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

pointers and arrays in C
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

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, ...

Multi-byte values in memory
Store across contiguous byte locations.

Alignment (Why?)

Bit order within byte always same. Byte ordering within larger value?

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
C: Variables are locations

Compiler maps variable name → location.

Declarations do not initialize!

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

```c
x = 0; // store 0 at 0x20
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

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`. 
C: Pointer example

C assignment:

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x24</td>
<td>y</td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td>x</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>p</td>
</tr>
<tr>
<td>0x00</td>
<td></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)
*p = 240;

What is the type of *p?
What is &x?
What is *(&y)?

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;

I see: "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.

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

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

Pointers:

int* p;

<table>
<thead>
<tr>
<th>Equivalence</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = a;</td>
<td></td>
</tr>
<tr>
<td>p = &amp;a[0];</td>
<td></td>
</tr>
<tr>
<td>*p = 0xh;</td>
<td></td>
</tr>
</tbody>
</table>

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.

*p = a[1] + 1;
C: Array allocation

Basic Principle

\[ A[N] \]

Array of length \( N \) with elements of type \( T \) and name \( A \)

Contiguous block of \( N \times \text{sizeof}(T) \) bytes of memory

Use \text{sizeof} to determine proper size in C.

C: Array access

Basic Principle

\[ A[N] \]

Array of length \( N \) with elements of type \( T \) and name \( A \)

Identifier \( A \) has type

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{val} )</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>( &amp;\text{val}[2] )</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>( \text{val} + 1 )</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>( \text{val} + \text{i} )</td>
<td>int *</td>
<td></td>
</tr>
</tbody>
</table>

C: Null-terminated strings

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

Does Endianness matter for strings?

\[
\text{int string\_length(char str[])}
\]

\[
\}
\]
C: 0 vs. '\0' vs. NULL

| NULL | Name: null pointer / null reference / null address  
| Type: void* | Size: 1 word (= 8 bytes on a 64-bit architecture)  
| Value: 0x00000000  
| 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?

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);

C: Dynamic array allocation

```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%0s\n", zip);
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?

---

**Zip code**

```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);
```

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.

---

http://xkcd.com/138/
C: Classic bug using `scanf`

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

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.

```
val
BA D4 FA CE
... 0x7FFFFFFF F3C
0x7FFFFFFF F38
0x7FFFFFFF F34
... 0x00000000 BAD4FACE
```

Declared, but not initialized – holds anything.

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.

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!

[C: Memory error messages](http://xkcd.com/371/)

Programming with Memory