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/
Programming with Memory

the memory model
pointers and arrays in C

https://cs.wellesley.edu/~cs240/
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

**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?
Is an `int` stored at address 0x00000002 aligned?

Yes

No

Maybe
Is an `int` stored at address 0x00000002 aligned?

Yes 0%
No 0%
Maybe 0%
Is an `int` stored at address 0x00000002 aligned?

<table>
<thead>
<tr>
<th>Choice</th>
<th>Votes</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>No</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Maybe</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
**endianness:** details

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

<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>

### 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>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>

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

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

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

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 **0x12** 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?

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x08</th>
<th>0x10</th>
<th>0x14</th>
<th>0x18</th>
<th>0x1C</th>
<th>0x20</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>F0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0C</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>20</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>08</td>
</tr>
</tbody>
</table>

memory drawn as 32-bit values, little endian order
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;

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

The compiler creates a map from variable name $\rightarrow$ location.
Declarations do not initialize!

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

x = 0; // store 0 @ 0x20

y = 0x3CD02700;

// 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

**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*
C: Pointer example

Declare a variable, \( p \)

\[
\text{int}\ *\ p; \\
\text{that will hold the address of a memory location holding an int}
\]

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

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

Take the address of the memory representing \( x \)

\[
p = \&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 \( p \)

\[
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 @ 0x14
int y = 2; // y @ 0x24, store 2 @ 0x24
p = &x;   // store 0x14 @ 0x04

// 1. load the contents @ 0x04 (=0x14)
// 2. load the contents @ 0x14 (=0x5)
// 3. add 1
// 4. store sum as contents @ 0x24
y = 1 + *p;

// 1. load the contents @ 0x04 (=0x14)
// 2. store 0xF0 as contents @ 0x14
*p = 240;
```

& = address of
* = contents at

What is the type of *p?
What is the type of &x?
What is *(&y) ?
What is the result of printing the decimal values of \`a\` and \`b\` at the end of this code?

```c
int a = 1;
int b = 5;
int* p = &a;
*p = *p + 1;
a = a + 1;
p = &b;
*p = *p * 2;
```

Options:
- 2, 10
- 3, 5
- 3, 10
- 6, 5
- None of the above
What is the result of printing the decimal values of `a` and `b` at the end of this code?

```c
int a = 1;
int b = 5;
int* p = &a;
*p = *p + 1;
a = a + 1;
p = &b;
*p = *p * 2;
```
What is the result of printing the decimal values of `a` and `b` at the end of this code?

```c
int a = 1;
int b = 5;
int* p = &a;
*p = *p + 1;
a = a + 1;
p = &b;
*p = *p * 2;
```

Options:

- 2, 10
- 3, 5
- 3, 10
- 6, 5
- None of the above
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;
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;

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:  
int a[6];

Indexing:  
a[0] = 0xf0;
a[5] = 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.

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

Declaration:  int a[6];

Indexing:    a[0] = 0xf0;
              a[5] = a[0];

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

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.
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:**

```c
a[0] = 0xf0;
a[5] = a[0];
```

**No bounds check:**

```c
a[6] = 0xBAD;
a[-1] = 0xBAD;
```
C: Arrays

Declaration:  
int a[6];

Indexing:  
a[0] = 0xf0;
a[5] = a[0];

No bounds check:  
a[6] = 0xBAD;
a[-1] = 0xBAD;

Pointers:  

\[
\begin{align*}
\text{int}^{*} & \quad \text{p}; \\
p & = a; \\
p & = &\&a[0];
\end{align*}
\]

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 \textit{immutable} pointer.

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

\[\begin{array}{cccc}
00 & 00 & 0B & AD \\
00 & 00 & 00 & F0 \\
0x24 & a[5] \\
0x20 & a[4] \\
0x1C & a[3] \\
0x18 & a[2] \\
0x14 & a[1] \\
0x10 & a[0] \\
0x0C & \ldots \\
0x08 & a[-1] \\
0x04 & \&a[0] \\
0x00 & \&a[1]
\end{array}\]
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;
\[
\begin{align*}
P &= a; \\
P &= &a[0]; \\
*p &= 0xA;
\end{align*}
\]

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

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

Address of \texttt{a[i]} is base address \texttt{a} plus \(i\) times element size in bytes.
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;
```

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.

```
0x24   0x20   0x1C   0x18   0x14   0x10   0x0C   0x08   0x04   0x00
```

```
00 00 00 0B  AD  0xBAD
00 00 00 F0  0xf0
00 00 00 0A  0xA
00 00 00 0B  AD  0xBAD
00 00 00 0C  0xC
```

 equivalents `{ `p = a; p = &a[0]; *p = 0xA;` }`
**C: Arrays**

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

Indexing:  
```c
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:  
```c
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:  
```c
int* p;
p = a;
p = &a[0];
*p = 0xA;
p[1] = 0xB;
*(p + 1) = 0xB;
```

**array indexing = address arithmetic**  
Both are scaled by the size of the type.

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:  

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

Indexing:  

\[
\begin{align*}
\text{a[0]} &= 0xf0; \\
\text{a[5]} &= \text{a[0]};
\end{align*}
\]

No bounds check:  

\[
\begin{align*}
\text{a[6]} &= 0xBAD; \\
\text{a[-1]} &= 0xBAD;
\end{align*}
\]

Pointers:  

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

array indexing = address arithmetic  
Both are scaled by the size of the type.

\[
\begin{align*}
\text{p[1]} &= 0xB; \\
*(\text{p + 1}) &= 0xB; \\
\text{p} &= \text{p + 2};
\end{align*}
\]

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

\[
\begin{align*}
\text{Address of } a[5] &= 0x24 \\
\text{Address of } a[0] &= 0x20 \\
\text{Address of } p &= 0x00
\end{align*}
\]

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 \textit{immutable} pointer.
# C: Arrays

### Declaration:

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

### Indexing:

```c
a[0] = 0xf0;
a[5] = a[0];
```

### No bounds check:

```c
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

### Pointers:

```c
int* p;
```

**equivalent**

```c
p = a;
p = &a[0];
*p = 0xA;
p[1] = 0xB;
*(p + 1) = 0xB;
p = p + 2;
```

---

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.

---

**array indexing = address arithmetic**

Both are scaled by the size of the type.
C: Arrays

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

Indexing:
\[
a[0] = 0xf0;
a[5] = a[0];
\]

No bounds check:
\[
a[6] = 0xBAD;
a[-1] = 0xBAD;
\]

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

\[
\begin{align*}
& p[1] = 0xB; \\
& *(p + 1) = 0xB; \\
& p = p + 2;
\end{align*}
\]

\small{
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.

- No; No.
- No; Yes.
- Yes; No.
- Yes; Yes.

- No; No. 0%
- No; Yes. 0%
- Yes; No. 0%
- Yes; Yes. 0%

No; No. 0%
No; Yes. 0%
Yes; No. 0%
Yes; Yes. 0%
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

Use `sizeof` to determine proper size in C.

Size depends on the machine word size
C: Array access

Basic Principle

\[ T \quad A[N]; \]
Array of length \( N \) with elements of type \( T \) and name \( A \)
Identifier \( A \) has type \( T^* \)

```
int val[5];
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>1</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>
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

| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | !      | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | "      | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | #      | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $      | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | %      | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | &      | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | '      | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (      | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | )      | 57 | 9 | 73 | I | 89 | Y | 105 | l | 121 | y |
| 42 | *      | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | +      | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | ,      | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 45 | -      | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | .      | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | /      | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | del |
C: Null-terminated strings

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

```
int string_length(char str[]) {

}
```

Does Endianness matter for strings?
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);
# C: 0 vs. '\0' vs. NULL

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>int</td>
<td>4 bytes</td>
<td>0x00000000</td>
<td>The integer zero.</td>
</tr>
<tr>
<td>'\0'</td>
<td>null character</td>
<td>char</td>
<td>1 byte</td>
<td>0x00</td>
<td>Terminator for C strings.</td>
</tr>
<tr>
<td>NULL</td>
<td>null pointer / null reference / null address</td>
<td>void*</td>
<td>1 word (= 8 bytes on a 64-bit architecture)</td>
<td>0x0000000000000000</td>
<td>The absence of a pointer where one is expected.</td>
</tr>
</tbody>
</table>

Address 0 is inaccessible, so *NULL is invalid; it crashes.

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>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

Heap:

Allocated block

Free block

Managed by memory allocator:

pointer to newly allocated block of at least that size

number of contiguous bytes required

```c
void* malloc(size_t size);

void free(void* ptr);
```
C: standard memory allocator

```c
#include <stdlib.h> // include C standard library

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
#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: Dynamic array allocation

```c
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?

---

[Diagram showing memory allocation and pointer operations]
// return a count of all zips that end with digit endNum
int zipCount(int* zips[], int endNum) {
}

Watch out! *cursor[4] means *(cursor[4])
MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?

I HATE YOU.

0x3A28213A
0x6339392C
0x7363682E.
C: `scanf` reads formatted input

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

Declared, but not initialized. Holds anything.

Read one int in decimal format from input.

Store it in memory at this address.

Store in memory at the address given by the address of val: store input @ 0x7F...F38.

val

CA  FE  12  34

0x7FFFFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFFFF38
0x7FFFFFFFFFFFFFFF34
C: Classic bug using `scanf`

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

Declared, but not initialized. Holds anything.

Store in memory at the address given by the **contents of `val`** (implicitly cast as a pointer):
store input @ 0xBAD4FACE.

Read one int in decimal_{10} format from input.

Store it in memory at this address.

---

**val**

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

| 0x7FFFFFFFFFFFFFFF3C |
| 0x7FFFFFFFFFFFFFFF38 |
| 0x7FFFFFFFFFFFFFFF34 |

<table>
<thead>
<tr>
<th>CA</th>
<th>FE</th>
<th>12</th>
<th>34</th>
</tr>
</thead>
</table>

| 0x00000000BAD4FACE |

---

**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!

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.
• Advances in programming language design since the 70’s have produced languages that fix C's problems while keeping strengths.