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

Why not just registers?
• Represent larger structures
• Computable addressing
• Indirection
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

Cell / location = 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)?

### 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, ...

---

**Address** | **Contents**
---|---
03 | 2A
02 | B6
01 | 00
00 | 0B

**Address** | **Contents**
---|---
03 | 0B
02 | 00
01 | B6
00 | 2A
Endianness in Machine Code

encodes: **add constant to register ebx**

Address: 8048366:

Contents: Instruction

81 c3 ab 12 00 00

Assembly Instruction

add $0x12ab,%ebx

encodes constant operand (0x000012ab) in little endian order

assembly version omits leading zeros
# Data, Addresses, and Pointers

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

**pointer** = address represented as data

---

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
<th>0x04</th>
<th>0x05</th>
<th>0x06</th>
<th>0x07</th>
<th>0x08</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<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>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</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 memory locations (for now)

Compiler maps variable → memory 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: Address and Pointer Primitives

*address* = index of a cell/location in memory
*pointer* = address represented as data

Expressions using addresses and pointers:

\&___  address of the memory location representing ___

*___  contents at the memory address given by ___

a.k.a. "dereference ___"

Pointer types:

___*  address of a memory location holding a ___
C: Address and Pointer Example

```c
int* p;

int x = 5;
int y = 2;

p = &x;

y = 1 + *p;
```
C: Address and Pointer Example

```c
int* p;

int x = 5;
int y = 2;

p = &x;

y = 1 + *p;
```

Declare a variable, `p` 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.

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`...

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

`&` = address of
`
`*` = contents at

---

Add 1 to the contents of memory at the address stored in `p`...
C: Address and Pointer Example

**C assignment:**

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
<td>0x03</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
</tr>
<tr>
<td>0x05</td>
<td>0x05</td>
</tr>
<tr>
<td>0x06</td>
<td>0x06</td>
</tr>
<tr>
<td>0x07</td>
<td>0x07</td>
</tr>
<tr>
<td>0x14</td>
<td>0x14</td>
</tr>
<tr>
<td>0x15</td>
<td>0x15</td>
</tr>
<tr>
<td>0x16</td>
<td>0x16</td>
</tr>
<tr>
<td>0x17</td>
<td>0x17</td>
</tr>
<tr>
<td>0x18</td>
<td>0x18</td>
</tr>
</tbody>
</table>

- 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;`

\& = address of
\* = contents at

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:

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

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

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

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

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

\[
\text{equivalent} \{ \\
\text{p}[1] = 0xB; \\
\text{*(p + 1) }= 0xB; \\
\text{p }= p + 2;
\}
\]

\textit{array indexing = address arithmetic}

Both are scaled by the size of the type.

\[
\text{*p }= a[1] + 1;
\]

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 \textit{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 \( N*\text{sizeof}(T) \) bytes of memory

Use `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];
```

<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 \textit{null} character.

![Hexadecimal representation of the string 'Harry Potter'](image)

Why?

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);

int string_length(char* str) {
    // Try with pointer arithmetic, but no array indexing.
NULL vs. '\0' vs. 0

The null character (i.e., '\0') is used to signify the end of a string of characters. It is one byte. Equal to 0b00000000.

There is also NULL. NULL is a pointer to an invalid memory address. NULL is often used to indicate the end of an array of pointers. Dereferencing a NULL pointer leads to crash! In x86-64 machines, NULL is 8 bytes (same as the size of a pointer), equivalent to 0x00000000.

0 is 0. It so happens that the null character and NULL are also literals of value 0. NULL and the null character do not need to be equivalent to 0 but are done so by convention.
### Memory 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>Stack</td>
<td>Procedure context</td>
<td>Compiler</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Heap</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Statics</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Literals</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Text</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
</tr>
</tbody>
</table>
C: Dynamic memory allocation in the heap

**Heap:**

Allocated block | Free block

**Managed by memory allocator:**

- `void* malloc(size_t size);`
  - Pointer to newly allocated block of at least that size

- `void free(void* ptr);`
  - Pointer to allocated block to free

- Number of contiguous bytes required
#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

zip

0x7fedd2400dc0 0x7fff58bdd938
0x7fedd2400dd0 0x7fedd2400dc0
0x7fedd2400dc8 0x7fedd2400dc4
0x7fedd2400dc0

0x7fedd2400dc0
0x7fedd2400dd0
0x7fedd2400dc0
0x7fedd2400dc8
0x7fedd2400dc4
0x7fedd2400dc0

0x7fedd2400dd0
0x7fedd2400dc0
0x7fedd2400dc8
0x7fedd2400dc4
0x7fedd2400dc0

0x7fedd2400dd0
0x7fedd2400dc0
0x7fedd2400dc8
0x7fedd2400dc4
0x7fedd2400dc0

0x7fedd2400dd0
0x7fedd2400dc0
0x7fedd2400dc8
0x7fedd2400dc4
0x7fedd2400dc0
C: Arrays of pointers to arrays of ...

```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;
```
Array of Zip Codes

Here we can use `NULL` to signify the end of an array of zip codes.

Why might it be important to end my array of pointers with `NULL`? What if we wanted a function to print out all the zip codes in the array but we didn’t know how long the array was?

What about ending zip codes with `NULL`? Does that make sense?
Zip Cycles

// return a count of all zips the end with digit endNum

int zipCount(int* zips[], int endNum) {

}
MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME A FEW POINTERS?

0x3A28213A
0x6339392C,
0x7363682E.

I HATE YOU.
C: scanf reads formatted input

```
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 0x7FFFFFFFFFFFFF38.
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>0x7FFFFFFFFFFFFFFF3C</th>
<th>0x7FFFFFFFFFFFFFFF38</th>
<th>0x7FFFFFFFFFFFFFFF34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BA D4 FA CE</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

|      | CA FE 12 34         | 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.