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)

Computer

Local storage
- Names, Size
- How many

Large storage
- Addresses, Locations

Encoded Instructions

Data

Registers

Instruction Logic

Instructions
- Names, Encodings
- Effects
- Arguments, Results

Microarchitecture

Digital Logic

Devices (transistors, etc.)

Solid-State Physics

Operating System

Compiler/Interpreter

Programming Language

Program, Application

Hardware

Software
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?

**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
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: Pointer operations and types

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

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: Types determine sizes

Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
</tr>
<tr>
<td>char</td>
<td>short int</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>long double</td>
<td>(pointer) *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>32-bit word</th>
<th>64-bit word</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

address size = word size

C: Pointer example

Declare a variable, p

int* p;

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

int x = 5;
int y = 2;

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

p = &x;

Take the address of the memory location representing x

... and store it in the memory location representing p.

Now, “p points to x.”

y = 1 + *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 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 0x0F as contents @ 0x14
*p = 240;

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;

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; means int* a;
int b;
Arrays are adjacent memory locations storing the same type of data.

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

**Array Element:**
- `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.
- Address of `a[i]` is base address `a` plus `i` times element size in bytes.

**Examples:**
1. **Integers:**
   ```c
   a[0] = 0xf0;
   a[5] = a[0];
   ```
2. **Hexadecimal:**
   ```c
   a[5] = 0xBAD;
   ```

---

**C: Arrays**

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

**Indexing:**
- `a[0] = 0xf0;`
- `a[5] = a[0];`

**Address Calculation:**
- Address of `a[i]` is base address `a` plus `i` times element size in bytes.

---

**C: Arrays**

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

**Indexing:**
- `a[0] = 0xf0;`
- `a[5] = a[0];`

**Address Calculation:**
- Address of `a[i]` is base address `a` plus `i` times element size in bytes.

---

**C: Arrays**

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

**Indexing:**
- `a[0] = 0xf0;`
- `a[5] = a[0];`

**Address Calculation:**
- Address of `a[i]` is base address `a` plus `i` times element size in bytes.

---

**C: Arrays**

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

**Indexing:**
- `a[0] = 0xf0;`
- `a[5] = a[0];`

**Address Calculation:**
- Address of `a[i]` is base address `a` plus `i` times element size in bytes.

**No bounds check:**
- `a[6] = 0xBAD;`
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] = 0xBBAD;`  
  `a[-1] = 0xBBAD;`
- **Address of** `a[i]` **is base address** `a`  
  `+`  
  `i` **times element size in bytes.**

- **Pointers:** `int *p;`  
  `p = a;`  
  `p = &a[0];`  
  `*p = 0xA;`
C: Arrays

Declaration: int a[6];

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

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

Pointers: int* p;

equivalent { 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.

Both are scaled by the size of the type.

array indexing = address arithmetic
Both are scaled by the size of the type.
C: Array allocation

Basic Principle

Array of length N with elements of type T and name A

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

char string[12];

\begin{tabular}{cccc}
  x & x+1 & x+2 & x+3 \\
\end{tabular}

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

int val[5];

\begin{tabular}{cccccc}
  x & x+4 & x+8 & x+12 & x+16 & x+20 \\
\end{tabular}

double a[3];

\begin{tabular}{cccc}
  x & x+8 & x+16 & x+24 \\
\end{tabular}

char* p[3];

\begin{tabular}{cccc}
  x & x+8 & x+16 & x+24 \\
\end{tabular}

(or char *p[3];)

\begin{tabular}{cccc}
  x & x+8 & x+12 & x+24 \\
\end{tabular}

C: Array access

Basic Principle

Array of length N with elements of type T and name A

Identifier A has type T*

int val[5];

\begin{tabular}{cccccc}
  0 & 2 & 4 & 8 & 1 \\
\end{tabular}

Expression Type Value
val[4] int 1
val int *
val+1 int *
&val[2] int *
val[5] int
*(val+1) int
val + i int *

C: Null-terminated strings

C strings: arrays of ASCII characters ending with \textit{null character}.

\begin{tabular}{ccccccccccccccc}
  0x7F & 0x65 & & 0x6C & & 0x65 & & 0x73 & & 0x6C & & 0x65 & & 0x79 & & 0x20 & & 0x43 & & 0x53 & & 0x00 \\
\end{tabular}

'W' 'e' 'l' 'l' 'e' 'y' 'C' 'S' '

Does Endianness matter for strings?

int string_length(char str[]) {

}
C: * and []

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

- *char*: pointer to a char
- *char*: pointer to the first char in a string of unknown length

```c
int strcmp(char* a, char* b);
```

C: 0 vs. \0 vs. NULL

<table>
<thead>
<tr>
<th>0</th>
<th>' \0 '</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: zero</td>
<td>Name: null character</td>
<td></td>
</tr>
<tr>
<td>Type: int</td>
<td>Type: char</td>
<td></td>
</tr>
<tr>
<td>Size: 4 bytes</td>
<td>Size: 1 byte</td>
<td></td>
</tr>
<tr>
<td>Value: 0x00000000</td>
<td>Value: 0x00</td>
<td></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^N-1</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Global variables/ static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></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:

- void* malloc(size_t size);
- void free(void* ptr);
C: standard memory allocator

#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: Dynamic array allocation

#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]);
}
printf("\n");
free(zip);

C: Array of pointers to arrays of ints

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;

// 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;
}

Zip code

// 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;
}
C: `scanf` reads formatted input

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

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 @ 0x7FFFFFFF38.

Best case: crash immediately with segmentation fault/bus error.

Worst case: program does literally anything.

C: Classic bug using `scanf`

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

Read one int in decimal format from input.

Store it in memory at this address.

Store input @ 0xBAD4FACE.

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.