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/
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

<table>
<thead>
<tr>
<th>Big-Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td>b2</td>
<td>c3</td>
<td>d4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little-Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d4</td>
<td>c3</td>
<td>b2</td>
<td>a1</td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Examples

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

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](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

---

- **Declares a variable, `ptr`, that is a pointer to (i.e. holds the address of) an `int` in memory**

  ```c
  int* ptr;
  ```

- **Declares two variables, `x` and `y`, that hold ints, and initializes them to 5 and 2, respectively**

  ```c
  int x = 5;
  int y = 2;
  ```

- **Sets `ptr` to the address of `x` ("`ptr` points to `x"”)**

  ```c
  ptr = &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"`**

  ```c
  y = 1 + *ptr;
  ```

- **`*` is also used with variable declarations**

  ```c
  int* &y;
  ```

---

What is `*(&y)`?

- Value stored in the address of `y`... so `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
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

```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

- \textbf{int x, y;}
- \textbf{x = 0;}
- \textbf{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 \( x \)

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td></td>
<td>03</td>
<td>27</td>
<td>D0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3C</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;`
- `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 \( z \)

\[ \begin{array}{cccc}
0x00 & 0x01 & 0x02 & 0x03 \\
0x04 & 0x08 & 0x0C & 0x10 \\
0x14 & 0x18 & 0x1C & 0x20 \\
0x24 & & & \\
\end{array} \]

\& = “address of”
\* = “dereference”
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int) = 4 bytes`
  - `int* z = &y + 3;` — “add 3 int-sized chunks to &y”
    - 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

- \textbf{int} \ x, \ y;
- \ x = 0;
- \ y = 0x3CD02700;
- \ x = y + 3;
  - Get value at \ y, add 3, store in \ x
- \textbf{int}^{*} \ z = \&y + 3;^{4}
  - 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”

The target of a pointer is also a location
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.

`a` (array name) returns the array’s address.

64-bit example (pointers are 64-bits wide)

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

- `a[1]`
- `a[3]`
- `a[5]`

- `a[0]` 0x10
- `a[2]` 0x18
- `a[4]` 0x20
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 checking: `a[6] = 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 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.

```
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
0x00 0x01 0x00 0x00 0x5F 0x00 0x00 0x00
0x00 0x01 0x00 0x00 0x5F 0x00 0x00 0x00
AD 0B 00 00
```

`&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;`

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

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

\[
p[i] = \ast(p + i)
\]
Arrays in C

Declaration: `int a[6];`

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

No bounds

checking: 
- `a[-1] = 0xBAD;`

Pointers: `int* p;`

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

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;`

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>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x8</td>
<td>0x10</td>
<td>0x18</td>
<td>0x20</td>
<td>0x28</td>
<td>0x30</td>
<td>0x38</td>
</tr>
<tr>
<td>0x1</td>
<td>0x9</td>
<td>0xA</td>
<td>0xB</td>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
</tr>
<tr>
<td>0x2</td>
<td>0xA</td>
<td>0xB</td>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td></td>
</tr>
<tr>
<td>0x3</td>
<td>0xB</td>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x4</td>
<td>0xC</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x5</td>
<td>0xD</td>
<td>0xE</td>
<td>0xF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x6</td>
<td>0xE</td>
<td>0xF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7</td>
<td>0xF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Store this value at address `p` points to(0x18)
**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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Data (decimal)</th>
<th>Address (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>5</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td>10</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>100</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Options:**

- (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>&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:** "Ice Creamery" stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>99</td>
<td>101</td>
<td>32</td>
<td>67</td>
<td>114</td>
<td>101</td>
<td>97</td>
<td>109</td>
<td>101</td>
<td>114</td>
<td>121</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x49</td>
<td>0x63</td>
<td>0x65</td>
<td>0x20</td>
<td>0x43</td>
<td>0x72</td>
<td>0x65</td>
<td>0x61</td>
<td>0x6d</td>
<td>0x65</td>
<td>0x72</td>
<td>0x79</td>
<td>0x00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Text</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I c e</td>
<td>C r e a m e r y</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";
```

 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

String literal

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>31 0x00 '1'</td>
</tr>
<tr>
<td>0x01 32</td>
<td>32 0x01 '2'</td>
</tr>
<tr>
<td>0x02 33</td>
<td>33 0x02 '3'</td>
</tr>
<tr>
<td>0x03 34</td>
<td>34 0x03 '4'</td>
</tr>
<tr>
<td>0x04 35</td>
<td>35 0x04 '5'</td>
</tr>
<tr>
<td>0x05 00</td>
<td>00 0x05 ' \0 '</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
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

```
int x = 12345;
0x7ffffff71dbc  0x39
0x7ffffff71dbd  0x30
0x7ffffff71dbe  0x00
0x7ffffff71dbf  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)
Assignment in C - Handout

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value
  - Store RHS value at LHS location

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
- `int* z = &y + 3;`
- `*z = y;`
Arrays in C - Handout

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;`
- `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)

Equivalent:
```c
p[1] = 0xB;
*(p+1) = 0xB;
p = p + 2;
*p = a[1] + 1;
```
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?