Hardware: Logical View

- CPU
- Memory
- Bus
- Disks
- Net
- USB
- Etc.
Hardware: Physical View

- Bus connections
- USB...
- I/O controller
- Storage connections
- CPU
- Memory

Winter 2016
CPU executes instructions; memory stores data

To execute an instruction, the CPU must:

- fetch an instruction;
- fetch the data used by the instruction; and, finally,
- execute the instruction on the data...
- which may result in writing data back to memory.
The CPU holds instructions temporarily in the **instruction cache**.

The CPU holds data temporarily in a fixed number of **registers**.

**Instruction and operand fetching** is HW-controlled.

**Data movement** is (assembly language) programmer-controlled.

We’ll learn about the instructions the CPU executes – take cse/ee470 to find out how it actually executes them.
The CPU holds instructions temporarily in the instruction cache. 
The CPU holds data temporarily in a fixed number of registers. 
Instruction fetching is HW-controlled. 
Data movement is programmer-controlled. 

How are data and instructions represented? 

How does a program find its data in memory?
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret

Machine code:
011101010000011000
100011010000010000000010
1000100111000010
1100000111111101000011111

OS:
Windows 8
Mac

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
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
How are data and instructions represented?
Binary Representations

- **Base 2 number representation**
  - A base 2 digit (0 or 1) is called a *bit*.
  - Represent $351_{10}$ as $0000001010111111_2$ or $101011111_2$

- **Electronic implementation**
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires
Describing Byte Values

- **Binary**  
  00000000₂ -- 11111111₂  
  - Byte = 8 bits (binary digits)

- **Decimal**  
  0₁₀ -- 255₁₀

- **Hexadecimal**  
  00₁₆ -- FF₁₆  
  - Byte = 2 hexadecimal (or “hex” or base 16) digits
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B₁₆ in the C language  
    - as 0xFA1D37B or 0xfa1d37b

- More on specific data types later...
How does a program find its data in memory?
Byte-Oriented Memory Organization

- Conceptually, memory is a single, large array of bytes, each with an unique *address* (index)
- The value of each byte in memory can be read and written
- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
- But not all values (*e.g.*, 351) fit in a single byte...
  - Store addresses to “remember” where other data is in memory
  - How much memory can we address with 1-byte (8-bit) addresses?
- Many operations actually use multi-byte values
Machine Words

- **Word size = address size = register size**

- **Word size bounds the size of the address space and memory**
  - word size = $w$ bits $\Rightarrow$ $2^w$ addresses
  - Until recently, most machines used **32-bit (4-byte) words**
    - Potential address space: $2^{32}$ addresses
      - $2^{32}$ bytes $\approx 4 \times 10^9$ bytes = 4 billion bytes = **4GB**
      - Became too small for memory-intensive applications
  - Current x86 systems use **64-bit (8-byte) words**
    - Potential address space: $2^{64}$ addresses
      - $2^{64}$ bytes $\approx 1.8 \times 10^{19}$ bytes = 18 billion billion bytes = **18 EB** (exabytes)
### Word-Oriented Memory Organization

- **Addresses specify locations of bytes in memory**
  - Address of word = address of first byte in word
  - Addresses of successive words differ by word size (in bytes): e.g., 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, .. 10?

<table>
<thead>
<tr>
<th>64-bit Words</th>
<th>32-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0015</td>
</tr>
</tbody>
</table>

(note: decimal addresses)
Word-Oriented Memory Organization

Addresses still specify locations of *bytes* in memory

- Address of word = address of first byte in word
- Addresses of successive words differ by word size (in bytes): e.g., 4 (32-bit) or 8 (64-bit)
- Address of word 0, 1, .. 10?

**Alignment**
A Picture of Memory (32-bit view)

- A “32-bit (4-byte) word-aligned” view of memory:
  - In this type of picture, each row is composed of 4 bytes
  - Each cell is a byte
  - A 32-bit pointer will fit on one row

```
0x00 0x01 0x02 0x03
0x04 0x05 0x06 0x07

0x00 0x04 0x08 0x0C
0x10 0x14 0x18 0x1C
0x20 0x24
```

(note hex addresses)
A Picture of Memory (64-bit view)

- A “64-bit (8-byte) word-aligned” view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row
A Picture of Memory (64-bit view)

A “64-bit (8-byte) word-aligned” view of memory:

- In this type of picture, each row is composed of 8 bytes
- Each cell is a byte
- A 64-bit pointer will fit on one row
Addresses and Pointers

- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address **0x04**
  - $351_{10} = 15F_{16} = 0x00 00 01 5F$

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

```
00 00 01 5F
0x00
0x04
0x08
0x0C
0x10
0x14
0x18
0x1C
0x20
0x24
```
Addresses and Pointers

- An **address** is a location in memory
- A **pointer** is a data object that holds an address
- The value 351 is stored at address **0x04**
  - \(351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F\)
- A pointer stored at address **0x1C** points to address **0x04**
**Addresses and Pointers**

- **An address** is a location in memory.
- **A pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**
  - \( 351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F \)
- A pointer stored at address **0x1C** points to address **0x04**
- A pointer to a pointer is stored at address **0x24**

![32-bit example (pointers are 32-bits wide)](image)
Addresses and Pointers

- An **address** is a location in memory.
- A **pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**
  - \(351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F\)
- A pointer stored at address **0x1C** points to address **0x04**
- A pointer to a pointer is stored at address **0x24**
- The value 12 is stored at address **0x14**
  - Is it a pointer?
Addresses and Pointers

- A 64-bit (8-byte) word-aligned view of memory

- The value 351 is stored at address **0x08**
  - $351_{10} = 15F_{16} = 0x00 00 01 5F$

- A pointer stored at address **0x38**
  - points to address **0x08**

- A pointer to a pointer is stored at address **0x48**

\[\begin{array}{cccccccc}
00 & 00 & 00 & 00 & 00 & 00 & 01 & 5F \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 08 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 38 \\
\end{array}\]

(Note hex addresses)

64-bit example (pointers are 64-bits wide)
## Data Representations

### Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>(reference)</td>
<td>pointer *</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

To use “bool” in C, you must include `<stdbool.h>`

**address size = word size**
More on Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data.
- Aligned means: Any primitive object of K bytes must have an address that is a multiple of K.
- This means we could expect these types to have starting addresses that are the following multiples:

<table>
<thead>
<tr>
<th>K</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, pointers</td>
</tr>
</tbody>
</table>

More about alignment later in the course
Byte Ordering

- How should bytes within a word be ordered in memory?

Example:
- Store the 4-byte (32-bit) word: 0xa1 b2 c3 d4
  - In what order will the bytes be stored?

Conventions!
- Big-endian, Little-endian
- Based on Gulliver’s Travels: tribes cut eggs on different sides (big, little)
Byte Ordering

- **Big-Endian** (PowerPC, SPARC, The Internet)
  - Least significant byte has highest address

- **Little-Endian** (x86)
  - Least significant byte has lowest address

**Example**
- Variable has 4-byte representation `0xa1b2c3d4`
- Address of variable is `0x100`

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x0100</td>
</tr>
<tr>
<td>0x101</td>
<td>0x0101</td>
</tr>
<tr>
<td>0x102</td>
<td>0x0102</td>
</tr>
<tr>
<td>0x103</td>
<td>0x0103</td>
</tr>
</tbody>
</table>

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

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

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

```c
long int y = 12345;
// or y = 0x3039;
```

(A `long int` is the size of a word)
Reading Byte-Reversed Listings

- **Disassembly**
  - Take binary machine code and generate an assembly code version
  - Does the reverse of the assembler

- **Example instruction in memory**
  - add value 0x12ab to register ‘ebx’ (*a special location in the CPU*)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>
Reading Byte-Reversed Listings

- **Disassembly**
  - Take binary machine code and generate an assembly code version
  - Does the reverse of the assembler

- **Example instruction in memory**
  - add value 0x12ab to register ‘ebx’ (*a special location in the CPU*)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>

Deciphering numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse (little-endian): ab 12 00 00
Addresses and Pointers in C

int* ptr;

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

int x = 5;
int y = 2;

Declares two variables, `x` and `y`, that hold ints, and sets them to 5 and 2, respectively.

ptr = &x;

Sets `ptr` to the address of `x`. Now, “`ptr` points to `x`”

“Dereference `ptr`”

What is `*(&y)`?

y = 1 + *ptr;

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`;”
Assignment in C

- A variable is represented by a memory location
- Initially, it may hold any value
- `int x, y;`
  - x is at location 0x04, y is at 0x18

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

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

* is also used with variable declarations

<table>
<thead>
<tr>
<th>Address (0x00-0x03)</th>
<th>Hex Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7 00 32 00</td>
<td>x</td>
</tr>
<tr>
<td>0x04</td>
<td>00 01 29 F3</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>EE EE EE EE</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td>FA CE CA FE</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>26 00 00 00</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td>00 00 10 00</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>01 00 00 00</td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td>FF 00 F4 96</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>00 00 00 00</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td>00 42 17 34</td>
<td></td>
</tr>
</tbody>
</table>

Winter 2016
Memory & data
Assignment in C

- A variable is represented by a memory location
- Initially, it may hold any value
- int x, y;
  - x is at location 0x04, y is at 0x18

`& = 'address of'
* = 'value at address' or '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;

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

& = 'address of'
* = 'value at address' or 'dereference'

\begin{align*}
0x00 & \quad 0x01 & \quad 0x02 & \quad 0x03 & \\
00 & 00 & 00 & 00 & 0x00 \\
00 & 00 & 00 & 00 & 0x04 \\
00 & 00 & 00 & 00 & \cdots \\
01 & 00 & 00 & 00 & 0x18 \\
00 & 00 & 00 & 00 & 0x1C \\
00 & 00 & 00 & 00 & 0x20 \\
00 & 00 & 00 & 00 & 0x24 \\
\end{align*}
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;
```

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

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

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;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, put it in x

\[
\begin{array}{cccc}
0x00 & 0x01 & 0x02 & 0x03 \\
03 & 27 & D0 & 3C \\
0x00 & 0x04 & x \\
0x08 & \\
0x0C & \\
0x10 & \\
0x14 & \\
0x18 & y \\
0x1C & \\
0x20 & \\
0x24 & \\
\end{array}
\]
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, put it in x
- int* z

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

& = ‘address of’
* = ‘value at address’ or ‘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, put it in x
- `int* z = &y + 3;`
  - Get address of y, add ???, put it in z

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

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

& = ‘address of’
* = ‘value at address’ or ‘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, put it in x

- **int* z = &y + 3;**
  - Get address of y, add 12, put it in z

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

<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>0x00</td>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
</tr>
<tr>
<td>y</td>
<td>0x18</td>
<td>0x1C</td>
<td>0x20</td>
<td>0x24</td>
</tr>
</tbody>
</table>

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

- **0x18 = 24 (decimal) + 12 = 0x24**

- Pointer arithmetic is scaled by size of target type

- Pointer arithmetic can be dangerous
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, put it in x
- int* z = &y + 3;
  - Get address of y, add 12, put it in z
- *z = y;
  - What does this do?
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;

x = 0;
y = 0x3CD02700;
x = y + 3;
*z = y;
```

The target of a pointer is also a memory location

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

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

<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>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>0X00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X0C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X1C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0X24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x
y
z
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[0]`: 0x00
- `a[1]`: 0x08
- `a[2]`: 0x10
- `a[3]`: 0x18
- `a[4]`: 0x20
- `a[5]`: 0x28

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

- `a`: 0x00 - 0x07
- `a[1]`: 0x08 - 0x0F
- `a[3]`: 0x10 - 0x17
- `a[5]`: 0x18 - 0x1F

- `a[0]`: 0x20 - 0x27
- `a[2]`: 0x28 - 0x2F
- `a[4]`: 0x30 - 0x37
- `a[6]`: 0x38 - 0x3F

- `a[1]`: 0x40 - 0x47
- `a[3]`: 0x48 - 0x4F
- `a[5]`: 0x50 - 0x57
- `a[7]`: 0x58 - 0x5F

- `a[2]`: 0x60 - 0x67
- `a[4]`: 0x68 - 0x6F
- `a[6]`: 0x70 - 0x77
- `a[8]`: 0x78 - 0x7F

- `a[3]`: 0x80 - 0x87
- `a[5]`: 0x88 - 0x8F
- `a[7]`: 0x90 - 0x97
- `a[9]`: 0x98 - 0x9F

- `a[4]`: 0xA0 - 0xA7
- `a[6]`: 0xA8 - 0xAF
- `a[8]`: 0xB0 - 0xB7
- `a[10]`: 0xB8 - 0xBF

- `a[5]`: 0xC0 - 0xC7
- `a[7]`: 0xC8 - 0xCF
- `a[9]`: 0xD0 - 0xD7
- `a[11]`: 0xD8 - 0xDF

- `a[6]`: 0xE0 - 0xE7
- `a[8]`: 0xE8 - 0xEF
- `a[10]`: 0xF0 - 0xF7
- `a[12]`: 0xF8 - 0xFF
### Arrays in C

**Declaration:**
```c
int a[6];
```

**Indexing:**
```c
a[0] = 0x015f;
a[5] = a[0];
```

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

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

---

<table>
<thead>
<tr>
<th>i</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
<td>5F 01 00 00</td>
</tr>
<tr>
<td>1</td>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x18</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0x28</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0x30</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x40</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0x48</td>
<td></td>
</tr>
</tbody>
</table>
# Arrays in C

**Declaration:**
```c
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.

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

## Memory Map
```
<table>
<thead>
<tr>
<th>0x00</th>
<th>0x08</th>
<th>0x10</th>
<th>0x18</th>
<th>0x20</th>
<th>0x28</th>
<th>0x30</th>
<th>0x38</th>
<th>0x40</th>
<th>0x48</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Winter 2016
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];

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 check: a[6] = 0xBAD;

Pointers: int* p;

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

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

---

Memory & data
Arrays in C

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

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

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

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

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

Equivalent:
\( \text{int* } p; \)
\( p = a; \)
\( p = \&a[0]; \)
\( *p = 0xA; \)
Arrays in C

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

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

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

Pointers: \texttt{int* p;}

\texttt{p = a; p = \&a[0]; *p = 0xA; p[1] = 0xB;}

Arrays are adjacent locations in memory storing the same type of data object
\texttt{a} is a name for the array’s address

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

<table>
<thead>
<tr>
<th>Address (in hex)</th>
<th>0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a[0]}</td>
<td>\texttt{0A 00 00 00 00}</td>
</tr>
<tr>
<td>\texttt{a[2]}</td>
<td>\texttt{5F 01 00 00}</td>
</tr>
<tr>
<td>\texttt{a[4]}</td>
<td>\texttt{10 00 00 00 00 00 00 00}</td>
</tr>
<tr>
<td>\texttt{p[1]}</td>
<td>\texttt{AD 0B 00 00}</td>
</tr>
</tbody>
</table>

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

Pointers:
\texttt{int* p; p = a; p = \&a[0]; *p = 0xA; p[1] = 0xB;}

\texttt{p} is an equivalent name for the array’s address.
Arrays in C

Declaration: int a[6];

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

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

check: a[-1] = 0xBAD;

Pointers: int* p;

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

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

No bounds check:

p[6] = 0xBAD;
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: 
- \( \text{int* } p; \)
- \( p = a; \)
- \( p = &a[0]; \)
- \( *p = 0xA; \)
- \( p[1] = 0xB; \)

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

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

---

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

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

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

Pointers:
- \( \text{int* } p; \)
- \( p = a; \)
- \( p = &a[0]; \)
- \( *p = 0xA; \)
- \( p[1] = 0xB; \)

Array indexing = address arithmetic
Both are scaled by the size of the type
### 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

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

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

Pointers:
```
{equivalent}
int* p;
p = a;
p = &a[0];
*p = 0xA;
```

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0xA</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x08</td>
<td>0xB</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x10</td>
<td>0x0A</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x18</td>
<td>0x0B</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x20</td>
<td>0x5F</td>
<td>0x01</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x28</td>
<td></td>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arrays in C

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

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

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

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

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

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

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

Pointers: \(\text{int* } p;\)

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

\(a[0] = 0x15f;\)
\(a[5] = a[0];\)

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

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

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>0A 00 00 00</td>
</tr>
<tr>
<td>0x08</td>
<td>AD 0B 00 00</td>
</tr>
<tr>
<td>0x10</td>
<td>0B 00 00 00</td>
</tr>
<tr>
<td>0x18</td>
<td>AD 0B 00 00</td>
</tr>
<tr>
<td>0x20</td>
<td>5F 01 00 00</td>
</tr>
<tr>
<td>0x28</td>
<td>0A 00 00 00</td>
</tr>
<tr>
<td>0x30</td>
<td>0B 00 00 00</td>
</tr>
<tr>
<td>0x38</td>
<td>0A 00 00 00</td>
</tr>
<tr>
<td>0x40</td>
<td>5F 01 00 00</td>
</tr>
<tr>
<td>0x48</td>
<td>18 00 00 00</td>
</tr>
</tbody>
</table>

---

Winter 2016
Arrays in C

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

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

No bounds check: \( a[6] = 0x\text{BAD}; \)  
\( a[-1] = 0x\text{BAD}; \)

Pointers: \( \text{int } *p; \)
\( p = a; \)  
\( p = \&a[0]; \)  
\( *p = 0xA; \)
\( p[1] = 0xB; \)  
\( *(p + 1) = 0xB; \)
\( p = p + 2; \)

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

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

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

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

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

Pointers:
\[ \text{int } * p; \]
\[ p = a; \]
\[ p = &a[0]; \]
\[ *p = 0x1A; \]

Equivalent:
\[ p[1] = 0xB; \]
\[ *(p + 1) = 0xB; \]
\[ p = p + 2; \]

Arrays are adjacent locations in memory storing the same type of data object
\[ a \text{ is a name for the array’s address} \]

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

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

Array indexing = address arithmetic
Both are scaled by the size of the type

Memory & data
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>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>&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>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>
<tr>
<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";`

- **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");
}
```

void show_int (int x) {
    show_bytes( (char *) &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((char *) &a, sizeof(int));
```

Result (Linux x86-64):

```c
int a = 12345;
0x7fffbb7f71dbc 0x39
0x7fffbb7f71dbd 0x30
0x7fffbb7f71dbe 0x00
0x7fffbb7f71dbf 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: $A | B = 1$ 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: $\neg A = 1$ when $A$ is 0 and vice-versa
  - DeMorgan’s Law: $\neg (A | B) = \neg A \& \neg B$
    $\neg (A \& B) = \neg A | \neg B$

<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>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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|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: ~A = 1 when A is 0 and vice-versa
  - DeMorgan’s Law: ~(A | B) = ~A & ~B
    ~A | ~B = ~(A & B) = ~A | ~B

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

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

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

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

- **Operate on bit vectors**
  - Operations applied bitwise

  \[
  \begin{array}{c}
  01101001 \\
  \& 01010101 \\
  01000001
  \\
  \hline
  01101001 \\
  | 01010101 \\
  01111101
  \\
  \hline
  01101001 \\
  ^ 01010101 \\
  00111100
  \\
  \hline
  \sim 01010101 \\
  10101010
  \end{array}
  \]

- **All of the properties of Boolean algebra apply**

  \[
  \begin{array}{c}
  01010101 \\
  ^ 01010101 \\
  00000000
  \end{array}
  \]

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

## Representation
- A \( w \)-bit vector represents subsets of \( \{0, ..., w-1\} \)
- \( a_j = 1 \) iff \( j \in A \)

\[
\begin{align*}
01101001 & \quad \{0, 3, 5, 6\} \\
01010101 & \quad \{0, 2, 4, 6\}
\end{align*}
\]

## Operations
- \& Intersection
- | Union
- ^ Symmetric difference
- ~ Complement

\[
\begin{align*}
010\ldots0101 & \quad \{0, 6\} \\
011\ldots1101 & \quad \{0, 2, 3, 4, 5, 6\} \\
001\ldots1100 & \quad \{2, 3, 4, 5\} \\
101\ldots0100 & \quad \{1, 3, 5, 7\}
\end{align*}
\]
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 --&gt; 0xBE
    - ~01000001₂ --&gt; 10111110₂
  - ~0x00 --&gt; 0xFF
    - ~00000000₂ --&gt; 11111111₂
  - 0x69 & 0x55 --&gt; 0x41
    - 01101001₂ & 01010101₂ --&gt; 01000001₂
  - 0x69 | 0x55 --&gt; 0x7D
    - 01101001₂ | 01010101₂ --&gt; 01111101₂

- Some 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`
  - `0x68 && 0x55`  -->  `0x01`
  - `0x68 || 0x55`  -->  `0x01`
  - `p && *p++` (avoids null pointer access, null pointer = 0x0000 0000 0000 0000 0000)