Programming with Memory

the memory model
pointers and arrays in C

Instruction Set Architecture (HW/SW Interface)

Computer

locally

Encoded
Instructions

Registers

Instruction
Logic

memory

Large storage

• Addresses, Locations

Local storage

• Names, Size
• How many

Instructions

• Names, Encodings
• Effects
• Arguments, Results

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

Program, Application

Programing Language

Compiler/Interpreter

Operating System

Instruction Set Architecture

Microarchitecture

Digital Logic

Devices (transistors, etc.)

Solid-State Physics
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!

```
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

**C assignment:**

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

```
int* p;  // p @ 0x04
int x = 5;  // x @ 0x14, store x @ 0x14
int y = 2;  // y @ 0x24, store y @ 0x24
p = &x;  // store x @ 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;
```

```
0x00 0x04 0x08 0x10 0x14 0x18 0x20 0x24
   x
   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;
```

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:**
```c
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.

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.

**Addressing:**
- `a[0]`: 0x24
- `a[1]`: 0x20
- `a[2]`: 0x1C
- `a[3]`: 0x18
- `a[4]`: 0x14
- `a[5]`: 0x10
- `a[6]`: 0x0C
- `a[7]`: 0x08
- `a[8]`: 0x04
- `a[9]`: 0x00

**Array indexing = address arithmetic**
Both are scaled by the size of the type.

```
*p = a[1] + 1;
```

---

### 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.

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

**Expression**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int *</td>
<td>int *</td>
<td>int *</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>

### 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*`

Use `sizeof` to determine proper size in C.

<table>
<thead>
<tr>
<th><code>int val[5];</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
</tr>
<tr>
<td><code>x + 4</code></td>
</tr>
<tr>
<td><code>x + 8</code></td>
</tr>
<tr>
<td><code>x + 12</code></td>
</tr>
<tr>
<td><code>x + 16</code></td>
</tr>
<tr>
<td><code>x + 20</code></td>
</tr>
</tbody>
</table>

**Expression**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int *</td>
<td>int *</td>
<td>int *</td>
<td>int</td>
<td>int</td>
</tr>
</tbody>
</table>
C: Null-terminated strings

C strings: arrays of ASCII characters ending with *null character*. 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: 0x00000000000000
  - 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>0</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run time</td>
</tr>
<tr>
<td>2^{N-1}</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

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.

#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;
zip0[1] = 2;
zip0[2] = 4;
zip0[3] = 8;
zip0[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);
```

`scanf` reads one `int` in decimal format from input and stores it in memory at the address given by the address of `val`. Best case: crash immediately with segmentation fault/bus error. Bad case: silently corrupt data stored at an arbitrary address. Worst case: program does literally anything.

C: Classic bug using `scanf`

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

`scanf` reads one `int` in decimal format from input and stores it in memory at the address given by the contents of `val` (implicitly cast as a pointer): store input at an arbitrary address. Best case: crash immediately with segmentation fault/bus error. Bad case: silently corrupt data stored at an arbitrary address, 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.