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

https://cs.wellesley.edu/~cs240/s20/
Programming with Memory

pointers and arrays in C
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

Programming with Memory
Byte-addressable memory = mutable byte array

Location / cell = 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>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A  B6  00  0B</td>
</tr>
</tbody>
</table>

**most significant byte**

**least significant byte**

<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>

**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, ...

Programming with Memory
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

The number 240 is stored at address 0x20.

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 locations

Compiler maps variable name → location.

Declarations do not initialize!

```c
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: Pointer operations and types

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

**pointer** = a reference to a location in memory, represented as 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: Pointer example

```c
int* p;

int x = 5;
int y = 2;

p = &x;

y = 1 + *p;
```
 Declare a variable, \( p \)

\[
\text{int* } p;
\]

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

\[
\text{int } x = 5; \\
\text{int } y = 2;
\]

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

\[
p = \& x;
\]

Get the address of the memory location representing \( x \)

... and store it in \( p \). Now, “\( p \) points to \( x \).”

Add 1 to the contents of memory at the address stored in \( p \)

\[
y = 1 + *p;
\]

... and store it in the memory location representing \( y \).
C: 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;
```
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;
```

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.

```
int* a, b; means int *a, b; means int* a; int b;
```
C: Arrays

Declaration: `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.
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 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 \( N \times \text{sizeof}(T) \) bytes of memory

\[
\text{char string}[12]; \quad \text{x} \quad \text{x + 12} \\
\text{int val[5];} \quad \text{x} \quad \text{x + 4} \quad \text{x + 8} \quad \text{x + 12} \quad \text{x + 16} \quad \text{x + 20} \\
\text{double a[3];} \quad \text{x} \quad \text{x + 8} \quad \text{x + 16} \quad \text{x + 24} \\
\text{char* p[3];} \quad \text{(or char* p[3];)} \quad \text{x} \quad \text{x + 4} \quad \text{x + 8} \quad \text{x + 12} \\
\]

Use \text{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];
```

Expression | Type   | Value
---        | ---    | ---
val[4]     | int    | 1
val        | int *  | 
val+1      | int *  | 
&val[2]    | int *  | 
val[5]     | int    | 
*(val+1)   | int    | 
val + i    | int *  |
C: Null-terminated strings

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

```c
int string_length(char str[]) {
    // Why?
}
```
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.
}
```
C: 0 vs. '\0' vs. NULL

<table>
<thead>
<tr>
<th>0</th>
<th>' \0 '</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: zero</td>
<td>Name: null character</td>
<td></td>
</tr>
<tr>
<td>Type: int</td>
<td>Type: char</td>
<td></td>
</tr>
<tr>
<td>Size: 4 bytes</td>
<td>Size: 1 byte</td>
<td></td>
</tr>
<tr>
<td>Value: 0x00000000</td>
<td>Value: 0x00</td>
<td></td>
</tr>
</tbody>
</table>

Is it important/necessary to encode the null character or the null pointer as 0x0?

What happens if a programmer mixes up these "zeroey" values?
# Memory address-space 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>$2^N$-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

- **Addr**: Memory address
- **Perm**: Access permissions (R: read, W: write, X: execute)
- **Contents**: Types of data structures
- **Managed by**: Entities responsible for managing the data structures
- **Initialized**: When the data structures are initialized
C: Dynamic memory allocation in the heap

Heap:

- Allocated block
- Free block

Managed by memory allocator:

- `void* malloc(size_t size);`
- `void free(void* ptr);`

- Pointer to newly allocated block of at least that size
- Number of contiguous bytes required
- Pointer to allocated block to free
#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: 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;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][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) {

}
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 given by the contents of `val`:
store into memory at 0xBAD4FACE.

<table>
<thead>
<tr>
<th>val</th>
<th>0x7FFFFFFF3C</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA D4 FA CE</td>
<td>0x7FFFFFFF38</td>
</tr>
<tr>
<td></td>
<td>0x7FFFFFFF34</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.

<table>
<thead>
<tr>
<th>val</th>
<th>0x7FFFFFFFFFFFFFF3C</th>
<th>0x7FFFFFFFFFFFFFF38</th>
<th>0x7FFFFFFFFFFFFFFF34</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA</td>
<td>D4</td>
<td>FA</td>
<td>CE</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>FE</td>
<td>12</td>
<td>34</td>
</tr>
</tbody>
</table>

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
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!

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 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.