Memory, Data, & Addressing II
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis

http://xkcd.com/371/
Administrivia

- Start-of-Course survey [Catalyst] due Thursday (3/30)
- Everyone has VM or access to attu? LET US KNOW if not
- Section - Room changes for 9:30 and 10:30 sections
- Lab 0, due Monday (4/3) @ 11:59pm
- Homework 1, due Monday (4/3) @ 11:59pm
- Readings in CSAPP – see schedule
- Office Hours – see schedule – LET US KNOW if you cannot make any of our posted office hours
- Lab 1 – coming soon!
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)`?

* is also used with variable declarations
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

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>00</td>
<td>32</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>29</td>
<td>F3</td>
</tr>
<tr>
<td>EE</td>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>FA</td>
<td>CE</td>
<td>CA</td>
<td>FE</td>
</tr>
<tr>
<td>26</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FF</td>
<td>00</td>
<td>F4</td>
<td>96</td>
</tr>
<tr>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Note: `x` is at address 0x04, and `y` is at 0x18.
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

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

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

```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

32-bit example (pointers are 32-bits wide)
& = “address of”
* = “dereference”

Declaration: initially contains garbage
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;

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”

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”

```
0x00 0x01 0x02 0x03
03 27 D0 3C
0x00 X
0x04
0x08
0x0C
0x10
0x14
0x18 Y
0x1C
0x20
0x24 Z
```
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”

The target of a pointer is also a memory location
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 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
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` 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.

```
  0x0  0x1  0x2  0x3  0x4  0x5  0x6  0x7
  0x8  0x9  0xA  0xB  0xC  0xD  0xE  0xF
0x00 AD  0B  00  00
0x01 5F  01  00  00
0x02 00  00  00  00
0x03 00  00  00  00
0x04 00  00  00  00
0x05 00  00  00  00
0x06 00  00  00  00
0x07 00  00  00  00
```
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: \[ \text{int* } p; \]
\[ p = a; \]
\[ p = &a[0]; \]
\[ *p = 0xA; \]

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

\[ p = &a[0]; \]
\[ *p = 0xA; \]

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

\[ p = p + 2; \]

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

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

\[ p = p + 2; \]
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;
```

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

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

```
array indexing = address arithmetic

\[ p[1] = 0xB; \]

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

\[ p = p + 2; \]

\[ *p = a[1] + 1; \]
```
Fill in the blanks!

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

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

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

  *p = a[1] + 1;
```
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

1. `void main() {`
2. `int a[] = {5,10};`
3. `int *p = a;`
4. `p = p + 1;`
5. `*p = *p + 1;`
6. `}

<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>

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

- **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)

### 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>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’
- 0x00 \0 = ASCII \0
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
- `\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 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)