The Hardware/Software Interface
CSE351 Spring 2015
Lecture 2

Instructor:
Katelin Bailey

Teaching Assistants:
Announcements

- On the website: cs.uw.edu/351
  - Speedometer!
  - Anonymous feedback form
  - Lecture slides on the web schedule
  - Videos for optional reference.
- Make sure you are subscribed to the mailing list.
- Check the discussion boards regularly
- Enrolling in cse390a - ask your advisor (not me) for add code
- Non-majors waiting on accounts can pick up account sheets from the front desk on Friday or Monday
- Lab 0, having fun?
- Regular section meetings starting tomorrow
Hardware: Logical View
Hardware: Physical View

- Bus connections
- PCI Slots
- PCI-Express Slots
  1 PCI-E X16, 2 PCI-E X1
- Back Panel Connectors
- I/O controller
- Serial ATA Headers
- Intel ICH10 Chipset
- Socket 775
  Core2 Quad/
  Core2 Extreme
  Ready
- Intel P45 Chipset
- DDR2
  1066+MHz
  Dual Channel
  Memory Slots
- Memory
- USB...
- CPU
- Storage connections
Hardware: 351 View (version 0)
- 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 programmer-controlled.
- We’ll learn about the instructions the CPU executes. Take 352 to find out how it executes them.
- How are data and instructions represented?
- How does a program find its data in memory?
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
110000011111101000011111
```

Computer system:

- OS: Windows 8, Mac
- Intel Core i5
- RAM
- Hard drive
- CPU: x86 architecture

- Memory, data, & addressing
- 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?

• How does a program find its data in memory?
Binary Representations

• **Base 2 number representation**
  • A base 2 digit (0 or 1) is called a bit.
  • Represent \(351_{10}\) as \(0000000101011111_2\) or \(101011111_2\)

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

- **Binary**  \(00000000_2 -- 11111111_2\)
  - Byte = 8 bits (binary digits)
- **Decimal**  \(0_{10} -- 255_{10}\)
- **Hexadecimal**  \(00_{16} -- FF_{16}\)
  - Byte = 2 hexadecimal (or “hex” or base 16) digits
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B\(_{16}\) in C
    - as 0xFA1D37B or 0xfa1d37b
- **More on specific data types later…**

<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>
How are data and instructions represented?

How does a program find its data in memory?
• 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

• fixed number of contiguous bytes in memory, chosen by HW
• the largest unit of data a machine instruction can use
• word size = address size = register size
• Word size bounds the size of the address space and memory.
  • word size = w bits  =>  \(2^w\) 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
    • (living humans / addressable bytes \(\approx\) 1.8)
    • 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)
    • (possible living acquaintances / addressable bytes \(\approx\) 2.8)
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?
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?
  - Alignment
Memory Alignment

• Data of size \( n \) only stored at addresses \( a \) where \( a \mod n = 0 \)
  • Convention or rule, depending on platform.
  • \( n \) is usually a power of 2.

• A 32-bit (4-byte) word-aligned view of memory:
  • Each row is a word composed of 4 bytes.
  • Cells in a row are the word’s bytes.

More about alignment later in the course.
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**.
  - \(351_{10} = 15F_{16} = 0x00 00 01 5F\)
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**.
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?
## 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></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 int</td>
<td></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></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>(reference)</td>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*Note: The sizes for x86-64 architectures may differ from the typical 32-bit sizes.*
## 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></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 int</td>
<td></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</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>

*address size = word size*
Byte Ordering

• How should bytes within a word be ordered in memory?
• Example: Store the 4-byte 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>0x100</td>
</tr>
<tr>
<td>0x101</td>
<td>0x101</td>
</tr>
<tr>
<td>0x102</td>
<td>0x102</td>
</tr>
<tr>
<td>0x103</td>
<td>0x103</td>
</tr>
<tr>
<td>a1</td>
<td>d4</td>
</tr>
<tr>
<td>b2</td>
<td>c3</td>
</tr>
<tr>
<td>c3</td>
<td>b2</td>
</tr>
<tr>
<td>d4</td>
<td>a1</td>
</tr>
</tbody>
</table>
```
**Byte Ordering Example**

```c
int x = 12345;
// long int = word
long int y = 12345;
```

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

---

### Byte Representation

<table>
<thead>
<tr>
<th>IA32, x86-64 x</th>
<th>SPARC x</th>
</tr>
</thead>
<tbody>
<tr>
<td>39 30 00 00</td>
<td>00 00 30 39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IA32 y</th>
<th>x86-64 y</th>
<th>32-bit SPARC y</th>
<th>64-bit SPARC y</th>
</tr>
</thead>
<tbody>
<tr>
<td>39 30 00 00</td>
<td>39 30 00 00</td>
<td>00 00 30 39</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

*higher addresses*  
*lower addresses*
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 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**
  • 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 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
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 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
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 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
Addresses and Pointers in C

\& = ‘address of’
* = ‘value at address’
or ‘dereference’
Addresses and Pointers in C

```c
int* ptr;
```

& = ‘address of’
*
= ‘value at address’
or ‘dereference’
Addresses and Pointers in C

```c
int* ptr;
```

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

`&` = ‘address of’
`*` = ‘value at address’ or ‘dereference’
Addresses and Pointers in C

```c
int* ptr;

int x = 5;
int y = 2;
```

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

\& = ‘address of’

\* = ‘value at address’ or ‘dereference’
Addresses and Pointers in C

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

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

`&` = ‘address of’

`*` = ‘value at address’ or ‘dereference’
Addresses and Pointers in C

```c
int* ptr;
int x = 5;
int y = 2;
ptr = &x;
```

- `int* ptr;` declares a variable, `ptr`, that is a pointer to (i.e., holds the address of) an int in memory.
- `int x = 5;` declares two variables, `x` and `y`, that hold ints, and sets them to 5 and 2, respectively.
- `int y = 2;` \& = ‘address of’
- `ptr = &x;` \* = ‘value at address’ or ‘dereference’
Addresses and Pointers in C

```c
int* ptr;
int x = 5;
int y = 2;
ptr = &x;
```

- Declares a variable, `ptr`, that is a pointer to (i.e., holds the address of) an int in memory.
- Declares two variables, `x` and `y`, that hold ints, and sets them to 5 and 2, respectively.
- Sets `ptr` to the address of `x`. Now, “`ptr` points to `x`.”

`&` = ‘address of’

`*` = ‘value at address’ or ‘dereference’
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`.”

y = 1 + *ptr;

& = ‘address of’
* = ‘value at address’ or ‘dereference’
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`.”

y = 1 + *ptr;

Sets `y` to 1 plus the value at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x;`
Addresses and Pointers in C

\[
\text{int* } \text{ptr;}
\]

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

\[
\text{int } x = 5;
\text{int } y = 2;
\]

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

\[
\text{ptr = &x;}
\]

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

\[
\text{y = 1 + *ptr;}
\]

Sets `y` to 1 plus the value at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x;`
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;

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

int y = 2;

ptr = &x;

Sets ptr to the address of x.

“Dereference ptr.”

Now, “ptr points to x.”

y = 1 + *ptr;

Sets y to 1 plus the value at the address held by ptr. Because ptr points to x, this is equivalent to y = 1 + x;

What is *(&y)?
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

- 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.

```c
int x, y;
// x is at location 0x04, y is at 0x18.
```

---

#### Assignment in C

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>x</td>
</tr>
<tr>
<td>0x08</td>
<td></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>y</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x24</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;

& = ‘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;`
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.

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td>3C</td>
<td>D0</td>
<td>00</td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x00 0x04 x 0x08 0x0C 0x10 0x14 0x18 y 0x1C 0x20 0x24
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

& = ‘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

\[
\begin{array}{cccc}
\text{0x00} & \text{0x04} & \text{x} & \\
\text{0x08} & \text{0x0C} & \text{0x10} & \\
\text{0x14} & \text{0x18} & \text{0x1C} & \\
\text{0x20} & \text{0x24} & & \\
\end{array}
\]

\& = ‘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

& = ‘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

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

Pointer arithmetic is scaled by size of target type.
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

0x18 = 24 (decimal)
\[ \frac{12}{36} = 0x24 \]

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

Pointer arithmetic is scaled by size of target type.
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

Pointer arithmetic can be dangerous.

0x18 = 24 (decimal)
+ 12
36 = 0x24

Pointer arithmetic is scaled by size of target type.
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?

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

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x04</th>
<th>0x08</th>
<th>0x0C</th>
<th>0x10</th>
<th>0x14</th>
<th>0x18</th>
<th>0x1C</th>
<th>0x20</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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!)
  - 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;`
  - // Get value of y, put it at the address stored in z

\[\begin{array}{cccc}
\text{0x00} & \text{0x04} & \text{x} & \text{0x08} \\
\text{0x0C} & \text{0x10} & \text{0x14} & \text{0x18} \\
\text{0x1C} & \text{0x20} & \text{z} & \text{0x24}
\end{array}\]

\& = ‘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
- *z = y;
  - // Get value of y, put it at the address stored in z

The target of a pointer is also a memory location.

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

- On the website: cs.uw.edu/351
  - Speedometer!
  - Anonymous feedback form
  - Lecture slides on the web schedule
  - Videos for optional reference.
- Make sure you are subscribed to the mailing list.
- Check the discussion boards regularly
- Enrolling in cse390a - ask your advisor (not me) for add code
- Non-majors waiting on accounts can pick up account sheets from the front desk on Friday or Monday
- Lab 0, having fun?
- Regular section meetings