Modern Digital Computer (from the outside)

"von Neumann" model

Processor

HW-controlled

Memory

instructions

data

SW-controlled

How does a program find its data in memory?

Byte-Addressable Memory

Memory = array of byte locations, unique address = index.

Read/Write

Programs refer to bytes in memory by their addresses.

Address = word

Address space = range of possible addresses

Words in Memory

Address of word = address of 1st byte in word

Alignment

Data of size $n$ bytes stored at $a$
only if $a \mod n = 0$

$n$ is a power of 2

Recommended (x86), or required (MIPS) depending on platform.

Why?

Byte ordering: what is the "1st" byte in a word?

Endianness:

byte order within memory words

Bit order within bytes is always the same.
Little Endianness in Machine Code

**Disassembly**
Take binary machine code and generate an assembly code version.

**Instruction as stored in memory**
Shows byte encoding of instruction as stored in memory,
with byte in lower address on left and byte in higher address on right.

```
<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</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>
```

**Encodes:** add constant to register ebx
(temporary storage in 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</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>

**When does endianness matter?**

**Mostly invisible most of the time.**

**Matters only when inspecting memory byte-by-byte.**
For now: endianness matters ONLY IN MEMORY.
Memory stores bytes, so must define how to split larger values into bytes.
It also matters on the network or in files.

**Byte order within word is always natural within the processor.**
Processor manipulates entire words, so need to split them up.

**Bit order within bytes is always natural.**
Addresses and Pointers

address = number of location in memory

pointer = data object that holds an address

The value 240 is stored at address 0x20.

240_{10} = F0_{16} = 0x00 00 F0

A pointer stored at address 0x08 points to address 0x20.

A pointer to a pointer is stored at address 0x00.

The value 12 is stored at address 0x10.

Is it a pointer?

Are any of these values pointers?

Addresses and Pointers in C

int* ptr;

& = 'address of'

* = 'contents at address or dereference'

int x = 5;

int y = 2;

ptr = &x;

Set ptr to the address of x.
Now, "ptr points to x"

What is *(&y) ?

y = 1 + *ptr;

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

Data Representations

Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>32-bit word</th>
<th>64-bit word</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>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>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>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

Writing pointer types

Spaces between base type, *, and variable name do not matter.
The following are equivalent:

int* ptr;

I prefer this

Suggests: "The variable ptr holds the address of an int in memory."

int *ptr;

will see this a lot in others' code

Suggests: "There is an int in memory at the address held by the variable ptr."

Caveat: do not declare multiple variables on same line if using the former.
Assignment in C

A variable is represented by a memory location. Initially, it may hold any value.

```c
int x, y;
// x is at location 0x20, y is at 0x0C.
```

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.

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

Assignment in C

Left-hand-side = right-hand-side;
LHS must evaluate to a place to store a value.
RHS must evaluate to a value.
Store RHS value at LHS location.

```c
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
// Get value at y, add 3, put it in x.
int* z = &y;
// Get address of y, put it in z.
*z = y;
// What does this do?
```

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.

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

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

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

```c
*p = a[1] + 1;
```
Array Allocation
Basic Principle
T A[N];
Array of data type T and length N
Contiguously allocated region of N * sizeof(T) bytes

char string[12];

int val[5];

double a[3];

char* p[3];

Basic Principle
T A[N];
Array of data type T and length N
Identifier A can be used as a pointer to array element 0:
Type T*

Reference Type Value
val int
val int *
val+1 int *
&val[2] int *
val[5] int
*(val+1) int
val + i int *

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

Null-terminated Strings
— C strings are arrays of characters ending with the null character.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>115</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>121</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

— Compute the string length.
— Does Endianness matter for strings?
Since
- array name == address of 0th element
- array indexing == pointer arithmetic

C programmers often use * where you might expect []:
- e.g.: char* a:
  - pointer to a char
  - pointer to the first char in a string of unknown length

```
int main(int argc, char** argv);
int strcmp(char* a, char* b);
```

**Dynamic memory allocation**

```c
#include <stdlib.h>

void* malloc(size_t size)

- Returns a pointer to a memory block of at least size bytes
  (typically) aligned to 8-byte boundary
- If size == 0, returns NULL
- Unsuccessful: returns NULL and sets errno

void free(void* p)

- Returns the block pointed at by p to pool of available memory
- p must come from a previous call to malloc
```

**Malloc/free Example**

```c
void foo(int n, int m) {
  int i, *p;
  /* allocate a block of n ints */
  p = (int*)malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");      // print an error message
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;
  free(p);      /* return p to available memory pool */
}
```

**Malloc rules:**
- cast result to proper pointer type
- Use sizeof(...) to determine size

**Free rules:**
- Free only objects acquired from malloc, and only once.
- Do not use an object after freeing it.
Memory-Related Perils and Pitfalls in C

(Terrible things to do with pointers, part 1.)

Dereferencing bad pointers

See lab exercises for:
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks

Scanf: read formatted input

int val; // Declared, but not initialized
...           // holds anything.
scanf("%d", &val);

Read one int from input. Store it in memory at this address
i.e., store it in memory at the address where the contents of val is stored:
store into memory at 0xFFFFFF38.

val              0xFFFFF3C
0xFFFFF38
0xFFFFF34

The classic scanf bug

Forget one symbol... unleash certain doom.

int val; // Declared, but not initialized
...           // holds anything.
scanf("%d", val);

Read one int from input. Store it in memory at this address
i.e., store it in memory at the address given by the contents of val:
store into memory at 0xBAD4FACE.

val
BA D4 FA CE
0xFFFFF3C
0xFFFFF38
0xFFFFF34...
0xBAD4FACE

Best case: segmentation fault, or bus error, crash.
Worst case: silently corrupt data stored at address 0xBAD4FACE, and val still holds 0xBAD4FACE.
C memory error messages

11: segmentation fault
   accessing address outside legal area of memory

10: bus error
   accessing misaligned or other problematic address

Practice debugging in lab!

Why C?

Why learn C?
• Think like actual computer: abstraction very close to machine level.
• Understand just how much Your Favorite Language provides.
• Understand just how much Your Favorite Language might cost.
• Classic.
• Still (more) widely used (than it should be).
• Pitfalls still fuel many security vulnerabilities, devastating bugs today.

Why not use C?
• Almost definitely not the right language for your next personal project.
• It "gets out of the programmer's" way even when the programmer is running towards a blind cliff.
• Many advances in other programming languages since then fix a lot of its problems while keeping strengths.