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
Solid-State Physics

Devices (transistors, etc.)

Digital Logic

Microarchitecture

Instruction Set Architecture

Operating System

Compiler/Interpreter

Programming Language

Program, Application

Software

Hardware
Instruction Set Architecture (HW/SW Interface)

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

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

- **Large storage**
  - Addresses, Locations

- **Computer**

**Diagram:**
- 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.

Alignment  (Why?)

Bit order within byte always same.

Byte ordering within larger value?
Endianness

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, ...

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

240 = F0

10 = 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 if values are pointers or not? How do we manage use of memory?

Programming with Memory
C: Variables are locations

Compiler maps variable name \(\rightarrow\) location.
Declarations do not initialize!

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

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;
```
C: Variables are locations

Compiler maps variable name \(\rightarrow\) 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: Variables are locations

Compiler maps variable name → location.

Declarations do not initialize!

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

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
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, represented as an address stored as data

Expressions using addresses and pointers:

```c
&___ address of the memory location representing ___
a.k.a. "reference to ___"
```

```c
*___ contents at the memory address given by ___
a.k.a. "dereference ___"
```

Pointer types:

```c
___* 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, $p$:

```c
int* p;
```

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

Declare two variables, `x` and `y`, that hold `int`s, and store 5 and 2 in them, respectively:

```c
int x = 5;
int y = 2;
```

Take the address of the memory location 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`:

```c
y = 1 + *p;
```

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

$\& = address\ of\ \star = contents\ at$
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;
```
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 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: \( \text{int} \ a[6]; \)

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

Indexing:  
```c
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.

Address Table:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>00</td>
</tr>
<tr>
<td>0x04</td>
<td>00</td>
</tr>
<tr>
<td>0x08</td>
<td>00</td>
</tr>
<tr>
<td>0x12</td>
<td>00</td>
</tr>
<tr>
<td>0x16</td>
<td>00</td>
</tr>
<tr>
<td>0x20</td>
<td>00</td>
</tr>
<tr>
<td>0x24</td>
<td>00</td>
</tr>
<tr>
<td>0x28</td>
<td>00</td>
</tr>
<tr>
<td>0x2C</td>
<td>00</td>
</tr>
<tr>
<td>0x30</td>
<td>00</td>
</tr>
</tbody>
</table>

a[0] = 0xf0
a[5] = a[0]
C: Arrays

Declaration: int a[6];

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

No bounds a[6] = 0xBAD;
check: a[-1] = 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.
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:
\[ \text{int* } p; \]
\[ p = a; \]
\[ p = \&a[0]; \]

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 \textit{immutable} pointer.
Address of \[ a[i] \] is base address \[ a \] plus \( i \) times element size in bytes.

\[
\begin{array}{c|c|c|c}
\hline
0x00 & 0x00 & 0x18 & 0x00 \\
0x00 & 0x00 & 0x1C & 0x00 \\
0x00 & 0x00 & 0x20 & 0x00 \\
0x00 & 0x00 & 0x24 & 0x00 \\
\hline
\end{array}
\]

\[ a[0] \]
\[ a[1] \]
\[ a[2] \]
\[ a[3] \]
\[ a[4] \]
\[ a[5] \]
\[ p \]
C: Arrays

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

Indexing:
\[ a[0] = 0\text{xf0}; \]
\[ a[5] = a[0]; \]

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

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

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

```
int a[6];
a[0] = 0xf0;
a[5] = a[0];
```

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

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

Indexing:  
\[
\text{a}[0] = 0xf0;
\]
\[
\text{a}[5] = \text{a}[0];
\]

No bounds check:  
\[
\text{a}[6] = 0x\text{BAD};
\]
\[
\text{a}[\text{-1}] = 0x\text{BAD};
\]

Pointers:  
\[
\text{int* } \textit{p};
\]
\[
\{ 
\quad \textit{p} = \textit{a};
\quad \textit{p} = \&\textit{a}[0];
\quad \&\textit{p} = 0xA;
\}
\]

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 \textit{a}[i] is base address \textit{a} plus \textit{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;
```

**Pointers:**

```c
int* p;

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

---

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: 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.

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

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:  
`int a[6];`

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

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

Pointers:  
```
int* p;
{
    p = a;
    p = &a[0];
    *p = 0xA;
}
```

`p[1] = 0xB;`

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.

`*p = a[1] + 1;`

Array indexing = address arithmetic
Both are scaled by the size of the type.
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 \texttt{sizeof} to determine proper size in C.

- \texttt{char string[12];}
- \texttt{int val[5];}
- \texttt{double a[3];}
- \texttt{char* p[3]; (or char *p[3];)}
C: Array access

Basic Principle

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

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{Expression} & \text{Type} & \text{Value} \\
\hline
\text{val[4]} & \text{int} & 1 \\
\text{val} & \text{int *} & \n \\
\text{val+1} & \text{int *} & \n \\
\&\text{val[2]} & \text{int *} & \n \\
\text{val[5]} & \text{int} & \n \\
\text{*(val+1)} & \text{int} & \n \\
\text{val + i} & \text{int *} & \n \\
\end{array}
\]
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

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

int strcmp(char* a, char* b);
# C: 0 vs. \0' vs. NULL

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

**NULL**

Name: null pointer / null reference / null address  
Type: void*  
Size: 1 word (= 8 bytes on a 64-bit architecture)  
Value: 0x0000000000000000  
Usage: The absence of a pointer where one is expected.  
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></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 permission
- **Contents**: Type of data
- **Managed by**: Entity responsible for managing the data
- **Initialized**: State of the data

- **RW**: Read/Write access
- **R**: Read-only access
- **X**: Execute-only access
C: Dynamic memory allocation in the heap

Heap:

Managed by memory allocator:

`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 error (no space), 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: 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[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;
    while (*zips) {
        if ((*zips)[4] == endNum) {
            count++;
        }
        zips++;
    }
    return count;
}
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 in decimal_{10} format from input.

Store it in memory at this address.

Store in memory at the address given by the address of val: store input @ 0x7FFFFFFF38.

```
val
CA FE 12 34
```

```
0x7FFFFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFFFF38
0x7FFFFFFFFFFFFF34
```
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.

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.

---

Read one int in decimal₁₀ format from input.

Store it in memory at this address.

```
val
  BA  D4  FA  CE
  ...  ...
  CA  FE  12  34
```

0x7FFFFFFFFFFFFF3C  0x7FFFFFFFFFFFFF38  0x7FFFFFFFFFFFFF34
0x00000000BAD4FACE
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.
- Many advances in programming language design since then have produced languages that fix C's problems while keeping strengths.