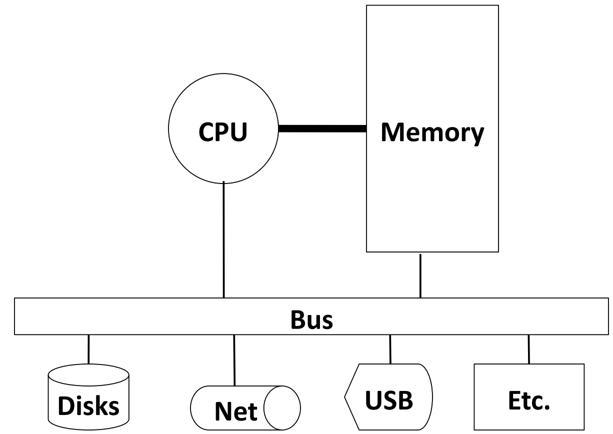
# Announcements

- **On the website: cs.uw.edu/351**
  - Speedometer!
  - Anonymous feedback form
  - Make sure you are subscribed to the mailing list
  - Lecture slides on the web schedule (these will be linked 1-2 days prior)
  - Lab 0, having fun? Make sure to start early
  - Discussion boards
  - Videos for optional reference – not exactly the same slides as we'll use
    - Tips for C, debugging, etc.
    - Lecture content
  - Office hours posted: if they don't work for you, let us know
- **Anyone not yet enrolled? If not, see me right after class**
- **New section being created for Th 11:30 – stay tuned**





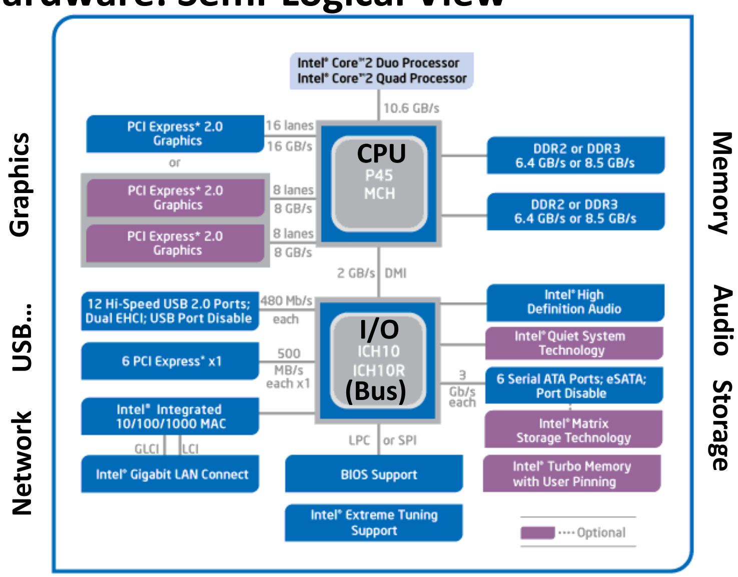

# Hardware: Logical View








# Hardware: Semi-Logical View

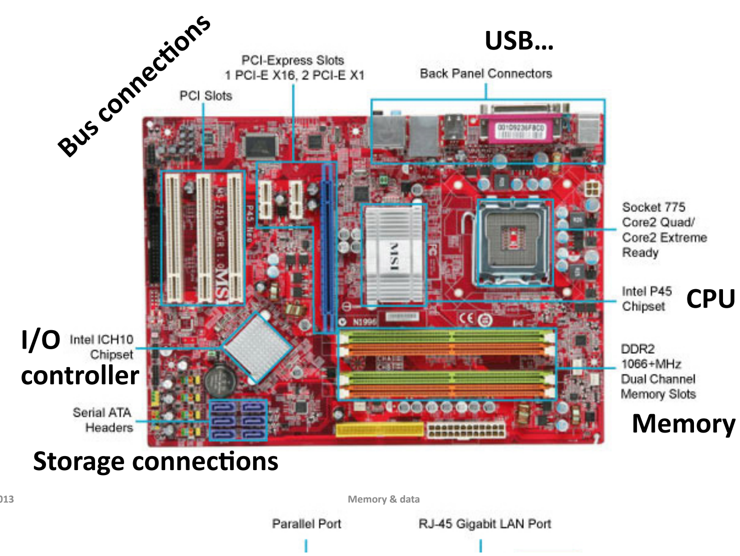



Intel® P45 Express Chipset Block Diagram

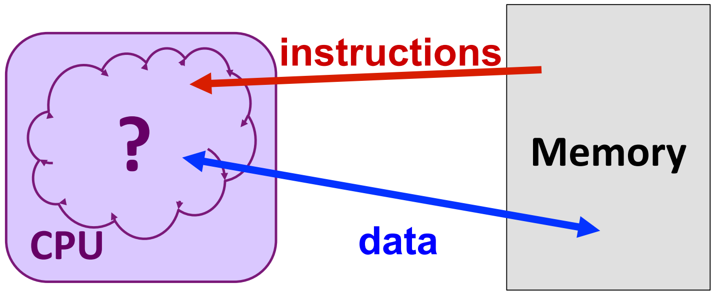




# Hardware: Physical View



# Hardware: 351 View (version 0)

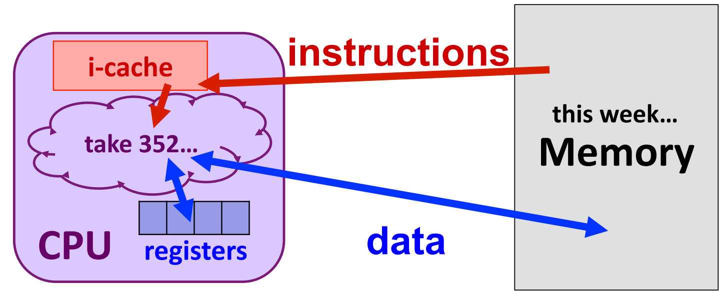



- CPU executes instructions; memory stores data
- To execute an instruction, the CPU must:
  - fetch an instruction;
  - fetch the data used by the instruction; and, finally,
  - execute the instruction on the data…
  - which may result in writing data back to memory.





# Hardware: 351 View (version 1)

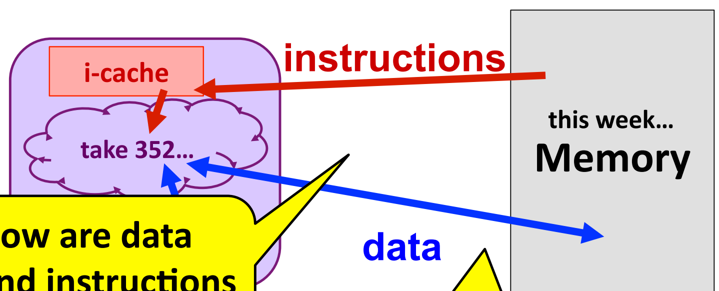



- The CPU holds instructions temporarily in the instruction cache
- The CPU holds data temporarily in a fixed number of registers
- Instruction and operand fetching is HW-controlled
- Data movement is programmer-controlled
- We'll learn about the instructions the CPU executes – take 352 to find out how it executes them





# Hardware: 351 View (version 1)




How are data and instructions represented?

How does a program find its data in memory?

- …ctions tem… cache.
- The CPU holds data temporarily … sters.
- Instruction fetching is HW-cont…
- Data movement is programmer…





# 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:

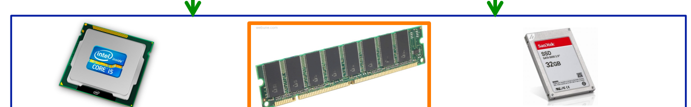
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:


Computer system:

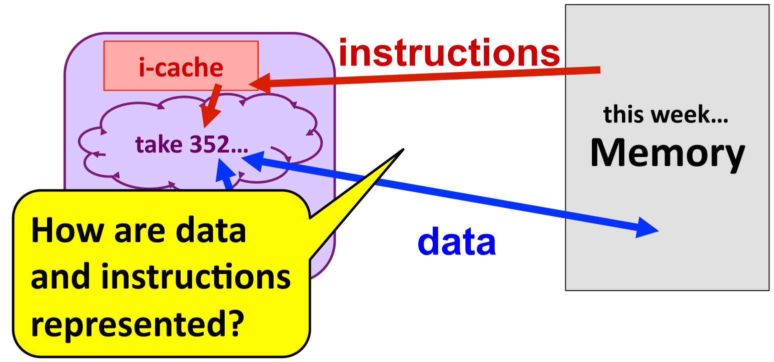


# Memory, Data, and Addressing

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





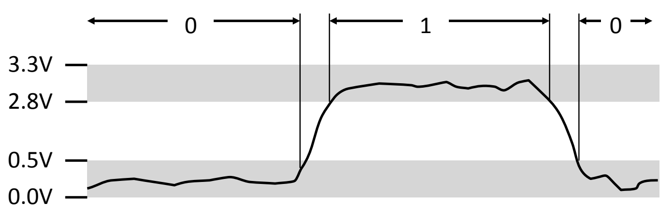







# Binary Representations

- **Base 2 number representation**
  - A base 2 digit (0 o 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



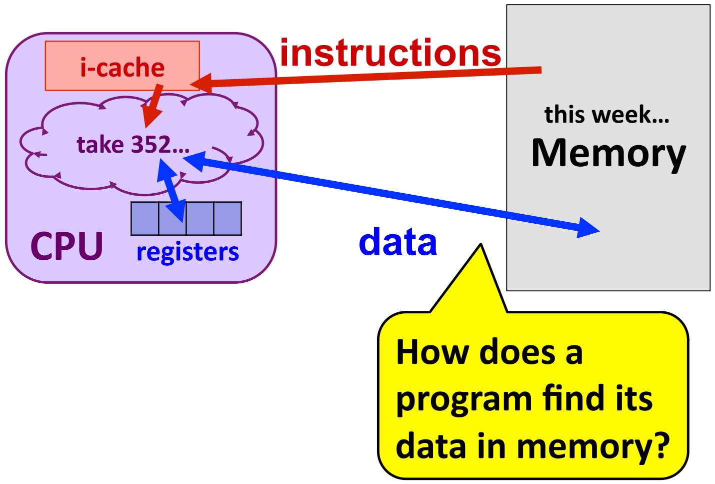





# Describing Byte Values

- **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…**

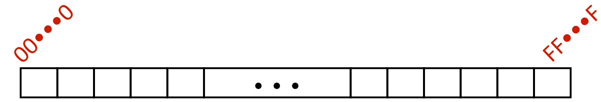
| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |








# 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

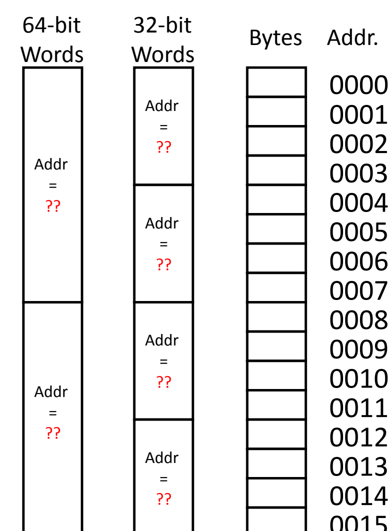
- **fixed number of contiguous bytes in memory, chosen by HW**
- **the largest unit of data a machine instruction can use**
- **word size = address size = register size**
- **Word size bounds the size of the *address space* and memory.**
  - word size = *w* bits => $2^w$ addresses
  - Until recently, most machines used **32-bit (4-byte) words**.
    - Potential address space: $\mathbf{2^{32}}$ addresses
      $2^{32}$ bytes ≈ **4 x $10^9$ bytes** = 4 billion bytes = **4GB**
      (living humans / addressable bytes ≈ 1.8)
    - Became too small for memory-intensive applications
  - Current x86 systems use **64-bit (8-byte) words**.
    - Potential address space: $\mathbf{2^{64}}$ addresses
      $2^{64}$ bytes ≈ **1.8 x $10^{19}$ bytes** = 18 billion billion bytes = **18 EB** (exabytes)
      (possible living acquaintances / addressable bytes ≈ 2.8)





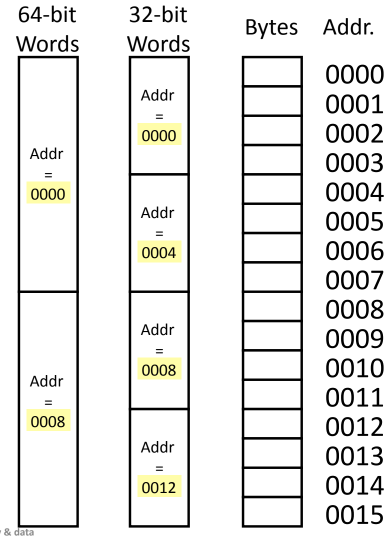
# Word-Oriented Memory Organization

- **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 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***

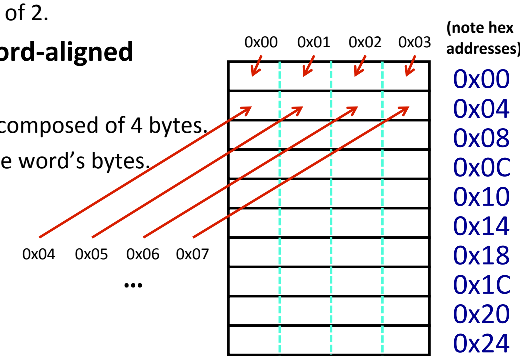







# Memory Alignment

- **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.



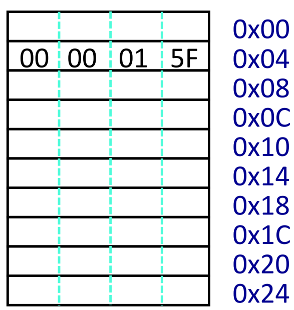

**More about alignment later in the course.**





# 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

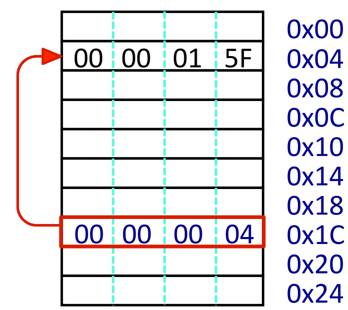






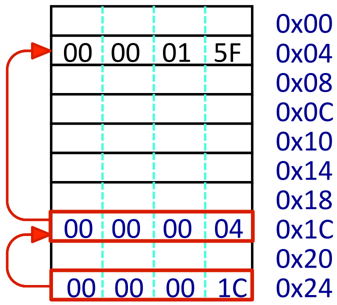

# Addresses and Pointers

- **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**.

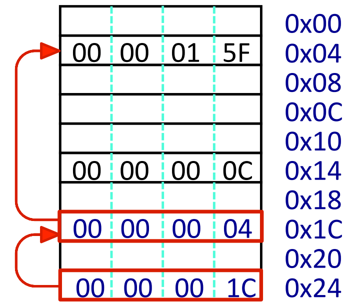







# Addresses and Pointers

- 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)**

| Java Data Type | C Data Type | Typical 32-bit | x86-64 |
|---|---|---|---|
| boolean | *bool* | 1 | 1 |
| byte | char | 1 | 1 |
| char | | 2 | 2 |
| short | short int | 2 | 2 |
| int | int | 4 | 4 |
| float | float | 4 | 4 |
| | long int | 4 | 8 |
| double | double | 8 | 8 |
| long | long long | 8 | 8 |
| | long double | 8 | 16 |
| (reference) | pointer * | 4 | 8 |

**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**

| | | | 0x100 | 0x101 | 0x102 | 0x103 | | |
|---|---|---|---|---|---|---|---|---|
| **Big Endian** | | | **a1** | **b2** | **c3** | **d4** | | |

| | | | 0x100 | 0x101 | 0x102 | 0x103 | | |
|---|---|---|---|---|---|---|---|---|
| **Little Endian** | | | **d4** | **c3** | **b2** | **a1** | | |

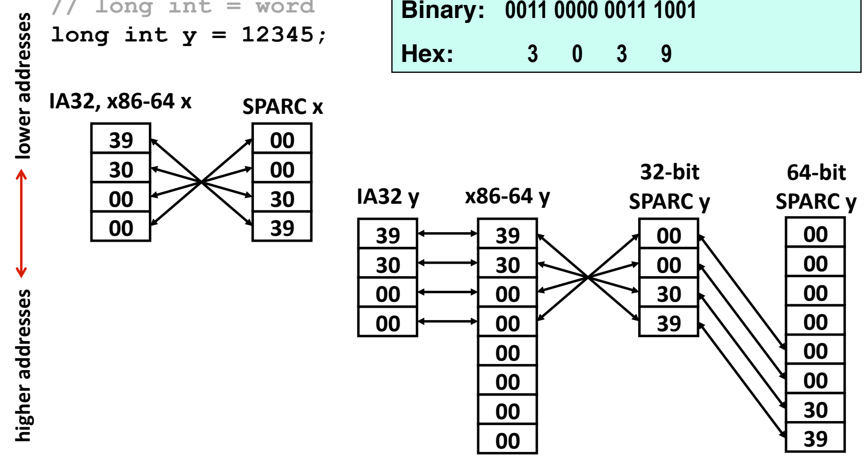




# Byte Ordering Example

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

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








# 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)*

| Address | Instruction Code | Assembly Rendition |
|---|---|---|
| 8048366: | 81 c3 ab 12 00 00 | add $0x12ab,%ebx |





# 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)*

| Address | Instruction Code | Assembly Rendition |
|---|---|---|
| 8048366: | 81 c3 ab 12 00 00 | add $0x12ab,%ebx |

**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'*

```
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`**."*

```
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

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

- **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.

| | | | | | |
|---|---|---|---|---|---|
| A7 | 00 | 32 | 00 | 0x00 | |
| 00 | 01 | 29 | F3 | 0x04 | x |
| EE | EE | EE | EE | 0x08 | |
| FA | CE | CA | FE | 0x0C | |
| 26 | 00 | 00 | 00 | 0x10 | |
| 00 | 00 | 10 | 00 | 0x14 | |
| 01 | 00 | 00 | 00 | 0x18 | y |
| FF | 00 | F4 | 96 | 0x1C | |
| 00 | 00 | 00 | 00 | 0x20 | |
| 00 | 42 | 17 | 34 | 0x24 | |





## Assignment in C

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

- **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.

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 00 | 01 | 29 | F3 | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 01 | 00 | 00 | 00 | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | |
| | | | | 0x24 | |





## Assignment in C

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

- **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;**

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 00 | 00 | 00 | 00 | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 01 | 00 | 00 | 00 | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | |
| | | | | 0x24 | |

# Assignment in C

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

- **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;**

| | | | | Address | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 00 | 00 | 00 | 00 | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | |
| | | | | 0x24 | |

**little endian!**





# Assignment in C

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

- **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 | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | |
| | | | | 0x24 | |





# Assignment in C

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

- **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 | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | z |
| | | | | 0x24 | |





# Assignment in C

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

- **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.

| | | | | Address | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| | | | | 0x20 | z |
| | | | | 0x24 | |

## Assignment in C

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

- **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.**

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| 24 | 00 | 00 | 00 | 0x20 | z |
| | | | | 0x24 | |

Pointer arithmetic is scaled by size of target type.





## Assignment in C

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

- **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?

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| 24 | 00 | 00 | 00 | 0x20 | z |
| | | | | 0x24 | |





## Assignment in C

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

- **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.

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 03 | 27 | D0 | 3C | 0x04 | x |
| | | | | 0x08 | |
| | | | | 0x0C | |
| | | | | 0x10 | |
| | | | | 0x14 | |
| 00 | 27 | D0 | 3C | 0x18 | y |
| | | | | 0x1C | |
| 24 | 00 | 00 | 00 | 0x20 | z |
| 00 | 27 | D0 | 3C | 0x24 | |





## Arrays in C

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

element type
name
number of elements

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

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

| | | | | |
|---|---|---|---|---|
| | | | | 0x00 |
| | | | | 0x04 |
| | | | | 0x08 |
| | | | | 0x0C |
| | | | | 0x10 |
| | | | | 0x14 |
| | | | | 0x18 |
| | | | | 0x1C |
| | | | | 0x20 |
| | | | | 0x24 |

# Arrays in C

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

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

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

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

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

| | | | | | |
|---|---|---|---|---|---|
| | | | | 0x00 | |
| 5F | 01 | 00 | 00 | 0x04 | a[0] |
| | | | | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| | | | | 0x1C | |
| | | | | 0x20 | |
| | | | | 0x24 | |





# Arrays in C

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

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

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

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

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

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

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





# Arrays in C

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

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

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

**Pointers:** int* p;

equivalent { 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, not a pointer to the array.

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 5F | 01 | 00 | 00 | 0x04 | a[0] |
| | | | | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

**Pointers:** int* p;

equivalent { p = a;
p = &a[0];

*p = 0xA;

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

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

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 5F | 01 | 00 | 00 | 0x04 | a[0] |
| | | | | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |

# Arrays in C

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

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

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

**Pointers:** **int* p;**

equivalent { **p = a;** **p = &a[0];** }
***p = 0xA;**

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

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

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| | | | | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

**Pointers:** **int* p;**

equivalent { **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, not a pointer to the array.

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

**Pointers:** **int* p;**

equivalent { **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.

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

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

**Pointers:** **int* p;**

equivalent { **p = a;** **p = &a[0];** }
***p = 0xA;**

equivalent { **p[1] = 0xB;** ***(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.

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

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

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |

# Arrays in C

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

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

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

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

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

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

**Pointers:** **int* p;**

equivalent { **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.

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 04 | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

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

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

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

**Pointers:** **int* p;**

equivalent { **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.

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| | | | | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 0C | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |





# Arrays in C

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

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

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

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

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

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

**Pointers:** **int* p;**

equivalent { **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.

| | | | | | |
|---|---|---|---|---|---|
| AD | 0B | 00 | 00 | 0x00 | |
| 0A | 00 | 00 | 00 | 0x04 | a[0] |
| 0B | 00 | 00 | 00 | 0x08 | a[1] |
| 0C | 00 | 00 | 00 | 0x0C | |
| | | | | 0x10 | … |
| | | | | 0x14 | |
| 5F | 01 | 00 | 00 | 0x18 | a[5] |
| AD | 0B | 00 | 00 | 0x1C | |
| 0C | 00 | 00 | 00 | 0x20 | p |
| | | | | 0x24 | |

***p = a[1] + 1;**





# 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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | ” | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ’ | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | I | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | del |

## 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 |
| H | a | r | r | y | | P | o | t | t | e | r | \0 |

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

- **How do we compute the string length?**



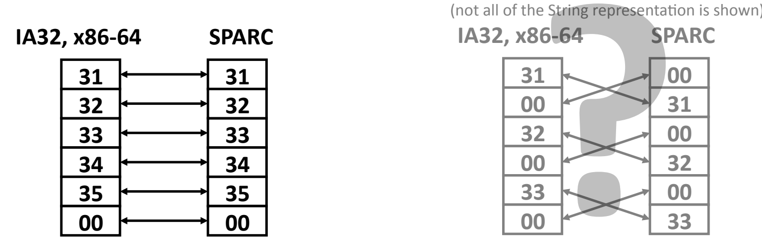


## Endianness and Strings

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

**Java** (char = 2 bytes) `String s = "123";`

(not all of the String representation is shown)




- **Byte ordering (endianness) is not an issue for 1-byte values.**
  - Arrays are not values; elements are values; chars are single bytes.
- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading zeros).
    Unicode can support the many characters in all languages in the world.
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)





## Examining Data Representations

- **Code to print byte representation of data**
  - Any data type can be treated as a *byte array* by casting it to `char`
  - C has *unchecked* casts. **<< DANGER >>**

```
typedef char byte;  // size of char == 1 byte

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

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

**printf directives:**

| | |
|---|---|
| %p | Print pointer |
| \t | Tab |
| %x | Print value as hex |
| \n | New line |





## `show_bytes` Execution Example

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

**Result (Linux):**

```
int a = 12345;
0x11ffffcb8    0x39
0x11ffffcb9    0x30
0x11ffffcba    0x00
0x11ffffcbb    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: ~A = 1 when A is 0 and vice-versa
  - DeMorgan’s Law: ~(A | B) = ~A & ~B

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| ~ | |
|---|---|
| 0 | 1 |
| 1 | 0 |





# General Boolean Algebras

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

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

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

```
  01010101
^ 01010101
  00000000
```

- **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 $j \in A$

```
01101001        { 0, 3, 5, 6 }
76543210

01010101        { 0, 2, 4, 6 }
76543210
```

- **Operations**
  - & Intersection `01000001` { 0, 6 }
  - | Union `01111101` { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference `00111100` { 2, 3, 4, 5 }
  - ~ Complement `10101010` { 1, 3, 5, 7 }





# Bit-Level Operations in C

- **`& | ^ ~`**
  - Apply to any “integral” data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
- **Examples (char data type)**
  - `~0x41 --> 0xBE`
    `~01000001`$_2$ `--> 10111110`$_2$
  - `~0x00 --> 0xFF`
    `~00000000`$_2$ `--> 11111111`$_2$
  - `0x69 & 0x55 --> 0x41`
    `01101001`$_2$ `& 01010101`$_2$ `--> 01000001`$_2$
  - `0x69 | 0x55 --> 0x7D`
    `01101001`$_2$ `| 01010101`$_2$ `--> 01111101`$_2$
- **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`
  - `0x69 && 0x55  -->  0x01`
  - `0x69 || 0x55  -->  0x01`
  - `p && *p++`   (avoids null pointer access, null pointer = 0x00000000 )