Programming with Memory

the memory model
pointers and arrays in C

Instruction Set Architecture (HW/SW Interface)

Computer

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 → 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

```
int* p;

int x = 5;
int y = 2;

p = &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 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.

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[1] is base address a plus i times element size in bytes.

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

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

Use \text{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*

Expression Type Value

\[ \text{val[4]} \quad \text{int} \quad 1 \]
\[ \text{val} \quad \text{int } * \]
\[ \text{val+1} \quad \text{int } * \]
\[ \&\text{val}[2] \quad \text{int } * \]
\[ \text{val[5]} \quad \text{int} \]
\[ *(\text{val+1}) \quad \text{int} \]
\[ \text{val + i} \quad \text{int } * \]
C strings: arrays of ASCII characters ending with null character.

Does Endianness matter for strings?

```c
int string_length(char str[]) {
}
```

C: Null-terminated strings

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>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>R</td>
<td>RW</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>X</td>
<td>RW</td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

Stack
Heap
Statics
Literals
Text

0
C: Dynamic memory allocation in the heap

Heap:

|   |   |   |   |   |   |   |   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Free block
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Allocated block

Managed by memory allocator:

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

```c
#include <stdlib.h>

void* malloc(size_t size);

void free(void* ptr);
```

C: Array of pointers to arrays of ints

```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");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
printf("n");
free(zip);
```

C: standard memory allocator

```c
#include <stdlib.h>

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) {
    perror("malloc");
    exit(0);
}
```

```c
printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
printf("n");
free(zip);
```
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 in decimal format from input.**
- **Store it in memory at this address.**

```
0x00000000
0x10004380
0x10008900
```

- **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: Classic bug using `scanf`

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

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

```
0x00000000BAD4FACE
0x7FFFFFFFFFFFFF3C
0x7FFFFFFFFFFFFF38
```

- **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.