tr>
<td>0x18</td>
<td>0x18</td>
</tr>
<tr>
<td>0x20</td>
<td>0x20</td>
</tr>
<tr>
<td>0x28</td>
<td>0x28</td>
</tr>
<tr>
<td>0x30</td>
<td>0x30</td>
</tr>
<tr>
<td>0x38</td>
<td>0x38</td>
</tr>
<tr>
<td>0x40</td>
<td>0x40</td>
</tr>
<tr>
<td>0x48</td>
<td>0x48</td>
</tr>
</tbody>
</table>
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

No bounds \( a[6] = 0xBAD; \)
checking: \( a[-1] = 0xBAD; \)

Pointers: \( \text{int* } p; \)
\( p = a; \)
\( p = \&a[0]; \)
\( *p = 0xA; \)

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

\( a \) is a name for the array’s address

The address of \( a[i] \) 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;`  
`p = a;`  
`p = &a[0];`  
`*p = 0xA;`

equivalent

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

equivalent

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

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

`a` is a name for the array’s address
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

No bounds \( a[6] = 0xBAD; \)
checking: \( a[-1] = 0xBAD; \)

Pointers: \( \text{int } *p; \)
\( \quad p = a; \)
\( \quad p = &a[0]; \)
\( \quad *p = 0xA; \)

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

\( p[1] = 0xB; \)
\( \quad * (p+1) = 0xB; \)
\( p = p + 2; \)
\( \quad *p = a[1] + 1; \)

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

\( a \) is a name for the array’s address

The address of \( a[i] \) is the address of \( a[0] \)
plus \( i \) times the element size in bytes
## 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 Code</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:** “Life is good” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal:</th>
<th>76</th>
<th>105</th>
<th>102</th>
<th>101</th>
<th>32</th>
<th>105</th>
<th>115</th>
<th>32</th>
<th>103</th>
<th>111</th>
<th>111</th>
<th>100</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hex:</strong></td>
<td>0x4c</td>
<td>0x69</td>
<td>0x66</td>
<td>0x65</td>
<td>0x20</td>
<td>0x69</td>
<td>0x73</td>
<td>0x20</td>
<td>0x67</td>
<td>0x6f</td>
<td>0x6f</td>
<td>0x64</td>
<td>0x00</td>
</tr>
<tr>
<td><strong>Text:</strong></td>
<td>L</td>
<td>i</td>
<td>f</td>
<td>e</td>
<td>i</td>
<td>s</td>
<td>g</td>
<td>o</td>
<td>o</td>
<td>d</td>
<td>\0</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
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

- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading zeros)
  - Unicode can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
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 a = 12345; // 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((char *) &a, sizeof(int));
```

- **Result (Linux x86-64):**
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

```c
int a = 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)