Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee
Natalie Andreeva
Teagan Horkan

http://xkcd.com/138/
Administrivia

- Homework 1 due tonight @ 11:59 pm
  - Reminder: autograded, 20 tries, no late submissions
- Lab 0 due on Monday (6/25)
  - You will be revisiting this program throughout this class!
- Lab 1a released tonight, due next Friday (6/29)
  - On pointers and C
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  \( T \)
   b) 64 bits is the size of an integer  \( F \)  \( (32 \text{ bits} = 4 \text{ B}) \)
   c) 64 bits is the width of a register  \( T \)

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?
   \( \begin{align*}
   \text{Ob} & \quad \underbrace{\ldots}_{\text{3 bytes}} \\
   \text{represent} & \quad 2^6 \text{ things} \\
   \rightarrow & \quad 2^6 \text{ addresses} \\
   \rightarrow & \quad 2^6 \text{ bytes of data} \\
   & \quad \underbrace{64 \text{ B}}_\text{64 B}
   \end{align*} \)
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
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)
  - Little-endian
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)**

```
0x00 0x01 0x02 0x03
0x00 00 01 29 F3
0x04 0x08 0x0C 0x10 0x14 0x18 0x1C 0x20 0x24
```

little-endian
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;`
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

- `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”
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 = &y + 3; // expect 0x1b`
  - Get address of `y`, “add 3”, store in `z`

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

Pointer arithmetic

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

X

Y

Z

get this instead

0x00 0x01 0x02 0x03
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int) = 4`

```
0x18
```

- `int* z = &y + 3;`
  - Get address of `y`, add `3* sizeof(int)`
  - `&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]; // &a is 0x10`

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 are 4 bytes each.

```
a[0] 0x0
a[1] 0x4
a[2] 0x8
a[3] 0xC
a[4] 0x10
a[5] 0x14
```

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

```
0x0  0x1  0x2  0x3  0x4  0x5  0x6  0x7  0x8  0x9  0xA  0xB  0xC  0xD  0xE  0xF
AD   0B   00   00   0A   00   00   00   00   00   00   00   5F   01   00   00
00   08   00   00   00   00   00   00   00   00   00   00   00   00   00   00
AD   0B   00   00   10   00   00   00   00   00   00   00   00   00   00   00
```
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;`  
equivalent `p = &a[0];`  
`*p = 0xA;`

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

Equivalent `p[1] = 0xB;`  
`*(p+1) = 0xB;`  
pointer arithmetic: `0x10 + 1 \rightarrow 0x14`  
`p = p + 2;`  
`0x10 + 2 \rightarrow 0x18`

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;`  
`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

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

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

store at `0x18`  
`*p = a[1] + 1 = 0xC;`

(array indexing = address arithmetic
(equivalent)

`p = p + 2;`  
`18 00 00 00 00 00 00 00`  
`0x0B`  
`AD 0B 00 00`
Question: Variable values after Line 3 executes are shown on the right. What are they after Line 4&5?


```c
void main() {
    int a[] = {5,10};
    int* p = a;
    p = p + 1;    // sizeof(int) = 4
    *p = *p + 1;
}
```

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Data (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>a[0] 5</td>
</tr>
<tr>
<td></td>
<td>a[1] 10</td>
</tr>
</tbody>
</table>

After Line 4:

- (A) 101 10 5 10
- (B) 104 10 5 10
- (C) 100 6 6 10
- (D) 100 6 6 10

Then After Line 5:

- (A) 101 11 5 11
- (B) 104 11 5 11
- (C) 101 6 6 10
- (D) 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>Decimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>49</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>50</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>51</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>52</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>53</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>54</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>55</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>56</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>57</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>58</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>59</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>60</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>61</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>62</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>63</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>49</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>50</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>51</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>52</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>53</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>54</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>55</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>56</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>57</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>58</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>59</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>60</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>61</td>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>62</td>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>63</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>64</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>65</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>66</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>67</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>68</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>69</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>70</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>71</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>72</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>73</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>74</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>75</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>76</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>77</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>78</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>79</td>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>80</td>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>81</td>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>82</td>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>83</td>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>84</td>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>85</td>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>86</td>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>87</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>88</td>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>89</td>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>90</td>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>91</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>92</td>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>93</td>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>94</td>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>95</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>96</td>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>97</td>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>98</td>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>99</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>100</td>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>101</td>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>102</td>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>103</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>104</td>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>105</td>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>106</td>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>107</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>108</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>110</td>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>111</td>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>

*ASCII: American Standard Code for Information Interchange*
Null-Terminated Strings

- **Example:** “Donald Trump” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>68</th>
<th>111</th>
<th>110</th>
<th>97</th>
<th>108</th>
<th>100</th>
<th>32</th>
<th>84</th>
<th>114</th>
<th>117</th>
<th>109</th>
<th>112</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>0x44</td>
<td>0x6F</td>
<td>0x6E</td>
<td>0x61</td>
<td>0x6C</td>
<td>0x64</td>
<td>0x20</td>
<td>0x54</td>
<td>0x72</td>
<td>0x75</td>
<td>0x6D</td>
<td>0x70</td>
<td>0x00</td>
</tr>
<tr>
<td>Text</td>
<td>D o n a l d T r u m p \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

13 bytes total!
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

C (char = 1 byte)

<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 00</td>
</tr>
<tr>
<td>0x01 32</td>
<td>0x01 00</td>
</tr>
<tr>
<td>0x02 33</td>
<td>0x02 00</td>
</tr>
<tr>
<td>0x03 34</td>
<td>0x03 00</td>
</tr>
<tr>
<td>0x04 35</td>
<td>0x04 00</td>
</tr>
<tr>
<td>0x05 00</td>
<td>0x05 00</td>
</tr>
</tbody>
</table>

0x31 = 49 decimal = ASCII '1'

String literal
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 \t 0x%.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 \t 0x%.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

```c
int x = 12345;
0x7fffffff71dbc 0x39
0x7fffffff71dbd 0x30
0x7fffffff71dbe 0x00
0x7fffffff71dbf 0x00
```
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**
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic (True → 1, False → 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: $A \wedge B = 1$ 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: $\sim (A \mid B) = \sim A \& \sim B$
  - $\sim (A \& B) = \sim A \mid \sim B$

<table>
<thead>
<tr>
<th>A</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
  - All of the properties of Boolean algebra apply

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 \\
& 01010101 & 01010101 & ^ 01010101 & \sim 01010101 \\
01000001 & \quad 01111101 & \quad 00111100 & \quad 10101010
\end{align*}
\]

- Examples of useful operations:

\[
\begin{align*}
x \land x & = 0 \\
\text{“sets to 1”} & \\
x | 1 & = 1, \\
\text{“leaves as is”} & \\
0 | 1 & = 1 \\
1 | 1 & = 1 \\
0 | 0 & = 0 \\
1 | 0 & = 1
\end{align*}
\]
Bit-Level Operations in C

- & (AND), | (OR), ^ (XOR), ~ (NOT)
  - View arguments as bit vectors, apply operations bitwise
  - Apply to any “integral” data type: long, int, short, char, unsigned

Examples with char a, b, c:

<table>
<thead>
<tr>
<th>C code</th>
<th>Internally</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = (char) 0x41;</td>
<td>0x41-&gt;0b 0100 0001</td>
<td></td>
</tr>
<tr>
<td>b = ~a;</td>
<td></td>
<td>0111 1110-&gt;0x BE</td>
</tr>
<tr>
<td>a = (char) 0x69;</td>
<td>0x69-&gt;0b 0110 1001</td>
<td></td>
</tr>
<tr>
<td>b = (char) 0x55;</td>
<td>0x55-&gt;0b 0101 0101</td>
<td></td>
</tr>
<tr>
<td>c = a &amp; b;</td>
<td>0b 0100 0001-&gt;0x41</td>
<td></td>
</tr>
<tr>
<td>a = (char) 0x41;</td>
<td>0x41-&gt;0b 0100 0001</td>
<td></td>
</tr>
<tr>
<td>b = a;</td>
<td>0b 0100 0001</td>
<td></td>
</tr>
<tr>
<td>c = a ^ b;</td>
<td>0b 0000 0000-&gt;0x00</td>
<td></td>
</tr>
</tbody>
</table>
Contrast: Logic Operations

- Logical operators in C: \&\& (AND), || (OR), ! (NOT)
  - 0 is False, anything nonzero is True
  - Always return 0 or 1
  - Early termination (a.k.a. short-circuit evaluation) of &&, ||

Examples (char data type)

- !0x41 \to 0x00
- !0x00 \to 0x01
- !!0x41 \to 0x01
- p && *p++
  - Avoids null pointer (0x0) access via early termination
  - Short for: if (p) { *p++; }

0xCC \&\& 0x33 \to 0x00
0x00 || 0x33 \to 0x01
Summary

- Assignment in C puts a value in a memory location
- 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)
- Bit-level operators allow for fine-grained manipulations of data
  - Bitwise AND (\&), OR (\|), and NOT (\~) different than logical AND (\&\&), OR (\|\|), and NOT (!)
  - Especially useful with bit masks