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)

- Instructions
  - Names, Encodings
  - Effects
  - Arguments, Results
- Local storage
  - Names, Size
  - How many
- Large storage
  - Addresses, Locations
- Processor
- Memory
- Instruction Logic
- Registers
- 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.
Byte ordering within larger value?

---

**Endianness**

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, ...

**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

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

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

At address 0x20, 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?

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
// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;
// 1. load the contents @ 0x0C
// 2. add 3
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>long</td>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</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

```c
int* p;
int x = 5;
int y = 2;
p = &x;
y = 1 + *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`.

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`.

C: 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.

Take the address of the memory representing `x`... and store it in the memory location representing `p`.

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`.

C: Pointer example

```c
int x = 5;
int y = 2;
p = &x;
y = 1 + *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`.

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:

```
int* ptr;
I see: "The variable ptr holds an address of an int in memory."
int * ptr;
int *ptr;  // more common C style
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;
int* a; int b;
```

**C: Arrays**

Declaration: `int a[6];`

- `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.
- Indexing: `a[0] = 0xf0;`
- 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 name for the array's base address, can be used as an immutable pointer.

Indexing: 

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

Declaration: 

\[ a[6]; \]

\[ a[0] = 0xf0; \]

\[ a[5] = a[0]; \]

No bounds check:

\[ a[6] = 0xBAD; \]

\[ a[-1] = 0xBAD; \]

Pointers:

\[ int* p; \]

\[ p = a; \]

\[ p = &a[0]; \]
C: Arrays

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;`

- **Example:**
  - `00 00 0B AD 0x24 a[5]
  - `00 00 00 FG 0x20 a[0]
  - `00 00 0C 0x04 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.

- **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;`

- **Example:**
  - `00 00 0B AD 0x24 a[5]
  - `00 00 00 FG 0x20 a[0]
  - `00 00 0C 0x04 P + 1 = 0xB;`

- **Array indexing = address arithmetic**
  - Both are scaled by the size of the type.
C: Arrays

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.

Declaration: int a[6];
Indexing: a[0] = 0xf0;
a[5] = a[0];
No bounds check: a[6] = 0xBAD;
Pointers: int* p;
equivalent { p[1] = 0xB;
  *(p + 1) = 0xB;
  p = p + 2;
}
array indexing = address arithmetic
Both are scaled by the size of the type.

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

Start this presentation to see live content. For more info about software, share the entire session. Get help at palcom.com/app.

C: Array allocation

Basic Principle

T A[N]; Array of length N with elements of type T and name A
Contiguous block of N*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

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

<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>

C: Null-terminated strings

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

<table>
<thead>
<tr>
<th>( 0x57 0x65 0x6C 0x6C 0x65 0x73 0x6C 0x65 0x79 0x20 0x43 0x53 0x00 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why?</td>
</tr>
</tbody>
</table>

Does Endianness matter for strings?

int string_length(char str[]) {

}

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: * and [ ]

C programmers often use * where you might expect []:

- 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>Value</th>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>int</td>
<td>4 bytes</td>
<td>The integer zero.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>null character</td>
<td>char</td>
<td>1 byte</td>
<td>Terminator for C strings.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>null pointer / null reference / null address</td>
<td>void*</td>
<td>1 word (= 8 bytes on a 64-bit architecture)</td>
<td>The absence of a pointer where one is expected. Address 0 is inaccessible, so *NULL is invalid; it crashes.</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>2^4-1</td>
<td>RW</td>
<td>Stack</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>0</td>
<td>RW</td>
<td>Heap</td>
<td>Programmer, malloc/free, new/ GC</td>
<td>Run time</td>
</tr>
<tr>
<td>X</td>
<td>R</td>
<td>Literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>Text</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

C: Dynamic memory allocation in the heap

Heap:

Managed by memory allocator:

- pointer to newly allocated block of at least that size
- number of contiguous bytes required
- void* malloc(size_t size);
- void free(void* ptr);
- pointer to allocated block to free

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) {
    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]);
}
free(zip);
```

```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;
```

```c
int zipCount(int* zips[], int endNum) {
    int count = 0;
    while (*zips) {
        if (*zips[4] == endNum) {
            count++;
        }
        zips++;
    }
    return count;
}
```
C: scanf reads formatted input

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

val
CA FE 12 34
```

C: Classic bug using scanf

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

val
CA FE 12 34
```

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!

![xkcd comic](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.