Programming with Memory
via C, pointers, and arrays

Why not just registers?
- Represent larger structures
- Computable addressing
- Indirection

byte-addressable memory = mutable byte array

Cell / location = element
- Addressed by unique numerical address
- Holds one byte
- Readable and writable

Address = index
- Unsigned number
- Represented by one word
- Computable and storable as a value

multi-byte values in memory
Store across contiguous byte locations.

Alignment (Why?)

Bit order within byte always same.
Byte ordering within larger value?

Endianness: To store a multi-byte value in memory, which byte is stored first (at a lower address)?

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, ...
Data, Addresses, and Pointers

address = index of a cell in memory
pointer = address represented as data

---

C: variables are memory locations
(for now)

Compiler maps variable \(\rightarrow\) memory location. Declarations do not initialize.

int \(x;\) // \(x\) at 0x20
int \(y;\) // \(y\) at 0x0C

\(x = 0;\) // store 0 at 0x20

// store 0x3CD02700 at 0x0C
\(y = 0x3CD02700;\)

// load the contents at 0x0C,
// add 3, and store sum at 0x20
\(x = y + 3;\)

---

C: Address and Pointer Primitives

address = index of a cell/location in memory
pointer = address represented as data

Expressions using addresses and pointers:

\&\(\_\_\\) address of the memory location representing \(\_\_\_

\_*\_\_\_\_* contents at the memory address given by \_\_\_
\ מישהו "dereference \_\_\_"

Pointer types:

\_\_\_* address of a memory location holding a \_\_\_

---

C: Address and Pointer Example

\& = address of
\_* = contents at

Declare a variable, \(p;\)

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.

Get the address of the memory location

\(p = \&x;\) representing \(x\)
... and store it in \(p\). Now, "\(p\) points to \(x\)."

Add 1 to the contents of memory at the address stored in \(p\)

\(y = i + \*p;\)
... and store it in the memory location representing \(y\).
C: Address and Pointer Example

C assignment:

\[ \& = \text{address of} \]
\[ * = \text{contents at} \]

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

- `int* p;` // p: 0x04
- `int x = 5;` // x: 0x14, store 5 at 0x14
- `int y = 2;` // y: 0x24, store 2 at 0x24
- `p = \&x;` // store 0x14 at 0x04
- // load the contents at 0x04 (0x14)
- // load the contents at 0x14 (0x5)
- // add 1 and store sum at 0x24
- `y = 1 + *p;`
- // load the contents at 0x04 (0x14)
- // store 0xFO (240) at 0x14
- `*p = 240;`

C: Arrays

Declaration:

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

Element type

- `int` is a name for the array’s base address, can be used as an immutable pointer.

Number of elements

Arrays are adjacent memory locations storing the same type of data.

Name

Arrays are adjacent memory locations storing the same type of data.

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

C: Array Allocation

Use `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 \( A \) can be used as a pointer to array element 0: \( A \) has type \( T^* \)

```
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>val+i</td>
<td>int *</td>
<td></td>
</tr>
</tbody>
</table>

C: Null-terminated strings

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

Does Endianness matter for strings?

```
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

```
int strcmp(char* a, char* b);
int string_length(char* str) {
  // Try with pointer arithmetic, but no array indexing.
}
```

Memory Layout

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

Managed by memory allocator:

- `malloc(size_t size)`: Returns a pointer to a block of memory of size `size` bytes.
- `free(void* ptr)`: Frees the memory pointed to by `ptr`.

C: standard memory allocator

```c
#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* p)
    
    // Deallocates the block at p, making its space available for new allocations.
    // p must be a malloc result that has not yet been freed.
    Rules:
    - p must be a malloc result that has not yet been freed.
    - Do not use *p 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("\n");
free(zip);
```

C: Arrays of pointers to arrays of...

```c
int** zips = (int**)malloc(sizeof(int*)*3);
...

for (int i = 0; i < ZIP_LENGTH; i++)
{
    int zips[i] = (int*)malloc(sizeof(int*)*5);
    ...
}
```
C: Memory-Related Perils and Pitfalls

Dereferencing bad pointers

See later exercises for:
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks

C: scanf reads formatted input

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

Declared, but not initialized
- holds anything.

Read one int from input.
Store it in memory at this address.

i.e., store it in memory at the address where the contents of val is stored:
store into memory at 0x7FFFFFFF38.

val

<table>
<thead>
<tr>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF3C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7FFFFFFF38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7FFFFFFF34</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C: classic bug using scanf

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

Declared, but not initialized
- holds anything.

Read one int from input.
Store it in memory at this address.

i.e., store it in memory at the address given by the contents of val:
store into memory at 0xBAD4FACE.

val

<table>
<thead>
<tr>
<th>BA</th>
<th>D4</th>
<th>FA</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF3C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7FFFFFFF38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x7FFFFFFF34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>FE</td>
<td>12</td>
<td>34</td>
</tr>
<tr>
<td>0x00000000BAD4FACE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C: memory error messages

Okay, human.
Huh?
Before you hit compile, listen up!

You know when you're falling asleep, and you imagine yourself walking or something.
And suddenly you misstep, stumble, and jolt awake?

Yeah!
Double-check your damn pointers, okay?

11: segmentation fault ("segfault", SIGSEGV)
accessing address outside legal area of memory

Best case: segmentation fault, or bus error, crash.

Bad case: silently corrupt data stored at address 0xBAD4FACE, and val still holds 0xBAD4FACE.
Worst case: arbitrary corruption

10: bus error
accessing misaligned or other problematic address

More to come on debugging!