Programming with Memory
via C, pointers, and arrays

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

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 → 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 ___
  a.k.a. "dereference ____"

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

### C: Address and Pointer Example

**C assignment:**
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 0x14 (0x14)
  // 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: `int a[6];`

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

No bounds check:
- `a[6]` = 0xBAD;
- `a[-1]` = 0xBAD;

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

**array indexing = address arithmetic**
Both are scaled by the size of the type.
- *`p` = `a[1]` + 1;

### C: Arrays Example

`int* p;`

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

- `p` = \&`x`;
- `y` = 1 + *`p`;

**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 1 times element size in bytes.
**C: Array Access**

**Basic Principle**

\[ T \ A[N]; \]

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>( \text{val}[4] )</td>
<td>int</td>
<td>-</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>int *</td>
<td>-</td>
</tr>
<tr>
<td>( \text{val}+1 )</td>
<td>int *</td>
<td>-</td>
</tr>
<tr>
<td>( &amp;\text{val}[2] )</td>
<td>int *</td>
<td>-</td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>int</td>
<td>-</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>-</td>
</tr>
<tr>
<td>( \text{val}+i )</td>
<td>int *</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x )</th>
<th>( x+4 )</th>
<th>( x+8 )</th>
<th>( x+12 )</th>
<th>( x+16 )</th>
<th>( x+20 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C: Null-terminated strings**

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

\[
\begin{array}{cccccccccc}
0x48 & 0x61 & 0x72 & 0x72 & 0x79 & 0x20 & 0x50 & 0x6f & 0x74 & 0x74 & 0x65 & 0x72 & 0x00 \\
\end{array}
\]

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: Dynamic memory allocation in the heap**

Managed by memory allocator:

- `void* malloc(size_t size);`
- `void free(void* ptr);`
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;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
free(zip);
```

C: Arrays of pointers to arrays of ...

```c
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int*)*5);
...
int* zip0 = zips[0];
z[0] = 0;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
```

C: scanf reads formatted input

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

C: classic bug using scanf

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

---

**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;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
free(zip);
```

---

**C: Arrays of pointers to arrays of ...**

```c
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int*)*5);
...
int* zip0 = zips[0];
z[0] = 0;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
```

---

**C: scanf reads formatted input**

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

---

**C: classic bug using scanf**

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

---

**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;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
    printf(" %d", zip[i]);
}
free(zip);
```

---

**C: Arrays of pointers to arrays of ...**

```c
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int*)*5);
...
int* zip0 = zips[0];
z[0] = 0;
z[1] = 2;
z[2] = 4;
z[3] = 8;
z[4] = 1;
```

---

**C: scanf reads formatted input**

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

---

**C: classic bug using scanf**

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