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

the memory model
pointers and arrays in C

Software
- Program, Application
  - Programming Language
  - Compiler/Interpreter
  - Operating System
  - Instruction Set Architecture
  - Microarchitecture
  - Digital Logic
  - Devices (transistors, etc.)
  - Solid-State Physics

Hardware

Instruction Set Architecture (HW/SW Interface)

- processor
- memory
- encoded instructions
- instruction logic
- registers
- data
- local storage
  - names, size
  - arguments, results
- large storage
  - addresses, locations

Computer
Byte-addressable memory = mutable byte array

Location / cell = element
- Identified by unique numerical address
- Holds one byte

Address = index
- Unsigned number
- Represented by one word
- Computable and storable as a value

Operations:
- Load: read contents at given address
- Store: write contents at given address

Multi-byte values in memory

Store across contiguous byte locations.
Example: 8 byte (64 bit) values

Alignment
Multi-byte values start at addresses that are multiples of their size

Bit order within byte always same.
Recall: byte ordering within larger value?
### Data, addresses, and pointers

**address** = index of a location in memory  
**pointer** = a reference to a location in memory, represented as an address stored as data

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x00</td>
</tr>
<tr>
<td>0001</td>
<td>0x04</td>
</tr>
<tr>
<td>0002</td>
<td>0x08</td>
</tr>
<tr>
<td>0003</td>
<td>0x10</td>
</tr>
<tr>
<td>0004</td>
<td>0x14</td>
</tr>
<tr>
<td>0005</td>
<td>0x18</td>
</tr>
<tr>
<td>0006</td>
<td>0x1C</td>
</tr>
<tr>
<td>0007</td>
<td>0x20</td>
</tr>
<tr>
<td>0008</td>
<td>0x24</td>
</tr>
</tbody>
</table>

Let’s store the number 240 at address 0x20.  

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

At address 0x08 we store a pointer to the contents at address 0x20.  
At address 0x00, we store a pointer to a pointer.  
The number 12 is stored at address 0x10.  
Is it a pointer?  
How do we know if values are pointers or not?  
How do we manage use of memory?

For these slides, we’ll draw the bytes in this reverse order so that multi-byte values can be read directly.  
memory drawn as 32-bit values, little endian order

---

### Endianness: details

In what order are the individual bytes of a multi-byte value stored in memory?

**Little Endian:** least significant byte first  
- low order byte at low address  
- high order byte at high address  
- used by x86, ... and CS240!

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x0B</td>
</tr>
<tr>
<td>0001</td>
<td>0x00</td>
</tr>
<tr>
<td>0002</td>
<td>0x00</td>
</tr>
<tr>
<td>0003</td>
<td>0x00</td>
</tr>
</tbody>
</table>

**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>0000</td>
<td>0x0B</td>
</tr>
<tr>
<td>0001</td>
<td>0x00</td>
</tr>
<tr>
<td>0002</td>
<td>0x00</td>
</tr>
<tr>
<td>0003</td>
<td>0x00</td>
</tr>
</tbody>
</table>
C: Variables are locations

The compiler creates a map from variable name → location.
Declarations do not initialize!

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

x = 0; // store 0 @ 0x20
y = 0x3CD02700; // store 0x3CD02700 @ 0x0C

// 1. load the contents @ 0x0C
// 2. add 3
// 3. store sum @ 0x20
x = y + 3;
```

C: Pointer operations and types

- **address** = index of a location in memory
- **pointer** = a reference to a location in memory, an address stored as data

**Expressions using addresses and pointers:**
- `& ___` address of the memory location representing ___
  a.k.a. "reference to ___"
- `* ___` contents at the memory address given by ___
  a.k.a. "dereference ___"

**Pointer types:**
- `___*` address of a memory location holding a ___
  a.k.a. "a reference to a ___"

C: Types determine sizes

<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>long</td>
<td>long int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>(reference)</td>
<td>(pointer) *</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

address size = word size
C: Pointer example

Declare a variable, p

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

Take the address of the memory representing x

... and store it in the memory location representing p.

Now, "p points to x."

Add 1 to the contents of memory at the address given by the contents of the memory location representing p.

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

What is the type of *p?
What is the type of &x?
What is *(&y)?
C: Pointer type syntax

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

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

```c
int * ptr;
Looks like: "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.

```c
int* a, b;
means
```

```c
int* a, int b;
means
```

C: Arrays

Declaration:

```c
int a[6];
```

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.

Indexing:

```c
a[0] = 0xf0;
```

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

C: Arrays

Declaration:

```c
int a[6];
```

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: Arrays**

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

**Indexing:**
- `a[0] = 0xf0;
- `a[5] = a[0];`

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

**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:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0xf0;
- `a[5] = a[0];`

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

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

**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:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0xf0;
- `a[5] = a[0];`

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

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

**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:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0xf0;
- `a[5] = a[0];`

**Pointers:**
```c
int* p;
```

```c
p = &a[0];
```

**Equivalent:**
```c
int* p;
```

```c
int* p = &a[0];
```

**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:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0xf0;
- `a[5] = a[0];`

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

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

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

- 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;`

Both are scaled by the size of the type.

Indexing: `array indexing = address arithmetic`

- `p[i]` is base address `a` plus `i` times element size in bytes.
- `p[i+1]` is base address `a` plus `i+1` times element size in bytes.

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

- 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;`

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

Pointers: `check: No bounds`

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

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

- 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;`

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

- `p[i]` is base address `a` plus `i` times element size in bytes.
- `p[i+1]` is base address `a` plus `i+1` times element size in bytes.

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

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

\[
\begin{align*}
\text{Declaration:} & \quad \text{int } a[6]; \\
\text{Indexing:} & \quad a[0] = 0xf0; \quad a[5] = a[0]; \\
\text{No bounds} & \quad a[6] = 0xBAD; \quad a[-1] = 0xBAD; \\
\text{check:} & \quad \text{Address of } a[1] \text{ is base address } a \text{ plus } i \text{ times element size in bytes.} \\
\text{Pointers:} & \quad \text{int* } p; \quad p = a; \quad p = &a[0]; \quad *p = 0xA; \\
\text{equivalent} & \quad \text{int* } p; \quad p = a; \quad p = &a[0]; \quad *p = 0xA; \\
\text{array indexing} & \quad \{ p[1] = 0xB; \quad *p = p + 2; \} \\
\text{both scaled by the size of the type.} & \quad \{ p[1] = 0xB; \quad *p = p + 2; \}
\end{align*}
\]

Assume \( p \) has type int *. Are \`p[2] = 5` and \`*p = 0xB` equivalent? What about \`p[2] = 5` and \`*p = 0xB`? 

\[
\begin{align*}
\text{No; No.} & \quad 0x00 \ 0x00 \ 0x04 \ A \quad 0x00 \ 0x02 \ 0x0A \ 0x09 \ 0x08 \ 0x04 \ 0x04 \ 0x04 \\
\text{No; Yes.} & \quad 0x00 \ 0x00 \ 0x04 \ A \quad 0x00 \ 0x02 \ 0x0A \ 0x09 \ 0x08 \ 0x04 \ 0x04 \ 0x04 \\
\text{Yes; No.} & \quad 0x00 \ 0x00 \ 0x04 \ A \quad 0x00 \ 0x02 \ 0x0A \ 0x09 \ 0x08 \ 0x04 \ 0x04 \ 0x04 \\
\text{Yes; Yes.} & \quad 0x00 \ 0x00 \ 0x04 \ A \quad 0x00 \ 0x02 \ 0x0A \ 0x09 \ 0x08 \ 0x04 \ 0x04 \ 0x04 \\
\end{align*}
\]
C: Array allocation

Basic Principle

T A[N];
Array of length N with elements of type T and name A
Identifier A has type T*

Use sizeof to determine proper size in C.

Contiguous block of N * sizeof(T) bytes of memory

size depends on the machine word size

C: Array access

Basic Principle

T A[N];
Array of length N with elements of type T and name A
Identifier A has type T*

Expression | Type | Value
---|---|---
val[4] | int | 1
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
C: Null-terminated strings

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

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);
```
C: Dynamic memory allocation in the heap

**Heap:**

![Heap diagram]

- Allocated block
- Free block

**Managed by memory allocator:**

- Pointer to newly allocated block of at least that size
- Number of contiguous bytes required
- Pointer to allocated block to free

```c
#include <stdlib.h>

void* malloc(size_t size);

void free(void* ptr);
```

### Rules:
- Check for error result.
- Cast result to relevant pointer type.
- Use sizeof(...) to determine size.

```c
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) {
    // if error occurred
    perror("malloc");
    exit(0);
}
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: Standard memory allocator

```c
#include <stdlib.h>

void* malloc(size_t size)

Allocates a memory block of at least `size` bytes and returns its address.
If memory error (e.g., allocator has no space left), returns NULL.

Rules:
- Check for error result.
- Cast result to relevant pointer type.
- Use sizeof(...) to determine size.

void free(void* ptr)

Deallocates the block referenced by `ptr`, making its space available for new allocations.

`ptr` must be a `malloc` result that has not yet been freed.

Rules:
- `ptr` must be a `malloc` result that has not yet been freed.
- Do not use `*ptr` after freeing.
```

---

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");
    exit(0);
}
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: Array of pointers to arrays of ints

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

---

Why terminate with NULL?

Why no NULL?
// return a count of all zips that end with digit endNum
int zipCount(int* zips[], int endNum) {
    int count = 0;
    int** cursor = zips;
    while (*cursor) {
        if ((*cursor)[4] == endNum) {
            count++;
        }
        cursor ++;
    }
    return count;
}

C: scanf reads formatted input

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

Read one int in decimal_10 format from input.
Store it in memory at this address.

val
CA PE 12 34

0x7FFFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFFF38
0x7FFFFFFFFFFFFFF34

C: Classic bug using scanf

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

Read one int in decimal_10 format from input.
Store it in memory at this address.

val
BA D4 PA CE

0x7FFFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFFF38
0x7FFFFFFFFFFFFFF34

0x0000000BAD4FACE

Best case: 🤡 crash immediately with segmentation fault/bus error.
Bad case: 😞 silently corrupt data stored @ 0xBAD4FACE, fail to store input in val, and keep going.
Worst case: 😨 program does literally anything.
C: Memory error messages

11: segmentation fault ("segfault", SIGSEGV)
   accessing address outside legal area of memory
10: bus error (SIGBUS)
   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.
- Advances in programming language design since the 70’s have produced languages that fix C’s problems while keeping strengths.