Computer Systems
CSE 410 Autumn 2013
2 – Memory and its Data
**Roadmap**

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

- **Java:**
  ```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:**
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111110100001111
```

**OS:**
- Windows 8
- Mac

**Computer system:**
- CPU
- RAM
- Hard drive

**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

- Preliminaries
- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations

- Reading: Bryant/O’Hallaron sec. 2.1
Hardware: Logical View

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

Intel® P45 Express Chipset Block Diagram

- Intel® Core™2 Duo Processor
- Intel® Core™2 Quad Processor
- P45 MCH
- DDR2 or DDR3 6.4 GB/s or 8.5 GB/s
- DDR2 or DDR3 6.4 GB/s or 8.5 GB/s
- 2 GB/s DMI
- Intel® High Definition Audio
- Intel® Quiet System Technology
- 6 Serial ATA Ports; eSATA; Port Disable
- Intel® Matrix Storage Technology
- Intel® Turbo Memory with User Pinning
- Intel® Gigabit LAN Connect
- Intel® Integrated 10/100/1000 MAC
- GLCi, LCI
- BIOS Support
- Intel® Extreme Tuning Support
- 12 Hi-Speed USB 2.0 Ports; Dual EHCI; USB Port Disable
- 6 PCI Express® x1
- 16 lanes 16 GB/s
- 8 lanes 8 GB/s
- 8 lanes 8 GB/s
- 480 Mb/s each
- 500 Mb/s each x1
- 3 Gb/s each
- 10.6 GB/s
Hardware: Physical View
There are a fixed number of registers in the CPU
  - Registers hold data

There is an I-cache in the CPU that holds recently fetched instructions
  - If you execute a loop that fits in the cache, the CPU goes to memory for those instructions only once, then executes it out of its cache

This slide is just an introduction.
We'll see a fuller explanation later in the course.
Performance: It's Not Just CPU Speed

- **Data and instructions reside in memory**
  - To execute an instruction, it must be fetched into the CPU
  - Next, the data on the which the instruction operates must be fetched from memory and brought to the CPU

- **CPU <-> Memory bandwidth can limit performance**
  - Improving performance 1: hardware improvements to increase memory bandwidth (e.g., DDR → DDR2 → DDR3)
  - Improving performance 2: move less data into/out of the CPU
  - Put some “memory” in the CPU chip itself (this is “cache” memory)
Binary Representations

- **Base 2 number representation**
  - Represent $351_{10}$ as $000000101011111_2$ or $101011111_2$

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

![Binary representation diagram](image-url)
Memory, Data, and Addressing

- Preliminaries
- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
Encoding Byte Values

- **Binary**
  - 00000000₂ -- 11111111₂
    - Byte = 8 bits (binary digits)
    - Example: 00101011₂ = 32+8+2+1 = 43₁₀
    - Example: 26₁₀ = 16+8+2 = 00101010₂

- **Decimal**
  - 0₁₀ -- 255₁₀

- **Hexadecimal**
  - 00₁₆ -- FF₁₆
    - Groups of 4 binary digits
    - Byte = 2 hexadecimal (hex) or base 16 digits
    - Base-16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C
      - as 0xFA1D37B or 0xfalld37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
What is memory, really?

- How do we find data in memory?
Byte-Oriented Memory Organization

Programs refer to addresses
- Conceptually, a very large array of bytes, each with an address (index)
- System provides an address space private to each “process”
  - Process = program being executed + its data + its “state”
  - Program can clobber its own data, but not that of others
  - Clobbering code or “state” often leads to crashes (or security holes)

Compiler + run-time system control memory allocation
- Where different program objects should be stored
- All allocation within a single address space
Machine Words

- Machine has a “word size”
  - Nominal size of integer-valued data
    - Including addresses
  - Until recently, most machines used 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Became too small for memory-intensive applications
  - More recent and high-end systems use 64 bits (8 bytes) words
    - Potential address space $\approx 1.8 \times 10^{19}$ bytes (18 EB – exabytes)
    - x86-64 supports 48-bit physical addresses: 256 TB (terabytes)
  - For backward-compatibility, many CPUs support different word sizes
    - Always a power-of-2 in the number of bytes: 1, 2, 4, 8, ...
Word-Oriented Memory Organization

- Addresses specify locations of bytes in memory
  - Address of first byte in word
  - Addresses of successive words differ by 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>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0015</td>
<td></td>
</tr>
</tbody>
</table>
Word-Oriented Memory Organization

- Addresses specify locations of bytes in memory
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, .. 10?
Memory, Data, and Addressing

- Preliminaries
- 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

- **Address** is a *location* in memory
- **Pointer** is a data object that *contains an address*
- **Address 0004** stores the value 351 (or $15F_{16}$)
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object that *contains an address*
- Address 0004 stores the value 351 (or $15F_{16}$)
- Pointer to address 0004 stored at address 001C
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object that *contains an address*

- Address 0004
  stores the value 351 (or $15F_{16}$)

- Pointer to address 0004
  stored at address 001C

- Pointer to a pointer
  in 0024

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0004</td>
<td>0008</td>
</tr>
<tr>
<td>0008</td>
<td>000C</td>
</tr>
<tr>
<td>0010</td>
<td>0014</td>
</tr>
<tr>
<td>0014</td>
<td>0018</td>
</tr>
<tr>
<td>0018</td>
<td>0020</td>
</tr>
<tr>
<td>0020</td>
<td>0024</td>
</tr>
</tbody>
</table>
Addresses and Pointers

- Address is a *location* in memory
- Pointer is a data object that *contains an address*
- Address 0004 stores the value 351 (or $15F_{16}$)
- Pointer to address 0004 stored at address 001C
- Pointer to a pointer in 0024
- Address 0014 stores the value 12
  - Is it a pointer?
# Data Representations

- **Sizes of objects (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>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</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></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>
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Say you want to store 0xaabbccdd
  - What order will the bytes be stored?
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Say you want to store 0xaabbccdd
  - What order will the bytes be stored?

**Endianness: big endian vs. little endian**
- Two different conventions, used by different architectures
- Origin: *Gulliver’s Travels* (see CS:APP2 textbook, section 2.1)
Byte Ordering Example

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

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

**Example**
- Variable has 4-byte representation `0x01234567`
- Address of variable is `0x100`

<table>
<thead>
<tr>
<th>Address</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Big Endian</strong></td>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
</tr>
<tr>
<td><strong>Little Endian</strong></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Representing Integers

- int A = 12345;
- int B = -12345;
- long int C = 12345;

Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

Two’s complement representation for negative integers (covered later)
Memory, Data, and Addressing

- Preliminaries
- 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

- **Variable declarations**
  - int x, y;
  - Finds two locations in memory in which to store 2 integers (1 word each)

- **Pointer declarations use ***
  - int *ptr;
  - Declares a variable `ptr` that is a pointer to a data item that is an integer

- **Assignment to a pointer**
  - ptr = &x;
  - Assigns `ptr` to point to the address where `x` is stored
    - (stores the address of `x` in `ptr`)
Addresses and Pointers in C

- To use the value pointed to by a pointer we use dereference (*):
  - Given a pointer, we can get the value it points to by using the * operator.
  - *ptr is the value at the memory address given by the value of ptr.

- Examples:
  - If ptr = &x then y = *ptr + 1 is the same as y = x + 1.
  - If ptr = &y then y = *ptr + 1 is the same as y = y + 1.
  - What is *(&x) equivalent to?
Addresses and Pointers in C

- We can do arithmetic on pointers
  - `ptr = ptr + 1;`  // really adds 4: type of `ptr` is `int*`, and an `int` uses 4 bytes!
  - Changes the value of the pointer so that it now points to the next data item in memory (that may be y, or it may not – this is dangerous!)

\& = ‘address of value’
* = ‘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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700

```
00 00 00 00 00 04 00 00 00 00 00 08 00 00 C0 00 01 00 00 14 00 27 00 3C 00 18 00 1C 00 20 00 24
```
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int x, y;
  - x = y + 3; //get value at y, add 3, put it in x

```
Memory
```

```
00 00 00 00 0004
00 00 00 00 0008
00 00 00 00 000C
00 00 00 00 0010
00 00 00 00 0014
00 27 D0 3C 0018
00 00 00 00 0020
00 00 00 00 0024
```
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int x, y;
  - \(x = y + 3;\) //get value at y, add 3, put it in x

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>0x0000</th>
<th>0x0004</th>
<th>0x0008</th>
<th>0x0010</th>
<th>0x0014</th>
<th>0x0018</th>
<th>0x0020</th>
<th>0x0024</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>03 27  D0 3C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
<td>0010</td>
<td>0014</td>
<td>0018</td>
</tr>
<tr>
<td>0008</td>
<td></td>
<td></td>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
<td>0010</td>
<td>0014</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
<td>0010</td>
</tr>
<tr>
<td>0014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
</tr>
<tr>
<td>0018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0004</td>
<td>0008</td>
</tr>
<tr>
<td>0020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
<td>0004</td>
</tr>
<tr>
<td>0024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

Memory
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int * x; int y;
  - x = &y + 3; // get address of y add ??

```c
int * x;
int y;

x = &y + 3; // get address of y add ??
```
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int * x; int y;
  - x = &y + 3; // get address of y add 12
    // 0x0018 + 0x000C = 0x0024

```
int * x;
int y;

x = &y + 3; // get address of y add 12
// 0x0018 + 0x000C = 0x0024
```

```plaintext
Memory

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>04</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>08</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0C</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>14</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

```
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int * x; int y;
    x = &y + 3; // get address of y add 12
    // 0x0018 + 0x000C = 0x0024
  - *x = y; // value of y copied to
    // location to which x points

```
24  00  00  00  0000
24  00  00  00  0004
24  00  00  00  0008
24  00  00  00  000C
24  00  00  00  0010
24  00  00  00  0014
24  00  00  00  0018
24  00  00  00  0020
24  00  00  00  0024
```

```
00  27  D0  3C  0000
00  27  D0  3C  0004
00  27  D0  3C  0008
00  27  D0  3C  000C
00  27  D0  3C  0010
00  27  D0  3C  0014
00  27  D0  3C  0018
00  27  D0  3C  0020
00  27  D0  3C  0024
```
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)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x3CD02700
  - int * x; int y;
    - x = &y + 3; // get address of y add 12
      // 0x0018 + 0x000C = 0x0024
  - *x = y; // value of y copied to
    // location to which x points

```c
int * x;
int y;
x = &y + 3; // get address of y add 12
// 0x0018 + 0x000C = 0x0024
*x = y; // value of y copied to
      // location to which x points
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
<td>0010</td>
</tr>
<tr>
<td>0014</td>
<td>0018</td>
<td>001C</td>
<td>0020</td>
<td>0024</td>
</tr>
</tbody>
</table>

Memory
Arrays

- Arrays represent adjacent locations in memory storing the same type of data object
  - e.g., int big_array[128];
    allocates 512 adjacent bytes in memory starting at 0x00ff0000

- Pointer arithmetic can be used for array indexing in C (if pointer and array have the same type!):
  - int *array_ptr;
    array_ptr = big_array; 0x00ff0000
    array_ptr = &big_array[0]; 0x00ff0000
    array_ptr = &big_array[3]; 0x00ff000c
    array_ptr = &big_array[0] + 3; 0x00ff000c (adds 3 * size of int)
    array_ptr = big_array + 3; 0x00ff000c (adds 3 * size of int)
    *array_ptr = *array_ptr + 1; 0x00ff000c (but big_array[3] is incremented)
    array_ptr = &big_array[130]; 0x00ff0208 (out of bounds, C doesn’t check)

- In general: &big_array[i] is the same as (big_array + i),
  which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);
Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

<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, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`

- How do we compute the string length??
Compatibility

```c
char S[6] = "12345";
```

<table>
<thead>
<tr>
<th>Linux/Alpha S</th>
<th>Sun S</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

- **Byte ordering (endianness)** is not an issue for standard C strings (char arrays)
- **Unicode characters** – up to 4 bytes/character
  - ASCII codes still work (leading 0 bit) but 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
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");
}
```

```c
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((pointer) &a, sizeof(int));
```

**Result (Linux):**

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

Memory, Data, and Addressing

- Preliminaries
- 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
    - 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

<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></td>
</tr>
</tbody>
</table>
Manipulating Bits

- Boolean operators can be applied to *bit vectors*: operations are applied *bitwise*

\[
\begin{array}{cccc}
01101001 & 01101001 & 01101001 & 01101001 \\
\& 01010101 & | 01010101 & ^ 01010101 & \sim 01010101 \\
01000001 & 01111101 & 00111100 & 10101010 \\
\end{array}
\]
Bit-Level Operations in C

- **Bitwise operators** `&`, `|`, `^`, `~` are available in C
  - Apply to any “integral” data type
    - `long`, `int`, `short`, `char`
  - Arguments are treated as bit vectors
  - Operations applied bitwise

- **Examples:**
  ```c
  char a, b, c;
  a = (char)0x41; // 0x41 -> 01000001
  b = ~a; // 10111110 -> 0xBE
  a = (char)0; // 0x00 -> 00000000
  b = ~a; // 11111111 -> 0xFF
  a = (char)0x69; // 0x41 -> 01101001
  b = (char)0x55; // 0x55 -> 01010101
  c = a & b; // 01000001 -> 0x41
  ```
Contrast: Logic Operations in C

- Logical operators in C: &&, ||, !
  - Behavior:
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination (&& and ||)

- Examples (char data type)
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - 0x69 && 0x55 --> 0x01
  - 0x00 && 0x55 --> 0x00
  - 0x69 || 0x55 --> 0x01
  - p && *p++ (avoids null pointer access: null pointer = 0x00000000)
    short for: if (p) { *p++; }

Memory
Representing & Manipulating Sets

- **Bit vectors can be used to represent sets**
  - Width $w$ bit vector represents subsets of $\{0, ..., w-1\}$
  - $a_j = 1$ if $j \in A$ – each bit in the vector represents the absence (0) or presence (1) of an element in the set

01101001 \quad \{ 0, 3, 5, 6 \}

76543210

01010101 \quad \{ 0, 2, 4, 6 \}

76543210

- **Operations**
  - & Intersection \quad 01000001 \quad \{ 0, 6 \}
  - | Union \quad 01111101 \quad \{ 0, 2, 3, 4, 5, 6 \}
  - ^ Symmetric difference \quad 00111100 \quad \{ 2, 3, 4, 5 \}
  - ~ Complement \quad 10101010 \quad \{ 1, 3, 5, 7 \}
- Slides past this point not used
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

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

<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**
  - Text representation of binary machine code
  - Generated by program that reads the machine code

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

<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
## Representing Pointers

- `int B = -12345;`
- `int *P = &B;`

<table>
<thead>
<tr>
<th>Sun P</th>
<th>IA32 P</th>
<th>x86-64 P</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>D4</td>
<td>0C</td>
</tr>
<tr>
<td>FF</td>
<td>F8</td>
<td>89</td>
</tr>
<tr>
<td>FB</td>
<td>FF</td>
<td>EC</td>
</tr>
<tr>
<td>2C</td>
<td>BF</td>
<td>FF</td>
</tr>
</tbody>
</table>

*Different compilers & machines assign different locations to objects*