Programming with Memory via C, pointers, and arrays

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

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

<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>

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 cell in memory
*pointer* = address represented as data

<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>

memory drawn as 32-bit values, little endian order
**C: variables are memory locations**  (for now)

Compiler maps variable → memory location.
Declarations do not initialize!

```c
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 ___  
  a.k.a. "dereference ___"

Pointer types:
- `___*` address of a memory location holding a ___

---

**C: Address and Pointer Example**

Declare a variable, `p` that will hold the address of a memory location holding an int

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

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

```
Get
the address of the memory location representing x
p = &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 = 1 + *p;
... and store it in the memory location representing y.
```

---

**C assignment:**

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

```c
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 0xF0 (240) at 0x14
*p = 240;
```
C: Arrays

Declaration:  
```c
int a[6];
```

Indexing:  
```c
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:  
```c
a[6] = 0xBAD;
```

Pointers:  
```c
int* p;
p = a;
p = &a[0];
*p = 0xA;
p[1] = 0xB;
(*p + 1) = 0xB;
p = p + 2;
```

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

C: Array Allocation

Basic Principle
```c
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.

C: Array Access

Basic Principle
```c
T  A[N];
```

Array of length `N` with elements of type `T` and name `A`

Identifier `A` has type

<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.

Why?

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

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

C: Dynamic memory allocation in the heap

Heap:

Managed by memory allocator:

pointer to newly allocated block of at least that size
number of contiguous bytes required

void* malloc(size_t size);

pointer to allocated block to free

void free(void* ptr);

Memory Layout

Addr       Perm  Contents                  Managed by      Initialized
2^N-1       Stack  Procedure context      Compiler        Run time
           Heap   Dynamic data structures     Programmer, malloc/free, new/GC Run time
RW         Statics Global variables/     Compiler/Assembler/Linker Startup
            Static data structures         
RW         Literals String literals      Compiler/Assembler/Linker Startup
R          Text    Instructions          Compiler/Assembler/Linker Startup
X

Zip Cycles

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

}
C: scanf reads formatted input

```c
int val; // Declared, but not initialized – holds anything.
...
scanf("%d", &val);
```

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 0xFFFFFF38.

```
val
BA D4 FA CE
0x7FFFFFFFFFFFF3C
0x7FFFFFFFFFFFF38
0x7FFFFFFFFFFFF34
```

C: classic bug using scanf

```c
int val; // Declared, but not initialized – holds anything.
...
scanf("%d", val);
```

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
BA D4 FA CE
0x7FFFFFFFFFFFF3C
0x7FFFFFFFFFFFF38
0x7FFFFFFFFFFFF34
```

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...