

# Chapter 16

## Pointers and Arrays

Based on slides © McGraw-Hill  
Additional material © 2004/2005/2006 Lewis/Martin

### Short-circuit Evaluation

What about this?

```
int func(int x)
{
    printf("%d", x);
    return x;
}

int main()
{
    int x = 0;
    int y = 1;
    if (func(x) && func(y)) {
        ...
    }
}
```

Answer: Prints just "0"

- Why? If left of "&&" operator is false, it *does not* evaluate right side
- "||" operator similar
- Java has similar semantics

### Agenda

#### 1. A few clarifications & addendums

- Short-circuit evaluation
- Static variables

#### 2. Then, on to pointers

- Pointers
- Arrays

### Static Variables

...and what about this?

```
int func(int x)
{
    static int sum = 0;
    sum = sum + x;
    return sum;
}

int main()
{
    printf("%d ", func(5));
    printf("%d", func(3));
}
```

```
int sum = 0;
int func(int x)
{
    sum = sum + x;
    return sum;
}
```

One difference: sum should be accessible only from func()

Displays "5 8"

- Why? static variables act much like global variables
- Its value persists across calls
- Actually stored in global segment (accessed by R4)
- Difference: only accessible within function

## Pointers and Arrays

We've seen examples of both of these in our LC-3 programs; now we'll see them in C

### 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

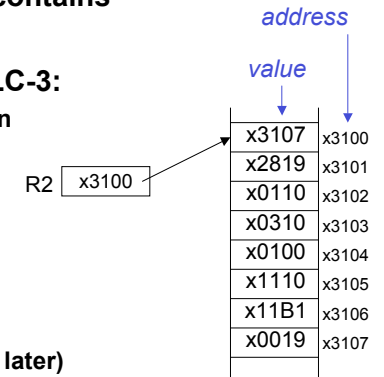
- A list of values arranged sequentially in memory
- Expression `a[4]` refers to the 5th element of the array `a`
- Example: video memory in Snake (2D)

## Address vs. Value

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

### Adding a column of numbers in LC-3:

- 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
- (It's also an array, but more on that later)

## Another Need for Addresses

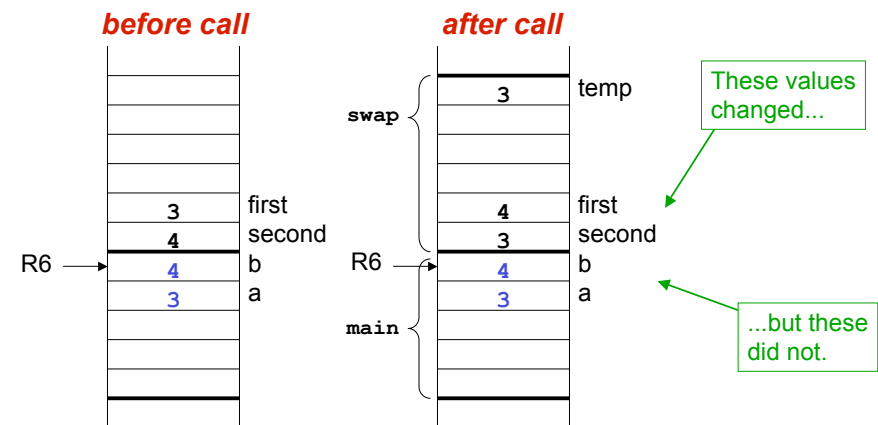
Consider the following function that's supposed to swap the values of its arguments.

```
void swap_wrong(int first, int second) {
    int temp = first;
    first = second;
    second = temp;
}

int main() {
    int a = 3, b = 4;
    swap_wrong(a, b);
}
```

What's wrong with this code?

## Executing the Swap Function



Swap needs addresses of variables outside its own activation record

## Pointers in C

C lets us talk about and manipulate pointers as variables and in expressions.

### Declaration

```
int *p; /* p is a pointer to an int */
```

A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, etc.

### Operators

**\*p** -- returns the value pointed to by p  
**&z** -- returns the address of variable z

## Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
printf("%d\n", i);
```

The diagram illustrates the memory state for the example code. It shows a memory stack with addresses from xEFF9 to xEFFE. Variable `i` is located at xEFFC and contains the value 4. Variable `ptr` is located at xEFFB and contains the address xEFFC. The code then increments the value at the address stored in `ptr` (which is `i`), changing it from 4 to 5. Finally, `printf` prints the value 5.

## 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, R6, #0 ; store in i
; ptr = &i;
ADD R0, R6, #0 ; R0 = R6 + 0 (addr of i)
STR R0, R6, #1 ; store in ptr
; *ptr = *ptr + 1;
LDR R0, R6, #1 ; R0 = ptr
LDR R1, R0, #0 ; load contents (*ptr)
ADD R1, R1, #1 ; add one
STR R1, R0, #0 ; store to *ptr
```

## Pointers as Arguments

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

```
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}
```

How would you do this in Java?

All arguments in C are pass-by-value.  
 Also true in Java, but Java has reference types

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

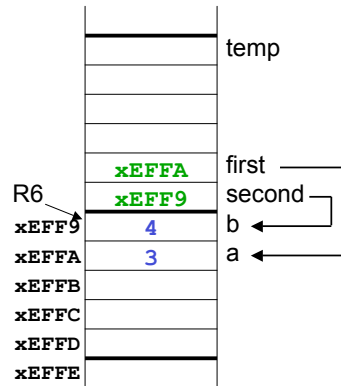
## Passing Pointers to a Function

main() wants to swap the values of “a” and “b”  
passes the addresses to swap():

```
swap(&a, &b);
```

Code for passing arguments:

```
ADD R0, R6, #0 ; addr of b
STR R0, R6, #-1
ADD R0, R6, #1 ; addr of a
STR R0, R6, #-2
```



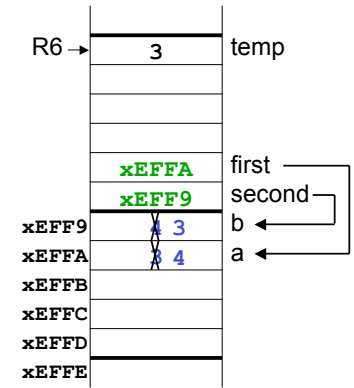
CSE 240

13

## Code Using Pointers

Inside the swap() routine

```
; int temp = *first;
LDR R0, R6, #4 ; R0=xEFFF8
LDR R1, R0, #0 ; R1=M[xEFFF8]=3
STR R1, R6, #0 ; temp=3
; *first = *second;
LDR R1, R6, #5 ; R1=xEFFF9
LDR R2, R1, #0 ; R2=M[xEFFF9]=4
LDR R0, R6, #4 ; R0=xEFFF8
STR R2, R0, #0 ; M[xEFFF8]=4
; *second = temp;
LDR R2, R6, #0 ; R2=3
LDR R1, R6, #5 ; R1=xEFFF9
STR R2, R1, #0 ; M[xEFFF9]=3
```



CSE 240

14

## Using Arguments for Results

Pass address of variable where you want result stored

- Useful for multiple results
- Example:
  - Return value via pointer
  - Return status code as function result

This solves the mystery of the ‘&’ for calling scanf():

```
scanf("%d %d", &data1, &data2);
```

read decimal integers  
into data1 and data2

CSE 240

15

## Null Pointer

Sometimes we want a pointer that points to nothing.  
In other words, we declare a pointer, but we’re not ready to actually point to something yet.

```
int *p;
p = NULL; /* p is a null pointer */
```

**NULL** is a predefined macro that contains a value that a non-null pointer should never hold.

- Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms
- Dereferencing a NULL pointer: program crash!

```
➢ int *p = NULL; printf("%d", *p); // CRASH!
```

CSE 240

16

## Pointer Problems

What does this do?

```
int *x;
*x = 10;
```

Answer: writes “10” into a random location in memory

- What would java do?

What’s wrong with:

```
int* func()
{
    int x = 10;
    return &x;
}
```

Answer: storage for “x” disappears on return, so the returned pointer is dangling

- What would java do?

CSE 240

17

## Declaring Pointers

The \* operator binds to the variable name, not the type

All the same:

- `int* x, y;`
- `int *x, y;`
- `int *x; int y;`

Suggested solution: Declare only one variable per line

- Avoids this problem
- Easier to comment
- Clearer
- Don’t worry about “saving space”

CSE 240

18

## Arrays

How do we allocate a group of memory locations?

- Character string
- Table of numbers

How about this?

Not too bad, but...

- What if there are 100 numbers?
- How do we write a loop to process each number?

```
int num0;
int num1;
int num2;
int num3;
```

Fortunately, C gives us a better way -- the **array**.

```
int num[4];
```

Declares a sequence of four integers, referenced by:

```
num[0], num[1], num[2], num[3].
```

CSE 240

19

## Array Syntax

Declaration

```
type variable[num_elements];
```

all array elements  
are of the same type

number of elements must be  
known at compile-time

Array Reference

```
variable[index];
```

i-th element of array (starting with zero);  
**no limit checking** at compile-time or run-time

CSE 240

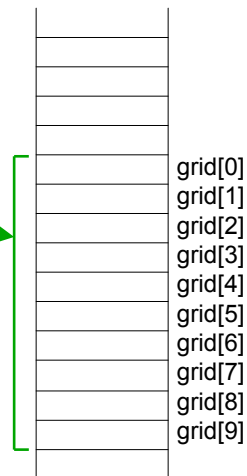
20

## Array as a Local Variable

Array elements are allocated as part of the activation record

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

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

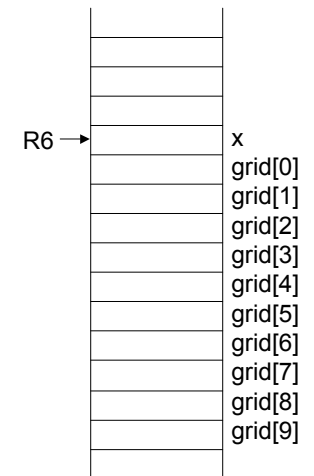


## LC-3 Code for Array References

```
; x = grid[3] + 1
ADD R0, R6, #1 ; R0 = &grid[0]
LDR R1, R0, #3 ; R1 = grid[3]
ADD R1, R1, #1 ; plus 1
STR R1, R6, #0 ; x = R1
```

```
; grid[6] = 5;
AND R0, R0, #0 ; R0 = 5
ADD R0, R0, #5 ; R0 = 5
ADD R1, R6, #1 ; R1 = &grid[0]
STR R0, R1, #6 ; grid[6] = R0
```

Compiler can combine

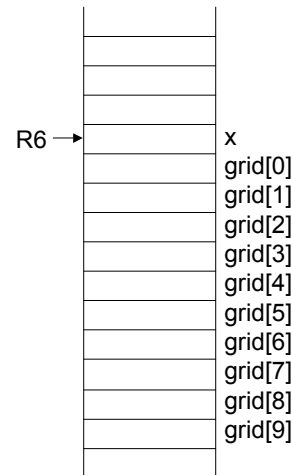


## More LC-3 Code

```
; grid[x+1] = grid[x] + 2
LDR R0, R6, #0 ; R0 = x
ADD R1, R6, #1 ; 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, R6, #0 ; R0 = x
ADD R0, R0, #1 ; R0 = x+1
ADD R1, R6, #1 ; R1 = &grid[0]
ADD R1, R0, R1 ; R1 = &grid[x+1]
STR R2, R1, #0 ; grid[x+1] = R2
```



## Passing Arrays as Arguments

### C passes arrays by address

- the address of the array (i.e., of the first element) is written to the function's activation record
- otherwise, would have to copy each element

```
int main()
{
    int numbers[MAX_NUMS];
    ...
    mean = average(numbers, MAX_NUMS);
    ...
}

int average(int values[], int size)
{
    int index, sum = 0;
    for (index = 0; index < size; index++) {
        sum = sum + values[index];
    }
    return (sum / size);
}
```

This must be a constant, e.g., #define MAX\_NUMS 10

## More on Passing Arrays

### No run-time length information

- C doesn't track length of arrays
- No Java-like `values.length` construct
- Thus, you need to pass length or use a sentinel

```
int average(int values[], int size)
{
    int index, sum;
    for (index = 0; index < size; index++) {
        sum = sum + values[index];
    }
    return (sum / size);
}
```

CSE 240

25

## Relationship between Arrays and Pointers

An array name is essentially a pointer to the first element in the array

```
char data[10];
char *cptr;

cptr = data; /* points to data[0] */
```

### Difference:

Can change the contents of cptr, as in

```
cptr = cptr + 1;
```

CSE 240

26

## Correspondence between Ptr and Array Notation

Given the declarations on the previous page, each line below gives three equivalent expressions:

<code>cptr</code>	<code>data</code>	<code>&amp;data[0]</code>
<code>(cptr + n)</code>	<code>(data + n)</code>	<code>&amp;data[n]</code>
<code>*cptr</code>	<code>*data</code>	<code>data[0]</code>
<code>*(cptr + n)</code>	<code>*(data + n)</code>	<code>data[n]</code>

CSE 240

27

## Pointer Subtraction and Equality

Nasty, but C allows it:

```
void function(int* start, int* end)
{
    int i;
    while (end - start >= 0) {
        *start = 0;
        start++;
    }
}

int array[10]...
function(array[0], array[9])
```

### Don't do this!

Alternative: `while (end != start) {`

- Significantly better, but still too nasty
- What if start is > end, or not part of same array?

CSE 240

28

## More on 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, which takes 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:

```
double x[10];  
double *y = x;  
*(y + 3) = 100;
```

allocates 20 words (2 per element)

same as x[3] -- base address plus 6

CSE 240

29

## A String is an Null-Terminated Character Array

Allocate space for a string just like any other array:

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

Space for string must contain room for **terminating zero**

Special syntax for initializing a string:

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

...which is the same as:

```
outputString[0] = 'R';  
outputString[1] = 'e';  
outputString[2] = 's';  
...  
outputString[9] = '\0'; // Null terminator
```

CSE 240


31

## 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


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


- Remember, C does not track array length

### Declaration with variable size

- Size of array must be known at compile time

```
void func(int num_elements)  
{  
    int temp[num_elements];  
    ...  
}
```



CSE 240

30

## I/O with Strings

Printf and scanf use "%s" format character for string

**Printf** -- print characters up to terminating zero

```
printf("%s", outputString);
```

**Scanf** -- read characters until whitespace, store result in string, and terminate with zero

```
scanf("%s", inputString);
```

Why no & operator?

CSE 240

32



## String Length - Array Style

```
int strlen(char str[])
{
    int i = 0;
    while (str[i] != '\0') {
        i++;
    }
    return i;
}
```

CSE 240

33

## String Length - Pointer Style

```
int strlen(char* str)
{
    int i = 0;
    while (*str != '\0') {
        i++;
        str++;
    }
    return i;
}
```

CSE 240

34

## String Copy - Array Style

```
void strcpy(char dest[], char src[])
{
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0'
}
```

**Clean, clear**

CSE 240

35

## String Copy - Array Style #2

```
void strcpy(char dest[], char src[])
{
    int i = 0;
    while ((dest[i] = src[i]) != '\0') {
        i++;
    }
}
```

**Use of assignment in while**

- Harder to read, in my opinion

CSE 240

36

## String Copy - Pointer Style

```
void strcpy(char* dest, char* src)
{
    while ((*dest = *src) != '\0') {
        dest++;
        src++;
    }
}
```

CSE 240

37

## String Copy - Pointer Style #2

```
void strcpy(char* dest, char* src)
{
    while ((*dest++ = *src++) != '\0') {
        // nothing
    }
}
```

### Difficult to read

- “Experienced C programmers would prefer...” - K&R
- I disagree: please avoid this type of code (really)

### What happens if `dest` is too small?

- Bad things...

CSE 240

38

## C String Library

### C has a limited string library

- All based on null-terminated strings
- `#include <string.h>` to use them

### Functions include

- `int strlen(char* str)`
- `void strcpy(char* dest, char* src)`
- `int strcmp(char* s1, char* s2)`
  - Returns 0 on equal, -1 or 1 if greater or less
  - Remember, 0 is false, so equal returns false!
- `strcat(char* dest, char* src)`
  - string concatenation (appending two strings)
- `strncpy(char* dest, char* src, int max_length)`
- `strncmp(char* s1, char* s2, int max_length)`
- `strncat(char* dest, char* src, int max_length)`
- Plus some more...

CSE 240

39

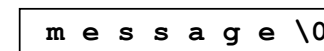
## String Declaration Nastiness

### What's the difference between:

- `char amessage[] = "message"`
- `char *pmessage = "message"`

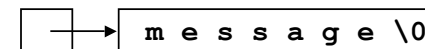
### Answer:

- `char amessage[] = "message" // single array`



```
m e s s a g e \0
```

- `char *pmessage = "message" // pointer and array`



```
□ → m e s s a g e \0
```

CSE 240

40

## Main(), revisited

### Main supports command line parameters

- Much like Java's

```
public static void main(String[] args)
```

### Main supports command line parameters:

```
int main(int argc, char *argv[])
```

```
{  
    int i;  
    for (i = 0; i<argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
}
```

← An array of strings

### Displays each command-line argument

- Zero-parameter is the program name