CS240 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
C programming, pointers, arrays
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

address space

range of possible addresses

0x00•••0

0xFF•••F
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)?

**least significant byte**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 2A | B6 | 00 | 0B |

**most significant byte**

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

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

<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>

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

Address 8048366:

Contents: Instruction

| 81 | c3 | ab | 12 | 00 | 00 |

Assembly Instruction

```
add $0x12ab,%ebx
```

encodes: \textbf{add constant to register ebx}

codes constant operand \((0x000012ab)\) 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

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

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

The number 12 is stored at address 0x10.

Is it a pointer? How do we know values are pointers or not? How do we manage use of memory?

Memory drawn as 32-bit values, little endian order.
Compiler maps variable → memory location.
Declarations do not initialize!

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

x = 0; // store 0 at 0x20
y = 0x3CD02700;

// store 0x3CD02700 at 0x0C
x = y + 3;
```

// load the contents at 0x0C,
// add 3, and store sum at 0x20
C: Address and Pointer Primitives

\textit{address} = index of a cell/location in memory

\textit{pointer} = address represented as data

Expressions using addresses and pointers:

\begin{align*}
\&\text{___} & \text{address of the memory location representing ___} \\
\ast\text{___} & \text{contents at the memory address given by ___} \\
\text{a.k.a. "dereference ___"}
\end{align*}

Pointer types:

\begin{align*}
\text{___}\ast & \text{address of a memory location holding a ___}
\end{align*}
C: Address and Pointer Example

int* p;

int x = 5;
int y = 2;

p = &x;

y = 1 + *p;
C: Address and Pointer Example

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.
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;
```
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;
```
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: int a[6];

Indexing: a[0] = 0xf0;
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.

*p = a[1] + 1;

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

```
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 \textit{null} character.

\begin{verbatim}
0x48 0x61 0x72 0x72 0x79 0x20 0x50 0x6F 0x74 0x74 0x65 0x72 0x00
'H' 'a' 'r' 'r' 'y' ' ' 'P' 'o' 't' 't' 'e' 'r' '\0'
\end{verbatim}

Does Endianness matter for strings?

```c
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

```c
int strcmp(char* a, char* b);
int string_length(char* str) {

  // Try with pointer arithmetic, but no array indexing.
}
```
<table>
<thead>
<tr>
<th>Addr</th>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>Heap</td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run time</td>
</tr>
<tr>
<td>Statics</td>
<td>RW</td>
<td>Global variables/ static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Literals</td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Text</td>
<td>X</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>
C: Dynamic memory allocation in the heap

Managed by memory allocator:

- **malloc**
  - **Input:** `size_t size`
  - **Output:** Pointer to newly allocated block of at least `size` bytes

- **free**
  - **Input:** Pointer to allocated block `ptr`
  - **Effect:** Frees allocated block

Heap:

- Allocated block
- Free block

Managed by memory allocator:
#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("\n");
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;
```

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

I HATE YOU.

0x3A28213A
0x6339392C,
0x7363682E.

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 0xFFFFFFF38.

<table>
<thead>
<tr>
<th>val</th>
<th>0x7FFFFFFFFFFFFF3C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>0x7FFFFFFFFFFFFF38</td>
</tr>
<tr>
<td>D4</td>
<td>0x7FFFFFFFFFFFFF34</td>
</tr>
<tr>
<td>FA</td>
<td></td>
</tr>
<tr>
<td>CE</td>
<td></td>
</tr>
</tbody>
</table>
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.

```
val
BA  D4  FA  CE
```

```
  0x7FFFFFFFFFFFFFF3C
  0x7FFFFFFFFFFFFFF38
  0x7FFFFFFFFFFFFFF34
  0x00000000BAD4FACE
```

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
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!

http://xkcd.com/371/
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 probably 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.