Programming with Memory

the memory model
pointers and arrays in C

https://cs.wellesley.edu/~cs240/
Instruction Set Architecture (HW/SW Interface)

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

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

- **Large storage**
  - Addresses, Locations

- **Processor**
  - Instruction Logic
  - Registers

- **Memory**
  - Encoded Instructions
  - Data

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?

**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 Table]

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>2A</td>
</tr>
<tr>
<td>02</td>
<td>B6</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0B</td>
</tr>
</tbody>
</table>

![Address Contents Table]

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>0B</td>
</tr>
<tr>
<td>02</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>B6</td>
</tr>
<tr>
<td>00</td>
<td>2A</td>
</tr>
</tbody>
</table>
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 \(= 10_{10} = F0_{16} = 0x00 00 00 F0\)

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?
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

**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

```
int* 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.

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

Take the address of the memory location representing x

```
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

```
y = 1 + *p;
```

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

& = address of
* = contents at

Programming with Memory   21
C: Pointer example

C assignment:  \[ \text{Left-hand-side} = \text{right-hand-side}; \]

```c
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;

*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**
- **name**
- **number of elements**

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.

```
+0  +1  +2  +3
+---------------
| 0x00 0x04 |   |
| 0x08 0x0C |   |
| 0x10 0x14 |   |
| 0x18 0x1C |   |
| 0x20 0x24 |   |
+---------------
```
C: Arrays

Declaration: \[\text{int a[6];}\]

Indexing:
\[\text{a[0] = 0xf0; a[5] = a[0];}\]
\[\text{a[6] = 0xBAD; a[-1] = 0xBAD;}\]

No bounds check:
\[\text{int}\ast p; p = a; p = \&a[0]; \ast p = 0xA; p[1] = 0xB; (p + 1) = 0xB; p = p + 2; \ast p = a[1] + 1;\]

Arrays are adjacent memory locations storing the same type of data.
\[\text{a is a name for the array’s base address, can be used as an } \text{immutable } \text{pointer.}\]
Address of \[\text{a[i]}\] is base address \[\text{a}\] plus \(i\) times element size in bytes.

\text{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*sizeof(T) bytes of memory
```

- **char string[12];**
  - Use `sizeof` to determine proper size in C.
- **int val[5];**
  - `x + 4`
  - `x + 8`
  - `x + 12`
  - `x + 16`
  - `x + 20`
- **double a[3];**
  - `x + 8`
  - `x + 16`
  - `x + 24`
- **char* p[3];**
  - (or `char *p[3];`)
  - `x + 4`
  - `x + 8`
  - `x + 12`

IA32

x86-64
C: Array access

Basic Principle

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

\[
\begin{array}{c}
\text{Expression} & \text{Type} & \text{Value} \\
\text{val[4]} & \text{int} & 1 \\
\text{val} & \text{int *} & \text{---} \\
\text{val+1} & \text{int *} & \text{---} \\
& \text{int *} & \text{---} \\
\text{val[5]} & \text{int} & \text{---} \\
\text{*(val+1)} & \text{int} & \text{---} \\
\text{val + i} & \text{int *} & \text{---}
\end{array}
\]
C: Null-terminated strings

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

Why?

```
0x48 0x61 0x72 0x72 0x79 0x20 0x50 0x6F 0x74 0x74 0x65 0x72 0x00
'H' 'a' 'r' 'r' 'y' ' ' 'P' 'o' 't' 't' 'e' 'r' '\0'
```

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

```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

### 0
- **Name:** zero
- **Type:** int
- **Size:** 4 bytes
- **Value:** 0x00000000
- **Usage:** The integer zero.

### '\0'
- **Name:** null character
- **Type:** char
- **Size:** 1 byte
- **Value:** 0x00
- **Usage:** Terminator for C strings.

### NULL
- **Name:** null pointer / null reference / null address
- **Type:** void*
- **Size:** 1 word (= 8 bytes on a 64-bit architecture)
- **Value:** 0x0000000000000000
- **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?
## 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>Heap</td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run time</td>
</tr>
<tr>
<td>Statics</td>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Literals</td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>Text</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);`

-Pointer to newly allocated block of at least that size
-Number of contiguous bytes required
-Person to allocated block to free

Programming with Memory 51
#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

```c
#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("
");
free(zip);
```

```plaintext
zip[0] = 0;  0x7fedd2400dc0  0x7fff58bdd938
zip[1] = 2;  1
zip[2] = 4;  8  0x7fedd2400dd0
zip[3] = 8;  0x7fedd2400dc4
zip[4] = 1;
```

Programming with Memory 53
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?
// 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.

0x3A28213A
0x6339392C,
0x7363682E.

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

```c
int val;

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

Declared, but not initialized. Holds anything.

Read one int in decimal10 format from input.

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

<table>
<thead>
<tr>
<th>val</th>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x7FFFFFFF</td>
<td>0x7FFFFFFF</td>
<td>0x7FFFFFFF</td>
<td>0x7FFFFFFF</td>
</tr>
</tbody>
</table>

Store it in memory at this address.
C: Classic bug using `scanf`

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

Declared, but not initialized.
Holds anything.

Read one int in decimal $_{10}$ 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.

<table>
<thead>
<tr>
<th>val</th>
<th>0x7FFFFFFFFFFFFFFF3C</th>
<th>0x7FFFFFFFFFFFFFFF38</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA D4</td>
<td>0x00000000BAD4FACE</td>
<td></td>
</tr>
<tr>
<td>FA CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0x7FFFFFFFFFFFFFFF34</td>
<td></td>
</tr>
<tr>
<td>CA FE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.