Programming with Memory

the memory model
pointers and arrays in C
Devices (transistors, etc.)
Solid-State Physics

Hardware

Software

Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)

Operating System
Compiler/Interpreter
Programming Language
Program, Application

Programming with Memory
Instruction Set Architecture (HW/SW Interface)

**processor**
- Instruction Logic
- Registers

**memory**
- Encoded Instructions
- Data

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

**Large storage**
- Addresses, Locations

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

Computer

Programming with Memory
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?

### LittleEndian: least significant byte first
- low order byte at low address
- high order byte at high address
- used by x86, ...

### BigEndian: 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
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 (decimal) = F0 (hexadecimal) = \texttt{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?

Memory drawn as 32-bit values, little endian order.
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

\textit{address} = index of a location in memory

\textit{pointer} = a reference to a location in memory, represented as an address stored as data

Expressions using addresses and pointers:
\begin{itemize}
  \item \texttt{&\_\_\_} \texttt{address of} the memory location representing \_\_\_
    a.k.a. "reference to \_\_\_"
  \item \texttt{\_*\_\_\_} \texttt{contents at} the memory address given by \_\_\_
    a.k.a. "dereference \_\_\_"
\end{itemize}

Pointer types:
\begin{itemize}
  \item \_\_\_\_\_*\_\_\_\_\_ address of a memory location holding a \_\_\_
    a.k.a. "a reference to a \_\_\_"
\end{itemize}
C: Pointer example

Declaring a pointer variable `p`:

```c
int* p;
```

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

Declaring two variables, `x` and `y`, that hold `ints`, and store 5 and 2 in them, respectively:

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

Take the address of the memory location representing `x`:

```c
p = &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
  * `*` = 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;

& = address of
* = contents at
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; means int* a; int b;
```
C: Arrays

Declaration: `int a[6];`

- **element type**: `int`
- **name**: `a`
- **number of elements**: 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:  
```c
int a[6];
```

Indexing:  
```c
a[0] = 0xf0;
```
```c
a[5] = a[0];
```
```c
a[6] = 0xBAD;
```
```c
a[-1] = 0xBAD;
```

No bounds check:  
```c
int* p;
```
```c
p = a;
```
```c
p = &a[0];
```
```c
*p = 0xA;
```
```c
p[1] = 0xB;
```
```c
*(p + 1) = 0xB;
```
```c
p = p + 2;
```
```c
*p = a[1] + 1;
```

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.

`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

- **char** string[12];
- **int** val[5];
- **double** a[3];
- **char*** p[3];

Use \texttt{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 \( T^* \)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{val}[4] )</td>
<td>( \text{int} )</td>
<td>1</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>( \text{int *} )</td>
<td></td>
</tr>
<tr>
<td>( \text{val}+1 )</td>
<td>( \text{int *} )</td>
<td></td>
</tr>
<tr>
<td>&amp;( \text{val}[2] )</td>
<td>( \text{int *} )</td>
<td></td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>( \text{int} )</td>
<td></td>
</tr>
<tr>
<td>*(( \text{val}+1 ))</td>
<td>( \text{int} )</td>
<td></td>
</tr>
<tr>
<td>( \text{val} + i )</td>
<td>( \text{int *} )</td>
<td></td>
</tr>
</tbody>
</table>
C: Null-terminated strings

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

```
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

```c
int strcmp(char* a, char* b);
int string_length(char* str) {
  // Try with pointer arithmetic, but no array indexing.
}
```
**C: 0 vs. ' \0 ' vs. NULL**

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>zero</td>
<td>int</td>
<td>4 bytes</td>
<td>0x00000000</td>
<td>The integer zero.</td>
</tr>
<tr>
<td><strong>' \0 '</strong></td>
<td>null character</td>
<td>char</td>
<td>1 byte</td>
<td>0x00</td>
<td>Terminator for C strings.</td>
</tr>
<tr>
<td><strong>NULL</strong></td>
<td>null pointer / null reference / null address</td>
<td>void*</td>
<td>1 word (= 8 bytes on a 64-bit architecture)</td>
<td>0x0000000000000000</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^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>

- **Stack**: Read-Write (RW) Procedure context managed by Compiler, initialized at Run time.
- **Heap**: Read-Write (RW) Dynamic data structures managed by Programmer, malloc/free, new/GC, initialized at Run time.
- **Statics**: Read-Write (RW) Global variables/static data structures managed by Compiler/Assembler/Linker, initialized at Startup.
- **Literals**: Read (R) String literals managed by Compiler/Assembler/Linker, initialized at Startup.
- **Text**: X Instructions managed by Compiler/Assembler/Linker, initialized at Startup.
C: Dynamic memory allocation in the heap

Heap:

Allocated block

Free block

Managed by memory allocator:

\texttt{void* malloc(size_t size);}  \hspace{1cm} \texttt{number of contiguous bytes required}

\texttt{void free(void* ptr);}  \hspace{1cm} \texttt{pointer to allocated block to free}
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.
#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;
zoom[2] = 4;
zoom[3] = 8;
zoom[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

```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

// return a count of all zips that end with digit endNum
int zipCount(int* zips[], int endNum) {

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

I HATE YOU.

http://xkcd.com/138/
C: `scanf` reads formatted input

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

Declared, but not initialized.
Holds anything.

Store in memory at the address given by the address of `val`:
store input @ 0x7F...F38.

Read one int in decimal₁₀ format from input.

Store it in memory at this address.

```
0x7FFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFF38
0x7FFFFFFFFFFFFF34
```

`val`

BA D4 FA CE
C: Classic bug using `scanf`

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

- Declared, but not initialized. Holds anything.
- Read one int in decimal₁₀ format from input.
- Store it in memory at this address.
- 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.
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.