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
Programming with Memory
via C, pointers, and arrays
Instruction Set Architecture (HW/SW Interface)

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

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

- **Large storage**
  - Addresses, Locations

- **Computer**

- **Processor**
  - Instruction Logic
  - Registers

- **Memory**
  - Encoded Instructions
  - Data
byte-addressable memory = mutable byte array

Fixed-length ordered sequence of cells

Cell = location = element
• Addressed by a unique numerical address
• Holds one byte
• Can be read and written by program

Address = index
• Unsigned number
• Represented by one word
• Can be computed and stored
multi-byte values in memory

Use $N$ contiguous byte locations to store an $N$-byte value.

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

Why?

Byte ordering:
Which byte is "first" in a multi-byte word?
**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

Bit order within bytes is always the same.
Endianness in x86 Machine Code

encodes: \textit{add constant to register ebx}

\begin{itemize}
  \item Address 8048366:
  \item Machine Instruction: 81 c3 \texttt{ab} 12 00 00
  \item Assembly Instruction: \texttt{add $0x12ab \%ebx}
\end{itemize}

encodes constant to add (0x000012ab) in little endian order

assembly version omits leading zeros
Data, Addresses, and Pointers

**address** = number of a location in memory

**pointer** = data that holds an address

The number 240 is stored at address **0x20**.

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

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?
C: variables are memory locations (for now)

Compiler manages the mapping from variable to memory. Declarations do not initialize!

```c
int x; // x stored at 0x20
int y; // y stored 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: Types determine sizes

### 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></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></td>
<td>long int</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></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
int* p;

Declare a variable, p, of type int* that is a pointer to (i.e., holds the address of) an int in memory.
(Does not initialize anything.)

int x = 5;
int y = 2;

Declare two variables, x and y, that hold ints, and set them to hold 5 and 2, respectively.

p = &x;

Set the variable p to hold the address of x.
Now, “p points to x.”

“Dereference p.”

y = 1 + *p;

Set y to hold: 1 plus the contents of memory at the address held by p. Because p points to x, this is equivalent to y=1+x;
C: Addresses and Pointers

Left-hand-side = right-hand-side;

RHS must provide a value.
LHS must provide a storage location.
Store RHS value in LHS location.

```c
int* p;          // p stored at 0x04
int x = 5;       // x stored at 0x14
int y = 2;       // y stored 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 address’
or ‘dereference’
C: Pointer Types

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

The following are equivalent:

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

I see: "The variable \text{ptr} holds an \textit{address of an int} in memory."

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

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

I prefer this

more common C style

I see: "Dereferencing the variable \text{ptr} will yield an \text{int}."

Or "The \textit{memory location} where the variable \text{ptr} points holds an \text{int}."

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

\[ \text{int}\ast \text{a, b; } \text{means int }\ast\text{a, b; } \text{means int}\ast \text{a; int b; } \]
C: Arrays

Declaration: int a[6];

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.
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;

{  
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 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 address of a[0] 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 \( N\times\text{sizeof}(T) \) bytes of memory

- `char string[12];`
- `int val[5];`
- `double a[3];`
- `char* p[3];` (or `char *p[3];`)

Use `sizeof` to determine proper size in C.
# C: Array Access

## Basic Principle

```plaintext
T  A[N];
```

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

Identifier \( A \) can be used as a pointer to array element 0: \( A \) has type \( T^* \)

```plaintext
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.

![Hexadecimal representation of a string](image)

Does Endianness matter for strings?

```c
int string_length(char str[]) {
}
```
C: * and []

- array name == address of 0th element
- array indexing == pointer arithmetic

So C programmers often use * where you might expect []:
- e.g.: char* is a:
  - 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.
}
```
Memory Layout

<table>
<thead>
<tr>
<th>Addr</th>
<th>Stack</th>
<th>Heap</th>
<th>Statics</th>
<th>Literals</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perm</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>Contents</td>
<td>Procedure context</td>
<td>Dynamic data structures</td>
<td>Global variables/static data structures</td>
<td>String literals</td>
<td>Instructions</td>
</tr>
<tr>
<td>Managed by</td>
<td>Compiler</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Compiler/Assembler/Linker</td>
<td>Compiler/Assembler/Linker</td>
<td>Compiler/Assembler/Linker</td>
</tr>
<tr>
<td>Initialized</td>
<td>Run-time</td>
<td>Run-time</td>
<td>Startup</td>
<td>Startup</td>
<td>Startup</td>
</tr>
</tbody>
</table>

Memory Layout:

- **Stack**: RW, Procedure context, Compiler, Run-time.
- **Heap**: RW, Dynamic data structures, Programmer, malloc/free, new/GC, Run-time.
- **Statics**: RW, Global variables/static data structures, Compiler/Assembler/Linker, Startup.
- **Literals**: R, String literals, Compiler/Assembler/Linker, Startup.
- **Text**: X, Instructions, Compiler/Assembler/Linker, Startup.
#include <stdlib.h>

void* malloc(size_t size)

Successful:
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
void foo(int n, int m) {
    // allocate a block of n ints
    int* p = (int *)malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc"); // print an error message
        exit(0);
    }
    for (int 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.
Man, I suck at this game.
Can you give me a few pointers?

I hate you.

0x3A28213A
0x6339392C,
0x7363682E.

http://xkcd.com/138/
C: Memory-Related Perils and Pitfalls
Terrible things to do with pointers, part 1.

Dereferencing bad pointers

See later exercises for:
Reading uninitialized memory
Overwriting memory
Referencing nonexistent variables
Freeing blocks multiple times
Referencing freed blocks
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>0xFFFFFFF3C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BA D4 FA CE</td>
</tr>
<tr>
<td>0x0</td>
<td>0xFFFFFFF38</td>
</tr>
<tr>
<td></td>
<td>0xFFFFFFF34</td>
</tr>
</tbody>
</table>
C: classic bug using scanf

int val;
...
scanf("%d", \ val);

Declared, but not initialized – holds anything.

Read one int from input.
Store it in memory at this address.

<table>
<thead>
<tr>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i.e., store it in memory at the address given by the contents of val: store into memory at 0xBAD4FACE.

<table>
<thead>
<tr>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xFFF</td>
<td></td>
<td></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
   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 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 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 other programming languages since then fix a lot of C's problems while keeping strengths.