Chapter 16
Pointers and Arrays
Pointers and Arrays

We've seen examples of both of these in our C programs; now we'll see how they are implemented in LC-3.

Pointer

• Address of a variable in memory
• Allows us to indirectly access variables
  ➢ in other words, we can talk about its address rather than its value

Array

• A list of values arranged sequentially in memory
• Example: a list of telephone numbers
• Expression \( a[4] \) refers to the 5th element of the array \( a \)
Address vs. Value

Sometimes we want to deal with the **address** of a memory location, rather than the **value** it contains.

Recall example from Chapter 6: adding a column of numbers.
- R2 contains address of first location.
- Read value, add to sum, and increment R2 until all numbers have been processed.

R2 is a pointer -- it contains the address of data we’re interested in.
Another Need for Addresses

Consider the following function that's supposed to swap the values of its arguments.

```c
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}
```

With LC-3 implementation, we see why this does not work as intended.
Executing the Swap Function

\[\begin{align*}
&\text{before call} & \text{after call} \\
\text{R6} & \rightarrow & 3 \\
& & 4 \\
& & 4 \\
& & 3 \\
& \rightarrow & \text{Swap} \\
\text{firstVal} & 3 \to 4 \\
\text{secondVal} & 4 \to 3 \\
\text{valueB} & 4 \to 3 \\
\text{valueA} & 3 \to 4 \\
\text{main} & \rightarrow & \text{tempVal} \\
\end{align*}\]

These values changed...

...but these did not.

Swap needs \underline{addresses} of variables outside its own activation record.
Example

```c
int i;
int *ptr;
i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

- `int i;` - store the value 4 into the memory location associated with i
- `int *ptr;` - store the address of i into the memory location associated with ptr
- `i = 4;` - read the contents of memory at the address stored in ptr
- `ptr = &i;` - store the result into memory at the address stored in ptr
- `*ptr = *ptr + 1;`
Example: LC-3 Code

; i is 1st local (offset 0), ptr is 2nd (offset -1)

; i = 4;

    AND   R0, R0, #0 ; clear R0
    ADD   R0, R0, #4 ; put 4 in R0
    STR   R0, R5, #0 ; store in i

; ptr = &i;

    ADD   R0, R5, #0 ; R0 = R5 + 0 (addr of i)
    STR   R0, R5, #-1 ; store in ptr

; *ptr = *ptr + 1;

    LDR   R0, R5, #-1 ; R0 = ptr
    LDR   R1, R0, #0 ; load contents (*ptr)
    ADD   R1, R1, #1 ; add one
    STR   R1, R0, #0 ; store result where R0 points
Pointers as Arguments

Passing a pointer into a function allows the function to read/change memory outside its activation record.

```c
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change.
Passing Pointers to a Function

main() wants to swap the values of valueA and valueB passes the addresses to NewSwap:

```
NewSwap(&valueA, &valueB);
```

Code for passing arguments:

```
ADD R0, R5, #-1 ; addr of valueB
ADD R6, R6, #-1 ; push
STR R0, R6, #0
ADD R0, R5, #0 ; addr of valueA
ADD R6, R6, #-1 ; push
STR R0, R6, #0
```
Code Using Pointers

Inside the NewSwap routine

; int tempVal = *firstVal;
LDR R0, R5, #4 ; R0=xEFFA
LDR R1, R0, #0 ; R1=M[xEFFA]=3
STR R1, R5, #4 ; tempVal=3
; *firstVal = *secondVal;
LDR R1, R5, #5 ; R1=xEFF9
LDR R2, R1, #0 ; R1=M[xEFF9]=4
STR R2, R0, #0 ; M[xEFFA]=4
; *secondVal = tempVal;
LDR R2, R5, #0 ; R2=3
STR R2, R1, #0 ; M[xEFF9]=3
Array as a Local Variable

Array elements are allocated as part of the activation record.

```c
int grid[10];
```

First element (grid[0]) is at lowest address of allocated space.

If grid is first variable allocated, then R5 will point to grid[9].
LC-3 Code for Array References

; x = grid[3] + 1
    ADD R0, R5, #-9 ; R0 = &grid[0]
    LDR R1, R0, #3 ; R1 = grid[3]
    ADD R1, R1, #1 ; plus 1
    STR R1, R5, #-10 ; x = R1

; grid[6] = 5;
    AND R0, R0, #0
    ADD R0, R0, #5 ; R0 = 5
    ADD R1, R5, #-9 ; R1 = &grid[0]
    STR R0, R1, #6 ; grid[6] = R0
More LC-3 Code

; grid[x+1] = grid[x] + 2
LDR R0, R5, #−10 ; R0 = x
ADD R1, R5, #−9 ; R1 = &grid[0]
ADD R1, R0, R1 ; R1 = &grid[x]
LDR R2, R1, #0 ; R2 = grid[x]
ADD R2, R2, #2 ; add 2

LDR R0, R5, #−10 ; R0 = x
ADD R0, R0, #1 ; R0 = x+1
ADD R1, R5, #−9 ; R1 = &grid[0]
ADD R1, R0, R1 ; R1 = &grid[x+1]
STR R2, R1, #0 ; grid[x+1] = R2
A String is an Array of Characters

Allocate space for a string just like any other array:

```c
char outputString[16];
```

Space for string must contain room for terminating zero.

Special syntax for initializing a string:

```c
char outputString[16] = "Result = ";
```

...which is the same as:

```c
outputString[0] = 'R';
outputString[1] = 'e';
outputString[2] = 's';
...
```
Common Pitfalls with Arrays in C

Overrun array limits

• There is no checking at run-time or compile-time to see whether reference is within array bounds.

```c
int array[10];
int i;
for (i = 0; i <= 10; i++) array[i] = 0;
```

Declaration with variable size

• Size of array must be known at compile time.

```c
void SomeFunction(int num_elements) {
    int temp[num_elements];
    ...
}
```
Pointer Arithmetic

Address calculations depend on size of elements

- In our LC-3 code, we've been assuming one word per element.
  - e.g., to find 4th element, we add 4 to base address
- It's ok, because we've only shown code for int and char, both of which take up one word.
- If double, we'd have to add 8 to find address of 4th element.

C does size calculations under the covers, depending on size of item being pointed to:

```c
double x[10];
double *y = x;
*(y + 3) = 13;
```

allocates 20 words (2 per element)
same as x[3] -- base address plus 6