Announcements

- **Lab 0 is due at 5pm. How is it going?**
  - Arrays yesterday in section and today in lecture.

- **Stuck with something?**
  - Post on the discussion board; office hours after lecture.

- **Lab 1 posted, due Monday, July 8.**

- **Vote for next week’s section time today.**
  - See website or email.
  - Current favorite (13/17) : Friday immediately after lecture.

- **Reading for each lecture posted on the website.**
  - I won’t be announcing these in general.
  - Memory, Data, and Addressing: CS:APP sections 2.0 – 2.1
  - Integer Representation: CS:APP sections 2.2 - 2.3

- **You can call me Ben.**
Today

- Brief review of pointers
- Arrays and address arithmetic
- Strings as arrays
- Boolean algebra and bitwise manipulations
- Start integer representations
Assignment in C (review)

- 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 = 0x0000015F;
- y = 0x3CD02700;
- int* z;
- z = &y;
  - // What does this do?

& = ‘address of’
* = ‘value at address’
or ‘dereference’

(little endian)
Assignment in C (review)

- **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 = 0x0000015F;`
- `y = 0x3CD02700;`
- `int* z;`
- `z = &y;`
  - // Get address of y, put it in z.
- `*z = x;`
  - // What does this do?

\& = ‘address of’
* = ‘value at address’
or ‘dereference’

(little endian)
Assignment in C (review)

- 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 = 0x0000015F;
- y = 0x3CD02700;
- int* z;
- z = &y;
  - // Get address of y, put it in z.
- *z = x;
  - // Get value in x, put it in memory
  - // at the address z points to.

& = ‘address of’
* = ‘value at address’ or ‘dereference’
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, not a pointer to the array.
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, not a pointer to the array.
- The address of `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>
</tr>
</thead>
<tbody>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Arrays in C

Declaration: int a[6];
Indexing: a[0] = 0x015f; a[5] = a[0];
No bounds check: a[6] = 0xBAD; 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, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.

int a[6];

| 5F 01 00 00 | 0x00 | a[0] |
| 5F 01 00 00 | 0x04 | a[1] |
|            | 0x08 |      |
|            | 0x0C |      |
|            | 0x10 |      |
|            | 0x14 |      |
|            | 0x18 |      |
| AD 0B 00 00 | 0x1C | a[5] |
|            | 0x20 |      |
|            | 0x24 |      |
Arrays in C

Declaration:  int a[6];

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

No bounds check:  a[6] = 0xBAD;
a[-1] = 0xBAD;

Pointers:  int* p;
  p = a;
p = &a[0];

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

p is a name for the array’s address, not a pointer to the array.

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] = 0xBA0D;
check:       a[-1] = 0xBA0D;

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, not a pointer to the array.

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>a[0]</td>
</tr>
<tr>
<td>0x08</td>
<td>a[1]</td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>a[5]</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td>p</td>
</tr>
</tbody>
</table>

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

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

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

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

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, not a pointer to the array.

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;
check:          a[-1] = 0xBAD;

Pointers:       int* p;
                p = a;
                p = &a[0];
                *p = 0x1A;
                p[1] = 0xB;

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array’s address, not a pointer to the array.
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 check: a[6] = 0xBAD;
a[-1] = 0xBAD;

Pointers: int* p;

\[
\begin{align*}
&\text{p = a;} \\
&\text{p = &a[0];} \\
&\text{*p = 0xA;} \\
&\text{p[1] = 0xB;} \\
\end{align*}
\]

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array’s address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.

Array indexing = address arithmetic
Both are scaled by the size of the type.
Arrays in C

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

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

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

Pointers: \[
\begin{align*}
\text{int*} \ p; \\
p &= a; \\
p &= &a[0]; \\
*p &= 0xA;
\end{align*}
\]

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

\textit{array indexing = address arithmetic}

Both are scaled by the size of the type.

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

\( a \) is a name for the array’s address, not a pointer to the array.

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 check:    a[6] = 0xBAD;
                       a[-1] = 0xBAD;

Pointers:       int* p;
                p = a;  // equivalent to p = &a[0];
                *p = 0xA;

Array indexing = address arithmetic

Both are scaled by the size of the type.

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

p is a name for the array’s address, not a pointer to the array.

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

Declaration: \hspace{1cm} \texttt{int a[6];}

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

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

Pointers:
\begin{align*}
\texttt{int* p;} \\
p &= a; \\
p &= \&a[0]; \\
*p &= 0xA;
\end{align*}

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

\textit{array indexing = address arithmetic}
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.
\begin{itemize}
\item \texttt{a} is a name for the array’s address, not a pointer to the array.
\end{itemize}

The address of \texttt{a[i]} is the address of \texttt{a[0]} plus \(i\) times the element size in bytes.

\begin{center}
\begin{tabular}{cccc}
| AD | OB | 00 | 00 |
| 0x00 |
| 0A | 00 | 00 | 00 |
| 0x04 |
| 0B | 00 | 00 | 00 |
| 0x08 |
| \texttt{a[0]} & | & | |
| \texttt{a[1]} & | & | |
| \texttt{0x0C} & | & | |
| \texttt{0x10} & | & | |
| \texttt{0x14} & | & | |
| \texttt{0x18} & | & | |
| \texttt{a[5]} & | & | |
| \texttt{0x1C} & | & | |
| \texttt{0x20} & | & | |
| \texttt{0x24} & | & | |
| \texttt{p} & | & | |
\end{tabular}
\end{center}
Arrays in C

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

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

No bounds check:
\( \texttt{a[6] = 0xBAD;} \)
\( \texttt{a[-1] = 0xBAD;} \)

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

\( \texttt{p[1] = 0xB;} \)
\( \texttt{*(p + 1) = 0xB;} \)
\( \texttt{p = p + 2;} \)

**array indexing = address arithmetic**
Both are scaled by the size of the type.

\( \texttt{*p = a[1] + 1;} \)

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

\( \texttt{a} \) is a name for the array’s address, not a pointer to the array.

The address of \( \texttt{a[i]} \) is the address of \( \texttt{a[0]} \) plus \( i \) times the element size in bytes.
Representing strings

- A C-style string is represented by an array of bytes (char).
  - Elements are one-byte ASCII codes for each character.
  - ASCII = American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>32</th>
<th>space</th>
<th>48</th>
<th>0</th>
<th>64</th>
<th>@</th>
<th>80</th>
<th>P</th>
<th>96</th>
<th>`</th>
<th>112</th>
<th>p</th>
</tr>
</thead>
<tbody>
<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>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</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>
<td>114</td>
<td>r</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>
<td>115</td>
<td>s</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>
<td>116</td>
<td>t</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>
<td>117</td>
<td>u</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>
<td>118</td>
<td>v</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>
<td>119</td>
<td>w</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>
<td>120</td>
<td>x</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>
<td>121</td>
<td>y</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>
<td>122</td>
<td>z</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>
<td>123</td>
<td>{</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>
<td>124</td>
<td></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>
<td>125</td>
<td>}</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>
<td>126</td>
<td>~</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>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>
Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

```
72 97 114 114 121 32 80 111 116 116 101 114 0
Harry Potter \0
```

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: string[12] = '\0';

- How do we compute the string length?
Endianness and Strings

C (char = 1 byte)
char s[6] = "12345";

Java (char = 2 bytes)
String s = "123";
(not all of the String representation is shown)

- Byte ordering (endianness) is not an issue for 1-byte values.
  - Arrays are not values; 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
typedef char byte;  // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

```c
void show_int (int x) {
    show_bytes((byte*) &x, sizeof(int));
}
```

printf directives:
- %p Print pointer
- \t Tab
- %x Print value as hex
- \n New line
show_bytes Execution Example

```c
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a);    // show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```c
int a = 12345;
0x11ffffffcb8 0x39
0x11ffffffcb9 0x30
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
**Boolean Algebra**

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  - OR: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
  - XOR: $A \^ B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  - NOT: $\sim A = 1$ when $A$ is 0 and vice-versa
  - DeMorgan’s Law: $\sim (A \mid B) = \sim A \& \sim B$

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 \\
\& 01010101 & \mid 01010101 & ^01010101 & ~01010101 \\
01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

- All of the properties of Boolean algebra apply

\[
01010101 \uparrow 01010101 = 00000000
\]

- How does this relate to set operations?
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of $\{0, \ldots, w–1\}$
  - $a_j = 1$ iff $j \in A$
    - $01101001$ \{0, 3, 5, 6\}
    - $01010101$ \{0, 2, 4, 6\}

- **Operations**
  - & Intersection
    - $01000001$ \{0, 6\}
  - | Union
    - $01111101$ \{0, 2, 3, 4, 5, 6\}
  - ^ Symmetric difference
    - $00111100$ \{2, 3, 4, 5\}
  - ~ Complement
    - $10101010$ \{1, 3, 5, 7\}
Bit-Level Operations in C

- \& | ^ ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors

- **Examples (char data type)**
  - \(~0x41 --> 0xBE\)
    - \(~01000001_2 --> 10111110_2\)
  - \(~0x00 --> 0xFF\)
    - \(~00000000_2 --> 11111111_2\)
  - \(0x69 \& 0x55 --> 0x41\)
    - \(01101001_2 \& 01010101_2 --> 01000001_2\)
  - \(0x69 | 0x55 --> 0x7D\)
    - \(01101001_2 | 01010101_2 --> 01111101_2\)

- Many bit-twiddling puzzles in Lab 1
Contrast: Logic Operations in C

- **Contrast to logical operators**
  - `&&`, `||`, `!`
    - 0 is “False”
    - Anything nonzero is “True”
    - Always return 0 or 1
    - Early termination  a.k.a. short-circuit evaluation

- **Examples (char data type)**
  - `!0x41  -->  0x00`
  - `!0x00  -->  0x01`
  - `!!0x41 -->  0x01`
  - `0x69 & & 0x55  -->  0x01`
  - `0x69 | | 0x55  -->  0x01`
  - `p & & *p++`  (avoids null pointer access, null pointer = 0x00000000 )