CS 240 Stage 2

Hardware-Software Interface

Memory addressing, C language, pointers
Assertions, debugging
Machine code, assembly language, program translation
Control flow
Procedures, stacks
Data layout, security, linking and loading
Hardware

- Devices (transistors, etc.)
- Digital Logic
- Microarchitecture
- Instruction Set Architecture

Software

- Operating System
- Compiler/Interpreter
- Programming Language
- Program, Application

Solid-State Physics

Digital Logic

Microarchitecture

Instruction Set Architecture

Compiler/Interpreter

Operating System

Programming Language

Program, Application

Devices (transistors, etc.)
Programming with Memory
via C, pointers, and arrays

Why not just registers?
• Represent larger structures
• Computable addressing
• Indirection
Instruction Set Architecture (HW/SW Interface)

- **Instructions**
  - Names, Encodings
  - Effects
  - Arguments, Results

- **Local storage**
  - Names, Size
  - How many

- **Large storage**
  - Addresses, Locations

- **Processor**
  - Instruction Logic
  - Registers

- **Memory**
  - Encoded Instructions
  - Data

**Computer**
byte-addressable memory = mutable byte array

Cell / location = element
• Addressed by unique numerical address
• Holds one byte
•Readable and writable

Address = index
• Unsigned number
• Represented by one word
• Computable and storable as a value
multi-byte values in memory

Store across contiguous byte locations.

Alignment (Why?)

Bit order within byte always same.
Byte ordering within larger value?
**Endianness:** To store a multi-byte value in memory, which byte is stored first (at a lower address)?

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>2A</td>
</tr>
<tr>
<td>02</td>
<td>B6</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0B</td>
</tr>
</tbody>
</table>

**LittleEndian:** least significant byte first
- low order byte at low address, high order byte at high address
- used by x86, ...

**BigEndian:** most significant byte first
- high order byte at low address, low order byte at high address
- used by networks, SPARC, ...

Address	Contents
---
03	0B
02	00
01	B6
00	2A
Endianness in Machine Code

Address 8048366:

Contents: Instruction 81 c3 ab 12 00 00

Assembly Instruction add $0x12ab %ebx

encodes: add constant to register ebx

encodes constant operand (0x00012ab) in little endian order

assembly version omits leading zeros
Data, Addresses, and Pointers

address = index of a cell in memory

pointer = address represented as data
C: variables are memory locations (for now)

Compiler maps variable $\rightarrow$ memory location.
Declarations do not initialize!

```
int x;  // x at 0x20
int y;  // y at 0x0C

x = 0;  // store 0 at 0x20

// store 0x3CD02700 at 0x0C
y = 0x3CD02700;

// load the contents at 0x0C,
// add 3, and store sum at 0x20
x = y + 3;
```
C: Address and Pointer Primitives

*address* = index of a cell/location in memory

*pointer* = address represented as data

Expressions using addresses and pointers:

&___ address of the memory location representing ___

*___ contents at the memory address given by ___

a.k.a. "dereference ___"

Pointer types:

___* address of a memory location holding a ___
int* p;

int x = 5;
int y = 2;

p = &x;

y = 1 + *p;
Declare a variable, `p`:

```c
int* p;
```

that will hold the address of a memory location holding an int.

Declare two variables, `x` and `y`, that hold ints, and store 5 and 2 in them, respectively:

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

Get the address of the memory location representing `x`:

```c
p = &x;
```

... and store it in `p`. Now, "`p` points to `x`.”

Add 1 to the contents of memory at the address stored in `p`:

```c
y = 1 + *p;
```

... and store it in the memory location representing `y`.

`& = address of
* = contents at`
C: Address and Pointer Example

C assignment:
Left-hand-side = right-hand-side;

int* p;     // p: 0x04
int x = 5;   // x: 0x14, store 5 at 0x14
int y = 2;   // y: 0x24, store 2 at 0x24
p = &x;      // store 0x14 at 0x04
            // load the contents at 0x04 (0x14)
            // load the contents at 0x14 (0x5)
            // add 1 and store sum at 0x24
y = 1 + *p;
            // load the contents at 0x04 (0x14)
            // store 0xF0 (240) at 0x14
*p = 240;

& = address of
* = contents at
C: Pointer Type Syntax

Spaces between base type, *, and variable name mostly do not matter.

The following are equivalent:

```
int* ptr;
```

I see: "The variable `ptr` holds an address of an `int` in memory."

```
int * ptr;
```

```
int *ptr;
```

I prefer this

more common C style

I see: "Dereferencing the variable `ptr` will yield an `int`.
Or  "The memory location where the variable `ptr` points holds an `int`."  

Caveat: do not declare multiple variables unless using the last form.

```
int* a, b; means int *a, b; means int* a; int b;
```
Arrays are adjacent memory locations storing the same type of data. 

*a* is a name for the array’s base address, can be used as an *immutable* pointer.
C: Arrays

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

Indexing:
\[
\begin{align*}
a[0] &= 0x\text{f}0; \\
a[5] &= a[0]; \\
\end{align*}
\]

No bounds check:
\[
\begin{align*}
a[6] &= 0xBAD; \\
a[-1] &= 0xBAD; \\
\end{align*}
\]

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

Address of \( a[i] \) is base address \( a \) plus \( i \) times element size in bytes.

* \( a \) is a name for the array’s base address, can be used as an \textit{immutable} pointer.

\[
\begin{align*}
\text{array indexing} &= \text{address arithmetic} \\
\text{Both are scaled by the size of the type.} \\
\end{align*}
\]

\[
\begin{align*}
*p &= a[1] + 1; \\
\end{align*}
\]
C: Array Allocation

Basic Principle

\[ T \ A[N]; \]

Array of length \( N \) with elements of type \( T \) and name \( A \)

Contiguous block of \( N*\text{sizeof}(T) \) bytes of memory

char string[12];

int val[5];

double a[3];

char* p[3];
(or char *p[3]);

Use \texttt{sizeof} to determine proper size in C.
C: Array Access

Basic Principle

\[
T \ A[N];
\]

Array of length \( N \) with elements of type \( T \) and name \( A \)

Identifier \( A \) has type

\[
\text{int val[5];}
\]

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td></td>
</tr>
</tbody>
</table>
C: Null-terminated strings

C strings: arrays of ASCII characters ending with null character.

Does Endianness matter for strings?

int string_length(char str[]) {

}
C: * and []

C programmers often use * where you might expect []:

e.g., char*:
   • pointer to a char
   • pointer to the first char in a string of unknown length

int strcmp(char* a, char* b);

int string_length(char* str) {
   // Try with pointer arithmetic, but no array indexing.
## Memory Layout

<table>
<thead>
<tr>
<th>Addr</th>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>$2^N-1$</td>
<td></td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Statics</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Literals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Text</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C: Dynamic memory allocation in the heap

Heap:

Allocated block

Free block

Managed by memory allocator:

pointer to newly allocated block of at least that size

number of contiguous bytes required

\textbf{void* malloc(size\_t size);}\ 

pointer to allocated block to free

\textbf{void free(void* ptr);}\ 

43
C: Dynamic array allocation

```c
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) { // if error occurred
    perror("malloc"); // print error message
    exit(0); // end the program
}
zip[0] = 0;
zip[1] = 2;
zip[2] = 4;
zip[3] = 8;
zip[4] = 1;
printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
printf("
");
free(zip);
```
C: Arrays of pointers to arrays of ...

```c
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int*)*5);
...
int* zip0 = zips[0];
zip0[0] = 0;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][4] = 1;
```

```plaintext
zips

0 2 4 8 1
```
MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME A FEW POINTERS?

0x3A28213A
0x6339392C,
0x7363682E.

I HATE YOU.

http://xkcd.com/138/
C: scanf reads formatted input

```c
int val;
...
scanf("%d", &val);
```

Declared, but not initialized – holds anything.

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 0xFFFFFFFF38.

```
val
BA D4 FA CE
0x7FFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFF38
0x7FFFFFFFFFFFFF34
```
C: classic bug using `scanf`

```c
int val;
...
scanf("%d", val);
```

Declared, but not initialized — holds anything.

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.

<table>
<thead>
<tr>
<th>val</th>
<th>BA D4 FA CE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x7FFFFFFFFF3C</td>
</tr>
<tr>
<td>...</td>
<td>0x7FFFFFFFFF38</td>
</tr>
<tr>
<td></td>
<td>0x7FFFFFFFFF34</td>
</tr>
<tr>
<td>...</td>
<td>0x000000BAD4FACE</td>
</tr>
</tbody>
</table>

Best case: segmentation fault, or bus error, crash.

Bad case: silently corrupt data stored at address 0xBAD4FACE, and `val` still holds 0xBAD4FACE.

Worst case: arbitrary corruption
C: memory error messages

11: segmentation fault ("segfault", SIGSEGV)
accessing address outside legal area of memory

10: bus error
accessing misaligned or other problematic address

More to come on debugging!
C: Why?

Why learn C?

- Think like actual computer (abstraction close to machine level) without dealing with machine code.
- 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 devastating reliability and security failures today.

Why not use C?

- Probably not the right language for your next personal project.
- It "gets out of the programmer's way" even when the programmer is unwittingly running toward a cliff.
- Many advances in programming language design since then have produced languages that fix C's problems while keeping strengths.