

# Chapter 16

## Pointers and Arrays

## 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
- Example: a list of telephone numbers
- Expression **a [4]** refers to the 5th element of the array **a**

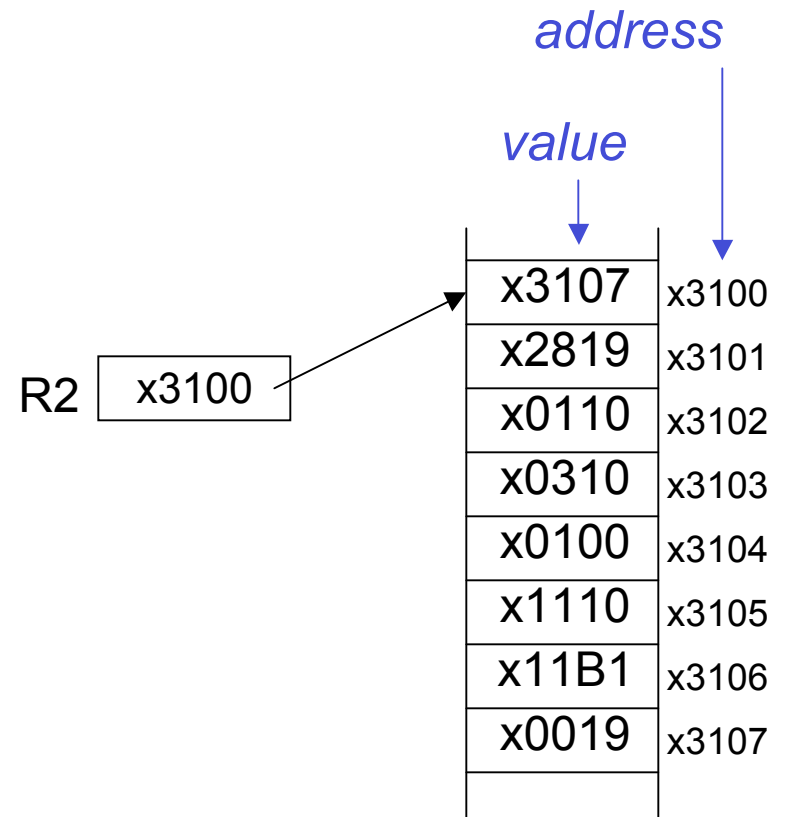
## Address vs. Value

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.

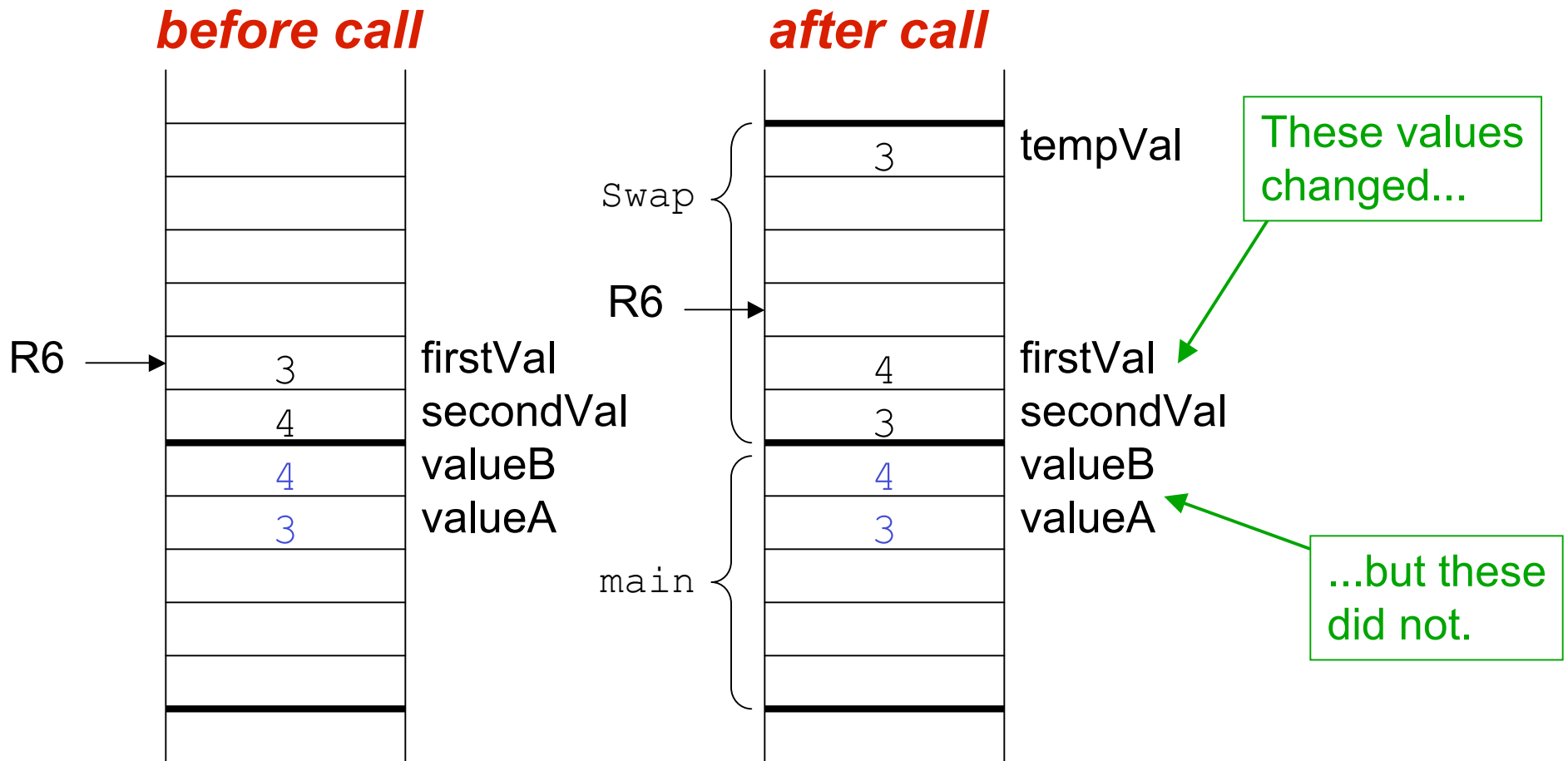


## Another Need for Addresses

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

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

## 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;
```


store the value 4 into the memory location associated with i



```
i = 4;
```


```
ptr = &i;
```

store the address of i into the memory location associated with ptr




```
*ptr = *ptr + 1;
```

read the contents of memory at the address stored in ptr



store the result into memory at the address stored in ptr



## 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)*

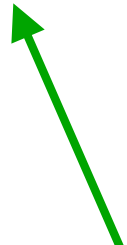
ADD R1, R1, #1 *; add one*

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.

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



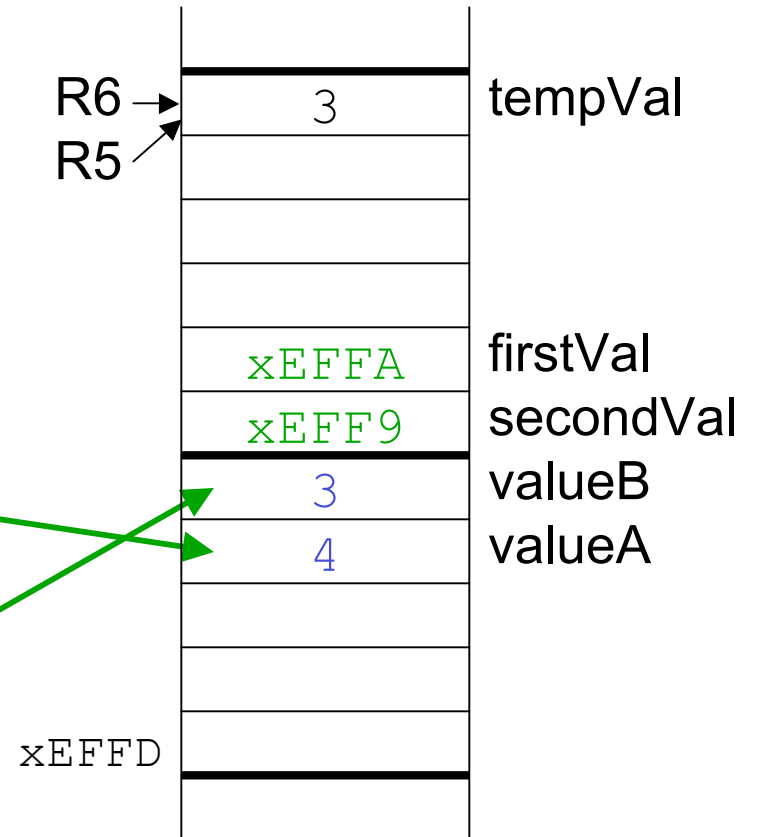
**Arguments are integer pointers.  
Caller passes addresses of variables that it wants function to change.**



# Code Using Pointers

## Inside the NewSwap routine

```
; int tempVal = *firstVal;  
LDR R0, R5, #4 ; R0=xEFFA  
LDR R1, R0, #0 ; R1=M[xEFA]=3  
STR R1, R5, #4 ; tempVal=3  
; *firstVal = *secondVal;  
LDR R1, R5, #5 ; R1=xEFF9  
LDR R2, R1, #0 ; R2=M[xEFF9]=4  
STR R2, R0, #0 ; M[xEFA]=4  
; *secondVal = tempVal;  
LDR R2, R5, #0 ; R2=3  
STR R2, R1, #0 ; M[xEFF9]=3
```



## 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.**

## 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 why ‘&’ with argument to scanf:**

```
scanf("%d ", &dataIn);
```



**read a decimal integer  
and store in dataIn**

## Syntax for Pointer Operators

### Declaring a pointer

```
type *var;  
type* var;
```

Either of these work -- whitespace doesn't matter.

Type of variable is `int*` (integer pointer), `char*` (char pointer), etc.

### Creating a pointer

```
&var
```

Must be applied to a memory object, such as a variable.

In other words, `&3` is not allowed.

### Dereferencing

Can be applied to any expression. All of these are legal:

`*var`                      contents of mem loc pointed to by var

`**var`                      contents of mem loc pointed to by  
memory location pointed to by var

`*3`                              contents of memory location 3

## Example using Pointers

**IntDivide performs both integer division and remainder, returning results via pointers. (Returns -1 if divide by zero.)**

```
int IntDivide(int x, int y, int *quoPtr, int *remPtr);

main()
{
    int dividend, divisor; /* numbers for divide op */
    int quotient, remainder; /* results */
    int error;
    /* ...code for dividend, divisor input removed... */
    error = IntDivide(dividend, divisor,
                     &quotient, &remainder);
    /* ...remaining code removed... */
}
```

## C Code for IntDivide

```
int IntDivide(int x, int y, int *quoPtr, int *remPtr)
{
    if (y != 0) {
        *quoPtr = x / y; /* quotient in *quoPtr */
        *remPtr = x % y; /* remainder in *remPtr */
        return 0;
    }
    else
        return -1;
}
```

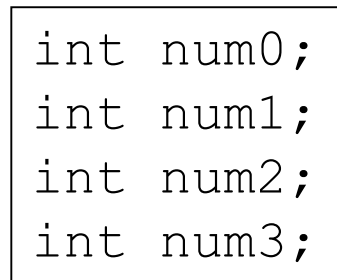
# Arrays

## How do we allocate a group of memory locations?

- character string
- table of numbers

How about this?

Not too bad, but...



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

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

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].
```

# 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

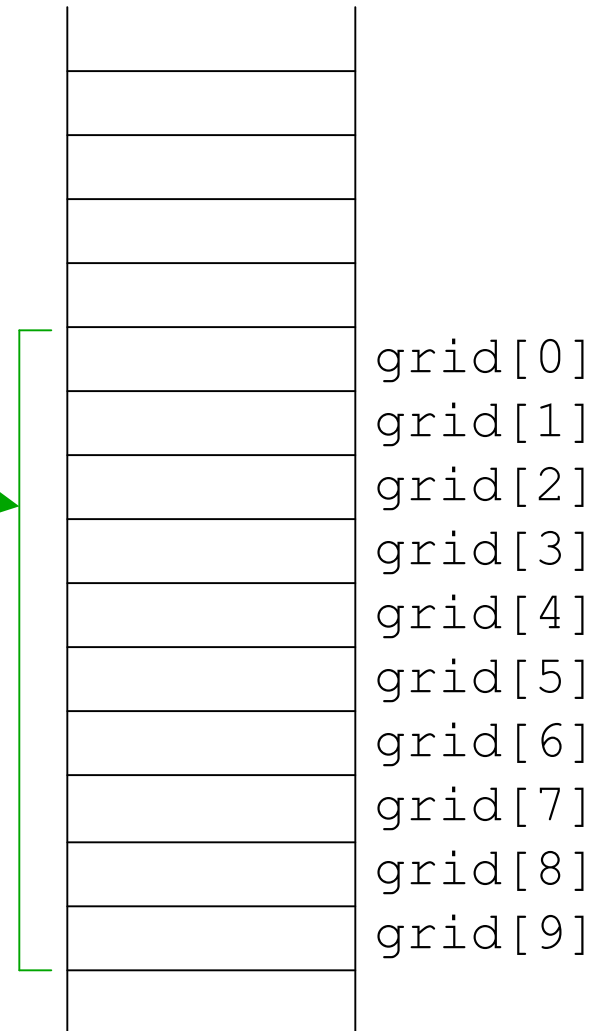
## Array as a Local Variable

Array elements are allocated as part of the activation record.

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

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

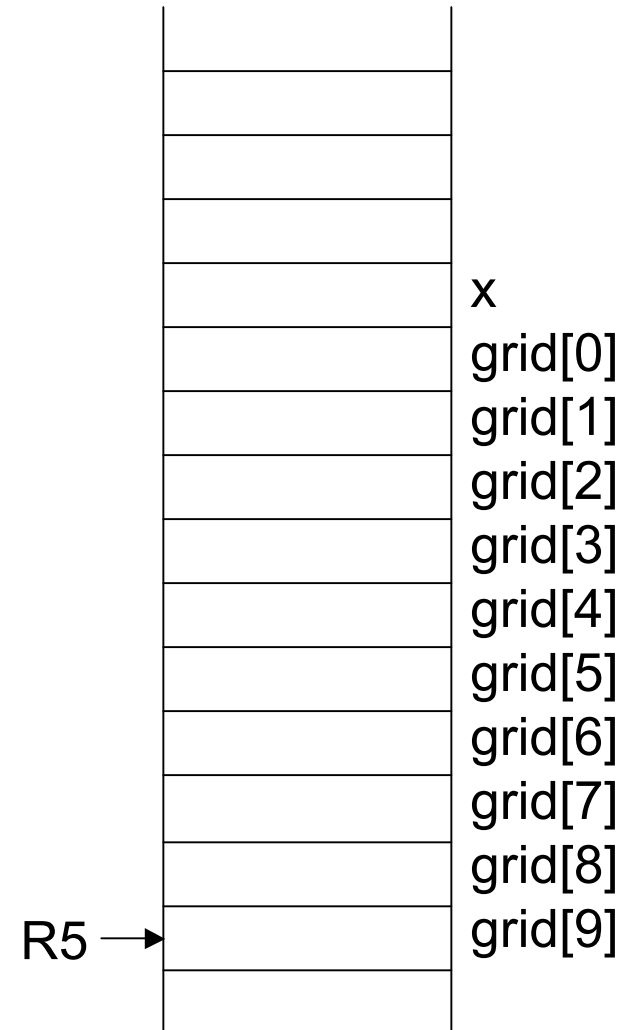
If `grid` is first variable allocated, then `R5` will point to `grid[9]`.



## LC-3 Code for Array References

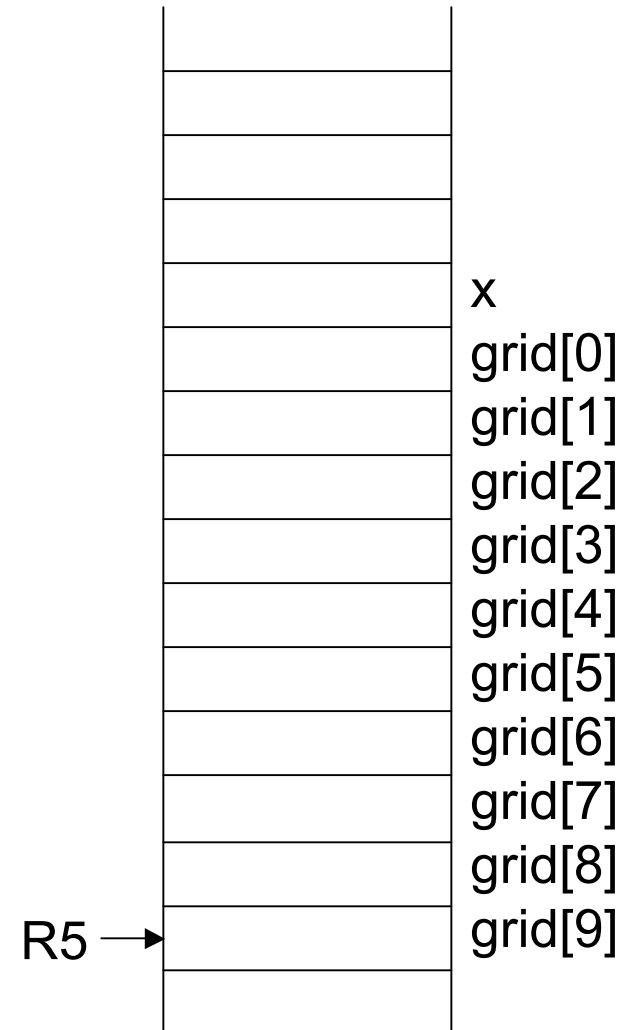
```
; x = grid[3] + 1
  ADD R0, R5, #-9   ; R0 = &grid[0]
  LDR R1, R0, #3    ; R1 = grid[3]
  ADD R1, R1, #1    ; plus 1
  STR R1, R5, #-10 ; x = R1

; 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
```



## Passing Arrays as Arguments

### C passes arrays by reference

- 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

```
main() {  
    int numbers[MAX_NUMS];  
    ...  
    mean = Average(numbers);  
    ...  
}  
int Average(int inputValues[MAX_NUMS]) {  
    ...  
    for (index = 0; index < MAX_NUMS; index++)  
        sum = sum + inputValues[index];  
    return (sum / MAX_NUMS);  
}
```

This must be a constant, e.g.,  
`#define MAX_NUMS 10`

## A String is an Array of Characters

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[16] = "Result = ";
```

...which is the same as:

```
outputString[0] = 'R';
```

```
outputString[1] = 'e';
```

```
outputString[2] = 's';
```

```
...
```

## 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);
```

## Relationship between Arrays and Pointers

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

```
char word[10];  
char *cptr;  
  
cptr = word; /* points to word[0] */
```

### ***Difference:***

Can change the contents of cptr, as in

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

(The identifier "word" is not a variable.)

## Correspondence between Ptr and Array Notation

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

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

## 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;
```

### Declaration with variable size

- Size of array must be known at compile time.

```
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:

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

allocates 20 words (2 per element)

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