Mc Graw Hill Education

Because learning changes everything.®

From Bits and Gates to C and Beyond

Pointers and Arrays

Chapter 16

Pointers and Arrays

Pointer = address of a variable (or any memory location)

- Allows indirect access to variables – for example, allow a function to change a variable in the caller.
- Pointer-based data structures that can grow or shrink to meet needs.

Array = sequence of same-typed values, contiguous in memory

- Create a group of data values with a single name, rather than many different variables.
- Convenient representation of vectors, matrices, strings, etc.

We encountered both of these concepts when learning LC-3 programming. Now we'll see how they are expressed and used in C.

Pointer = Indirect Access

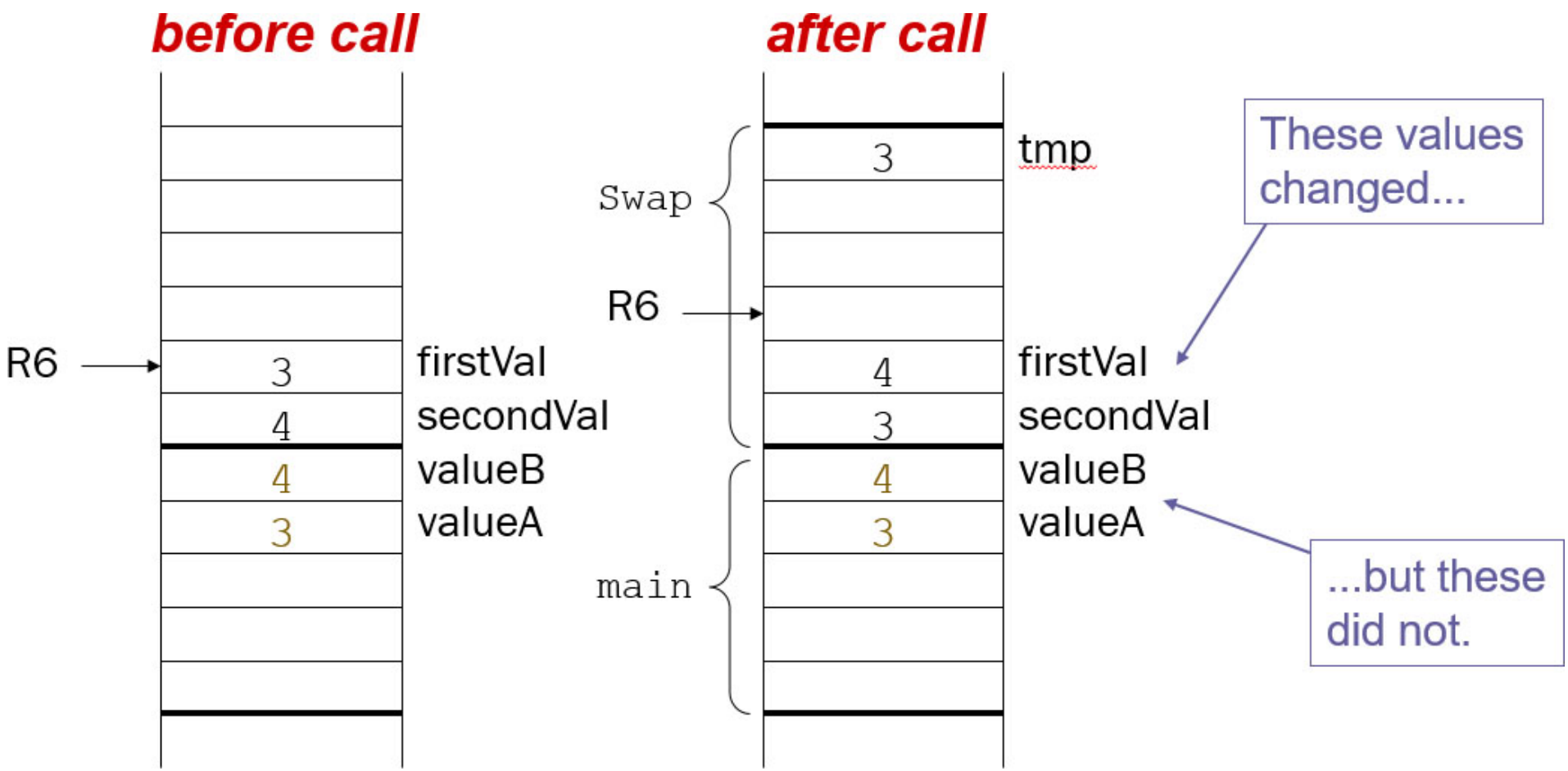
Suppose we want to write a function that swaps the values of two variables. Here is our first attempt:

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

Does this work? Consider the following code -- what are the values of valueA and valueB when the function returns?

```
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
```

Executing the Swap Function



To access something outside of its activation record, Swap needs an **address**.

Access the text alternative for slide images.

Pointers in C: Variables and Operators

C allows us to declare a variable that holds a **pointer** value.

We need to specify the type of data to which the pointer points.
Add * to the type name to declare a pointer to that type.

```
int * p;    // variable p holds a pointer to an int
```

We say: the type of `p` is `int*`. We can point to any type: `char*`, `double*`, etc.

Operators

To create a pointer value, we use the (unary) `&` operator -- calculates the address of a variable. Suppose `z` is an `int` variable...

```
p = &z;    // get the address of variable z, assign to p
           // now: p 'points to' z
```

To get the value in the location referenced by a pointer, use the (unary) `*` operator.

```
z = *p;    // dereference the pointer p, assign value to z
```

Implementing the & Operator

```
int object;
int *ptr;

object = 4;
ptr = &object;
```

```
; offset of object = 0
; offset of ptr = -1

AND R0, R0, #0
ADD R0, R0, #4
STR R0, R5, #0   ; object = 4

ADD R0, R5, #0   ; address of object
STR R0, R5, #-1  ; ptr = &object
```

The compiler knows the address of every variable.
When we use LDR/STR, we add an offset to R5 (or R4) and then load/store.
To compute the address without load/store, just use ADD.

Implementing the * Operator

```
int object;
int *ptr;

object = 4;
ptr = &object;
*ptr = *ptr + 1;
```

```
; offset of object = 0
; offset of ptr = -1

AND R0, R0, #0
ADD R0, R0, #4
STR R0, R5, #0   ; object = 4
ADD R0, R5, #0   ; address of object
STR R0, R5, #-1  ; ptr = &object

LDR R0, R5, #-1  ; R0 has value of ptr
LDR R1, R0, #0   ; *ptr: use R0 as address to
                 ; load from where ptr points
ADD R1, R1, #1   ; *ptr + 1
STR R1, R0, #0   ; store to where ptr points
```

We load the value of a pointer variable, and then use that address for loads and stores that dereference the variable.

Passing a Reference (Pointer) to a Function

Now we can fix our Swap function. Instead of passing the values, we want to pass the addresses of the variables.

So the type of our parameters are `int*` instead of `int`.

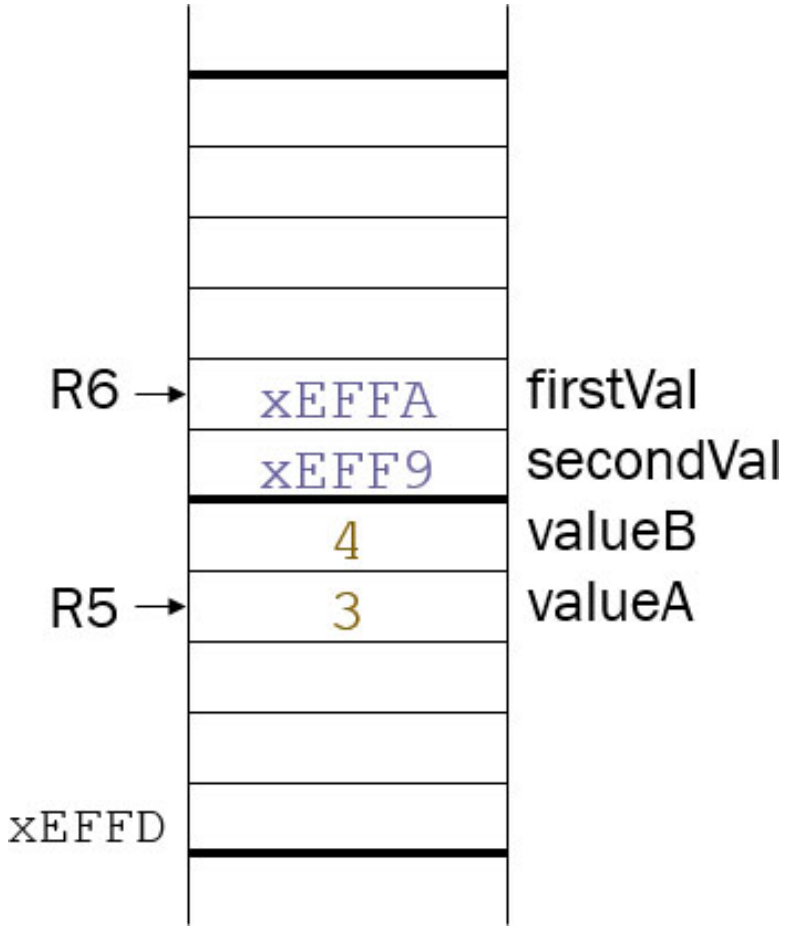
```
void NewSwap(int *firstVal, int *secondVal)
{
   int tmp;   // still an int, because it will hold a value
   // use dereference to manipulate the non-local values
   tmp = *firstVal;
   *firstVal = *secondVal;
   *secondVal = tmp;
}
```

And when we call this function, we don't pass the values of valueA and valueB, we pass their addresses:

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

Code that Calls NewSwap

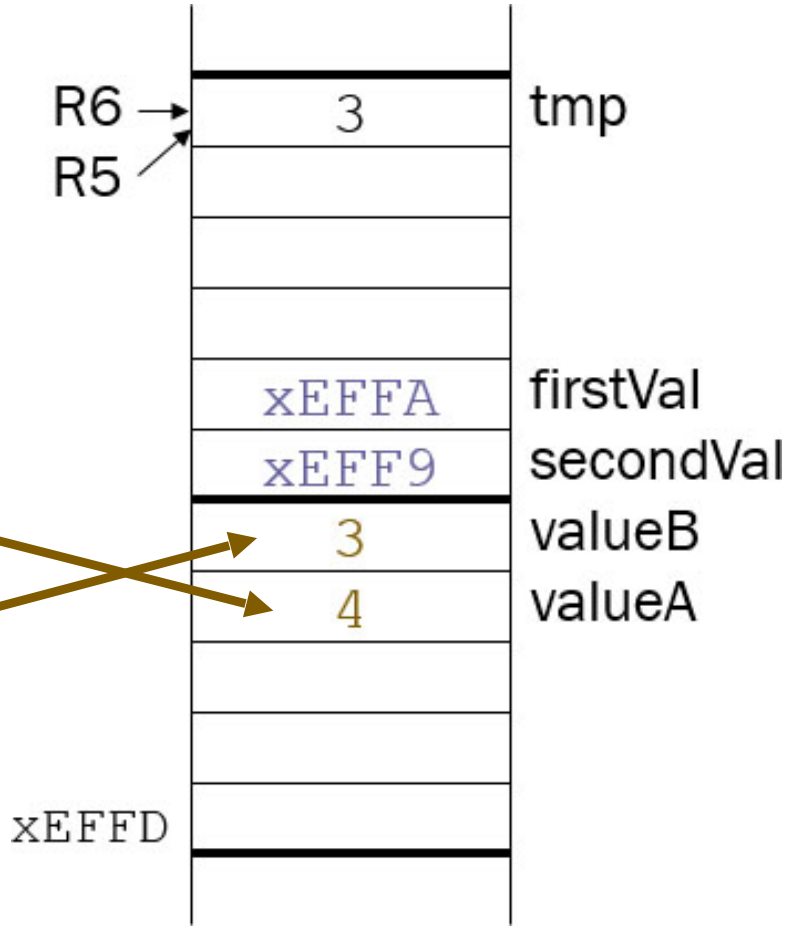
```
NewSwap(&valueA, &valueB);


ADD R0, R5, #-1   ; &valueB
ADD R6, R6, #-1   ; push
STR R0, R6, #0
ADD R0, R5, #0    ; &valueA
ADD R6, R6, #-1   ; push
STR R0, R6, #0

JSR NewSwap

ADD R6, R6, #3    ; pop r.v. (none)
                  ; and arguments
```



Access the text alternative for slide images.

Inside the NewSwap Function

```
; tmp = *firstVal;
LDR   R0, R5, #4 ; R0=xEFFA
LDR   R1, R0, #0 ; R1=M[xEFFA]=3
STR   R1, R5, #0 ; tmp=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 = tmp;
LDR   R2, R5, #0 ; R2=3
STR   R2, R1, #0 ; M[xEFF9]=3
```



Access the text alternative for slide images.

NULL Pointer

Sometimes we want to create a pointer that points to nothing.
Perhaps we don't know yet where it should point...

This is known as a **null pointer**.

By convention, we use the value 0 to represent the null pointer.
C provides the symbolic name **NULL** -- using NULL makes the code more readable and makes it clear that we intend the pointer to be null.

```
int *ptr = NULL;  // declare ptr, initialize it to NULL
```

Best practice: Initialize unused pointers to NULL. If you don't initialize, the pointer will point to a random memory location, which can lead to strange and interesting bugs.

If your code tries to dereference a null pointer, the program will crash -- that's a much better outcome than reading/writing from an unknown memory location.

`scanf` and Pointers

We've actually been using pointers in our C programs all along:

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

Now that you understand what `&` means, you can see that we are passing the address of the `inValue` variable to the `scanf` function. That's because we want `scanf` to modify that variable when it reads the input value -- the only way this can happen is if we tell `scanf` the location of the variable.

We can use this feature to create functions that calculate and return multiple values. A function can only have one return value, but we can use as many pointers as we want, giving locations to store additional values.

For an example, see the next slide...

Example: Using Pointers to Return Values 1

Write a function that divides two integers and returns both the quotient and the remainder. Since we can only have one return value, we'll use pointers to tell the function where to put the result.

Use the return value as an error code. Return -1 if the divisor is zero, and 0 otherwise.

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

Example: Using Pointers to Return Values 2

It's a good idea to return some sort of status / error code for a function like this. The caller needs to know whether anything useful was stored in the specified location.

```
int dividend, divisor;
int quotient, remainder;
int error;

// ... code to get dividend and divisor from user ...

error = IntDivide(dividend, divisor, &quotient, &remainder);
if (!error)
   printf("Quotient: %d, Remainder: %d\n", quotient, remainder);
else
   printf("IntDivide failed.\n");
```

Arrays

An array is a convenient way to create a collection of values, all of the same type, identified by a single name.

Consider creating a gradebook program for a class with 100 students. You need to store 100 different grades -- do you want 100 different variables?

Instead, we declare an array of doubles to hold a grade for each student.

```
double grades[100];
```

type

variable name

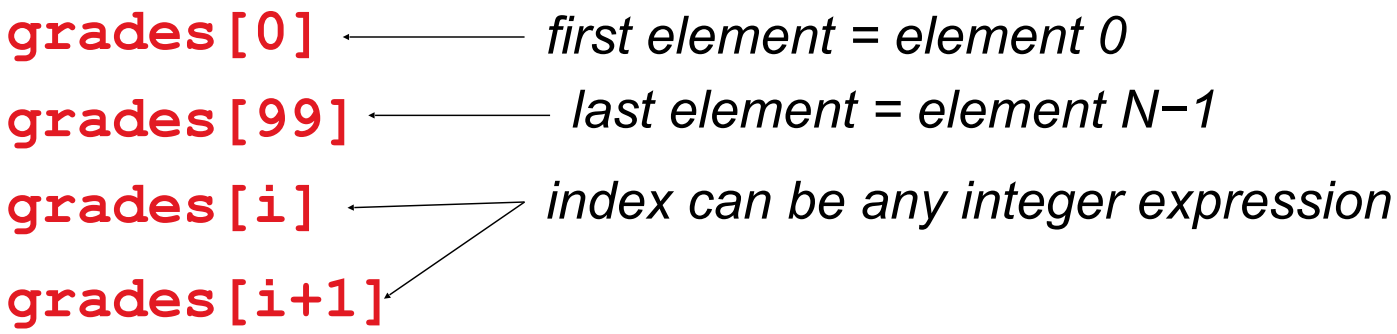
size -- number of elements

Array Indexing

Each element of the array is assigned an **index**, which indicates its place in the array.

A N-element array has indices from 0 to (N − 1).

To access an array element:



It is convenient to use a loop to process elements of an array. For example, here we compute the average grade:

```
double sum = 0.0;
for (int i = 0; i < 100; i++) sum += grade[i];
sum = sum / 100.0;
```

Implementing Arrays

When we declare an array with a fixed size, the compiler allocates the appropriate amount of space on the stack (or in the global area).

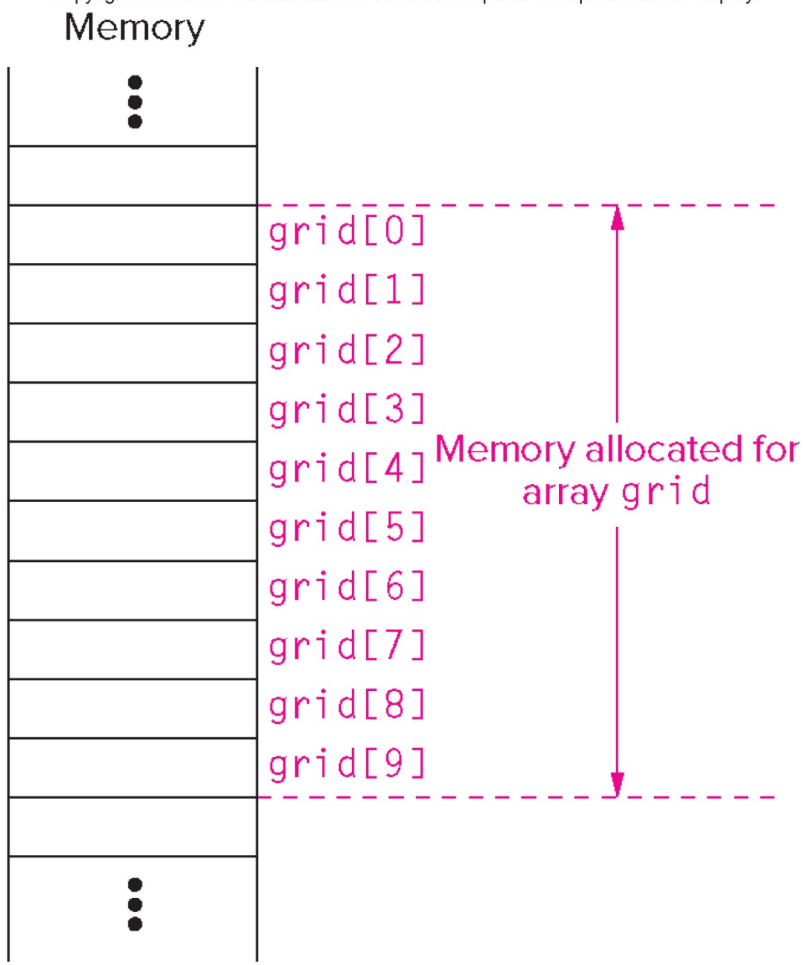
Space = (# elements) x (size of element)

The elements are always contiguous in memory, and element 0 is always at the lowest allocated address.

The figure to the right illustrates the memory allocation for this array:

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

The symbol table stores the offset of the first element of the array.



Access the text alternative for slide images.

Accessing an Array Element

When an expression contains an array element, the compiler needs to load/store from/to the appropriate memory address. The compiler must do the following:

1. Get the starting address of the array.
2. Evaluate the index expression.
3. Multiply the index (step 2) by the size of array element -- the type of the array elements is part of the declaration, and is stored in the symbol table.
   (For our LC-3 examples, we will use integer arrays, so the "multiply by one" step can be ignored.)
4. Add the scaled index (step 3) to the starting address (step 1).
5. Load or store using the address computed in step 5.

Implementation: Constant index

```
int grid[10];
// ...
grid[6] = grid[3] + 1;
```

```
; first element of grid has offset -9
; step 1 - get base address
ADD  R0, R5, #-9
; step 2, 3, 4, 5 -
; add index to address and load
LDR  R1, R0, #3  ; R1 has grid[3]
ADD  R1, R1, #1
; step 1 - R0 still has base address
; step 2, 3, 4, 5 -
; add index to base address and store
STR  R1, R0, #6
```

When index is constant, compiler can calculate the address.
Can use offset of LDR/STR to combine add with load/store.

Implementation: Index expression

```
int grid[10];
int x;
// ...
grid[x+1] = grid[x] + 2;
```

```
; first element of grid has offset -9
; step 1 - get base address
ADD  R0, R5, #-9
; step 2,3 - evaluate index (times 1)
LDR  R1, R5, #-10   ; x is index
ADD  R2, R0, R1 ; step 4 - add to address
LDR  R3, R2, #0 ; step 5 - load
ADD  R3, R3, #2 ; grid[x]+2
; step 2,3 - evaluate index (times 1)
LDR  R1, R5, #-10
ADD  R1, R1, #1     ; x + 1 is index
ADD  R2, R0, R1 ; step 4 - add to address
STR  R3, R2, #0 ; step 5 - store
```

When index is an expression, must generate code to evaluate the expression and compute the address at runtime.

Example: Counting Repeated Values 1

Write a program to

a. read a sequence of integers from the keyboard,
b. count the number of times each number is repeated,
c. print each number and the number of times it was entered.

Use a symbolic value MAX_NUMS as the number of items.

Declare two arrays:

- One to hold the numbers that are entered.
- One to hold the number of times each input value occurs.

Example: Counting Repeated Values 2

```
#include <stdio.h>
#define MAX_NUMS 10

int main(void)
{
   int repIndex;           // loop index
   int numbers[MAX_NUM];   // array of input values
   int repeats[MAX_NUM];   // occurrences of each value

   // get numbers from user
   printf("Enter %d numbers.\n", MAX_NUMS);
   for (int index = 0; index < MAX_NUMS; index++) {
      printf("Input number %d: ", index);
      scanf("%d", &numbers[index]);  // note: address of element
   }

   // next slide...
```

Example: Counting Repeated Values 3

```
   // scan through entire array, counting number of
   // repeats for each element in original array
   for (int index = 0; index < MAX_NUMS; index++) {
      repeats[index] = 0;
      for (repIndex = 0; repIndex < MAX_NUMS; repIndex++) {
         if (numbers[repIndex] == numbers[index]) {
            repeats[index]++;
            // note: can increment/decrement an array element
         }
      }
   }

   // print results
   for (int index = 0; index < MAX_NUMS; index++) {
      printf("Original number %d. Number of repeats %d.\n",
             numbers[index], repeats[index]);
   }
}
```

Array as Parameters

In C, arrays are always passed by reference.

So far, when we call a function, we push a value on the stack for each parameter. When the expression is a variable name, this is essentially pushing a **copy** of the variable to the callee function.

Any change to the parameter variable is local to the callee function and does not affect the original variable (as we saw with the Swap function).

We do not want to create a copy of an array.

Could be thousands of elements! The callee function might just need to access a small part of the array, or just read the array, so it's a waste of effort to copy all of those elements.

Instead, we pass a reference (pointer!) to the array.

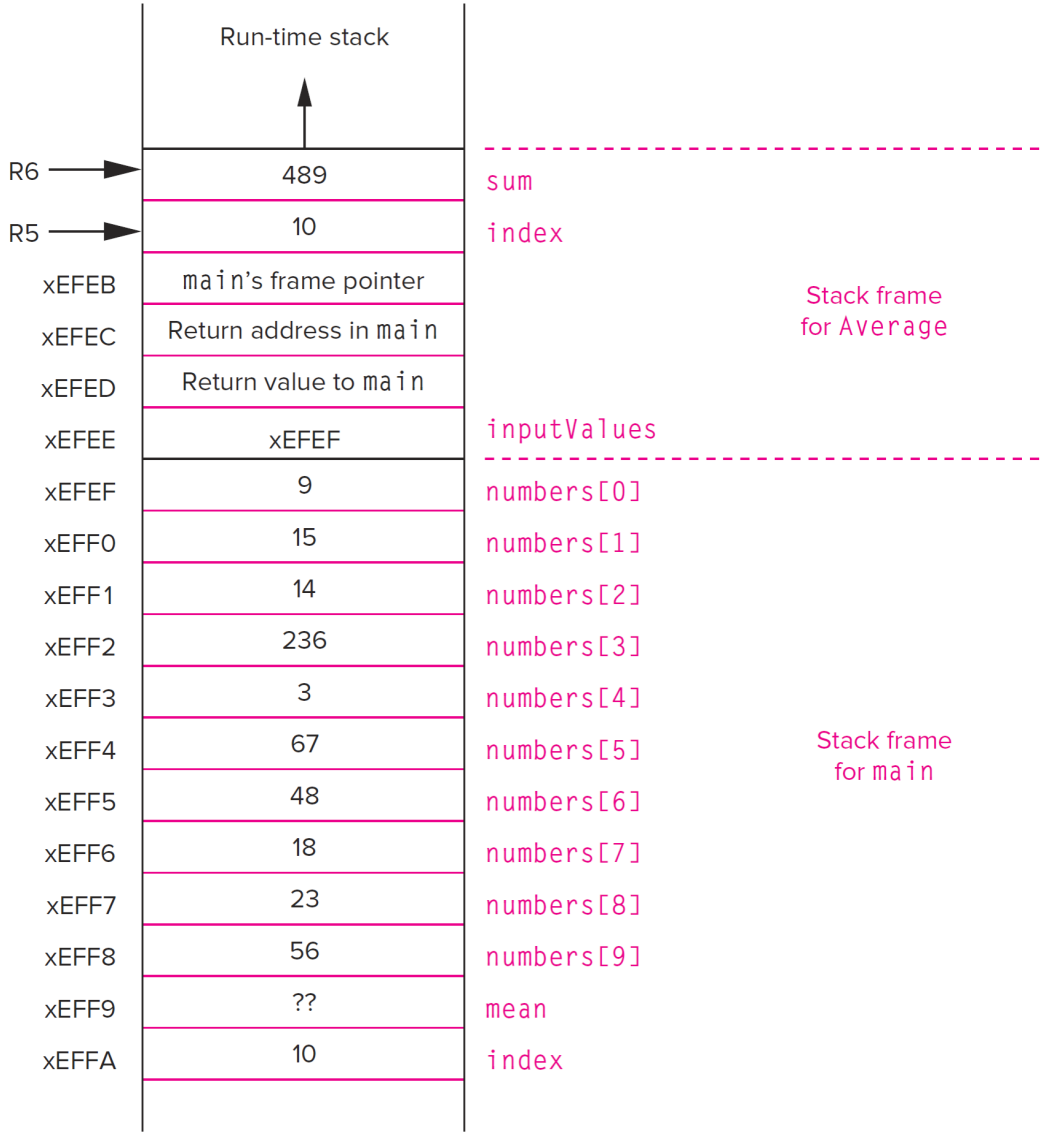
Function: Calculate the Average of an Array

```
// declaring the function
int Average(int data[]);  // [] tells compiler that the parameter
                          // is an array
// for this problem, assume MAX_NUMS is the size of the array


// function implementation
int Average(int inputValues[]) {
   int sum = 0;

   for (index = 0; index < MAX_NUMS; index++) {
      sum += inputValues[index];   // just access array as usual
   }

   return sum / MAX_NUMS;   // note: integer division
}
```

Calling the Function

```
#define MAX_NUMS 10
int Average(int n[]);

int main(void) {
   int mean;    // will hold average
   int numbers[MAX_NUMS];   // data values

   printf("Enter %d numbers.\n", MAX_NUMS);
   for (int index = 0; index < MAX_NUMS; index++) {
      printf("Input number %d: ", index);
      scanf("%d", &numbers[index]);
   }

   mean = Average(numbers);
   // name of array represents a pointer to the first element

   printf("The average is %d.\n", mean);
}
```

View of Run-Time Stack



Access the text alternative for slide images.

Notes on Passing Array as Parameter

Array is always passed by reference. This is part of the language specification. There's no way to override this behavior.

Callee function cannot determine the size of the array.
In the callee's activation record, there is only a pointer.

Three options:

- Size is implicit (for example, five cards in a poker hand) or represented by a symbolic value (example, MAX_NUMS).
- Caller passes the size as an additional parameter -- most common.
  ```
  int Average(int data[], int size);
  ```
- Array includes a sentinel value that is used to designate the end of the array. This is common with strings (next...).

String = Array of Characters

There is no native string type in C. Instead, we use an array of characters to hold the string data.

```
char word[10];
```

The array must be large enough to hold the character data, including the null character that terminates the string.

Therefore, the `word` array is large enough to hold a 9-character string.

Take care to distinguish the array that holds the data and the string itself (which is the content of the array).

Initializing a String

First method -- store data to the string array a character at a time.

```
char word[10];   // we want string to be "giraffe"

word[0] = 'g';
word[1] = 'i';
word[2] = 'r';
word[3] = 'a';
word[4] = 'f';
word[5] = 'f';
word[6] = 'e';
word[7] = '\0';  // null terminator
```

Second method -- initialize using a literal string.

```
char word[10] = "giraffe";
```

This only works to **initialize** a string array. **Cannot assign a literal string to an array**, because there is no assignment operator in C that works for arrays.

Basic I/O with Strings

The %s format code is used for strings for both input and output.

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

The `printf` function matches a char array with `%s`, will print character in the array until null `'\0'` is reached.

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

The `scanf` function will read characters until the next white space character, storing them into the array. It will put a terminating null character at the end.

Note: No ampersand! The variable `word` is already a pointer -- it represents the address of the first location in the array.

String Example: Calculate the Length of a String

```
int StringLength(char string[]) {
   int index = 0;

   while (string[index] != '\0') {
      index = index + 1;
   }
   return index;
}
```

The length of a string does not include the null terminator.

String Example: Reversing a String

```
int StringLength(char str[]);  // previous slide
void CharSwap(char *firstChar, char *secondChar);  // NewSwap for
char
void Reverse(char str[]);      // reverse string in place


void Reverse(char string[]) {
   int length;

   length = StringLength(string);

   for (int index = 0; index < length/2; index++)
      CharSwap(&string[index], &string[length-(index+1)]);

}
```

Relationship between Arrays and Pointers

We've noted that an array variable is actually a pointer -- it represents the address of the first element of the array.

It's not actually a pointer variable, because there is no separate memory location that holds the address; it is calculated by the compiler whenever needed.

But can use its value as a pointer, including assigning it to a pointer variable.

```
char word[10];  // array of characters
char *cptr;     // variable that holds a pointer to a char
cptr = word;    // cptr now holds address of first element of word
```

The difference is that I can now change the value stored in `cptr`, because it stores the address in a memory location.

```
cptr++;   // now point to the next array element
```

I can't do the same thing with `word`.

Pointer Notation versus Array Notation

We can always rewrite an array expression as a pointer expression.

We can also rewrite a pointer expression as an array expression, but that only makes sense if the pointer actually points to (or into) an array.

Table 16.1 Expressions with Similar Meanings

	Using a Pointer	Using Name of Array	Using Array Notation
Address of array	`cptr`	`word`	`&word[0]`
0th element	`*cptr`	`*word`	`word[0]`
Address of element n	`(cptr + n)`	`(word + n)`	`&word[n]`
Element n	`*(cptr + n)`	`*(word + n)`	`word[n]`

Advice: Use whichever notation is most "natural" to the problem you're solving. Pointer notation can be more compact, but it might be harder for a reader of the code to understand.

Access the text alternative for slide images.

Pointer Arithmetic

In the previous slide, we show that `cptr+n` is the address of element n of the array: `&word[n]`.

But what if the array holds something other than integers -- something that has a size other than 1? Do we have to do something like `cptr+(2*n)` if each element is two memory locations?

No, we don't.

Pointer arithmetic is not the same as integer arithmetic. Since we know what type of data is being referenced, the compiler will do the appropriate multiplication by the size of the data type.

This means that `cptr+n` works for an array of integers, an array of doubles, or an array of anything.

It also makes the code portable, where an `int` may occupy a different number of memory locations for different architectures. It insulates the programmer from having to know the exact memory sizes for different platforms.

Example of Pointer Notation

One common use of pointer notation is for functions that manipulate strings. Here is the `StringLength` function rewritten using pointers.

Note that the incoming parameter is explicitly typed as a pointer variable. This is not a contradiction with earlier statements -- the parameter is a variable and it contains a pointer, so it can be treated as such.

```
int StringLength(char *string) {
   int length = 0;
   while (*string != '\0') {
      length++;
      string++;  // move to the next character in string
   }
   return length;
}
```

We can also move the incrementing of the pointer inside the test condition and remove the extraneous comparison with zero...

```
   while (*string++) length++;
```

Example: Insertion Sort

We shown an example that sorts the items (integers) in an array using the **insertion sort** algorithm.

The goal is to sort the integers in ascending order. We segregate the array into a sorted part (at the beginning) and an unsorted part (at the end). When we start, the sorted part is only element 0 and elements 1 through N-1 are unsorted; we repeatedly pick the first integer from the unsorted part and insert it into its proper position in the sorted part -- this increases the sorted part by one each iteration.

```
Given an array A with elements 0 through N-1,

For index from 1 to N-1
   item = A[index]
   For pos from index-1 to 0
      If A[pos] > item
         Shift A[pos] to A[pos+1]
      Else
         A[pos] = item, exit inner loop
      End If
   End For
End For
```

Insertion Sort Function

```
void InsertionSort(int list[]) {
   int unsorted;    // index marking beginning of unsorted part
   int sorted;      // index for sorted part
   int unsortedItem;  // temp value for moving value

   for (unsorted = 1; unsorted < MAX_NUMS; unsorted++) {
      unsortedItem = list[unsorted];  // item to be inserted

      // working backwards, shift sorted values to the right
      // until we make reach the position of the new item

      for (sorted = unsorted - 1;
           (sorted >=0) && (list[sorted] > unsortedItem);
           sorted--) {
         list[sorted+1] = list[sorted];  // shift to right
      }

      // after loop, sorted+1 is position where item should go

      list[sorted+1] = unsortedItem;
   }
}
```

Warning: No Bounds Checking on Array Access

C does not check to make sure that the index expression is within the range of elements allocated by the array.

> On Slide 18, there is no step between 3 and 4 that says "make sure the index is non-negative and less than the array size."

If your code goes beyond the array (in either direction), the code generated by the compiler will simply read/write that memory location.

This is a common source of errors in array-based code, and you must take steps to guard against it.

- If the index expression is based on input data from the user, or a parameter passed from another function, you may want to insert an if-statement to check its value.
- If the index is generated locally, make sure it is not possible to create an index outside the proper range.

Variable-Length Arrays

Early versions of C required that the size of an array be known by the compiler when it is declared. This is the only way that the compiler knows how much space to allocate on the stack.

In recent versions (since 1999), you are allowed to declare arrays for which the length is a variable (or derived from a variable). For instance...

```
int Calculate(int len) {
   int data[len];  // length not known until runtime
   ...
}
```

Requires a different way of allocating the array, which will be explained when we discuss *dynamic memory allocation* in later chapters.

Multi-Dimensional Arrays

We can extend the notion of a one-dimensional array to two or more dimensions.

Consider a image captured by a digital camera.

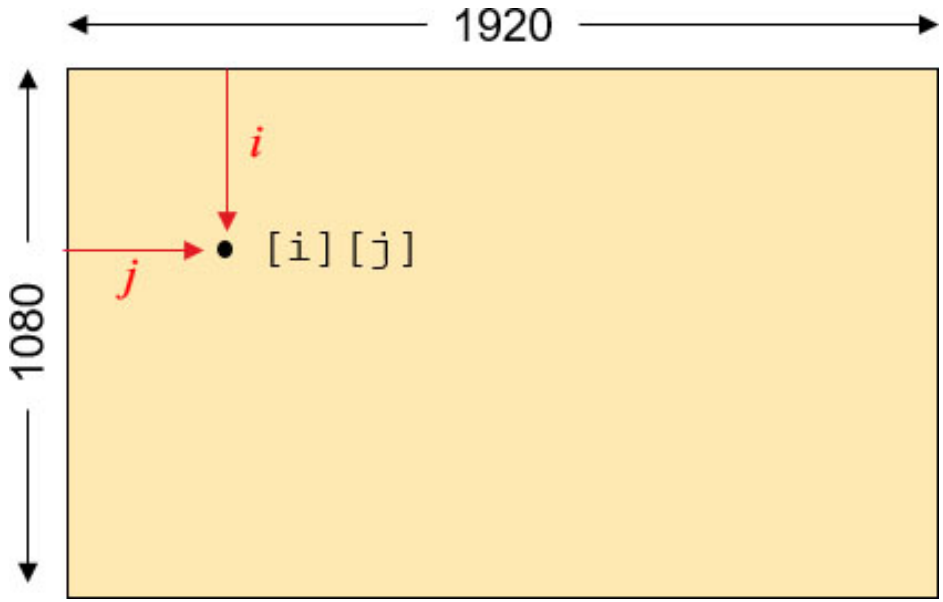
It is helpful to consider a two-dimensional grid of pixels, for example with 1080 rows and 1920 columns.

We can declare a two-dimensional array like this:

```
int image[1080][1920];
```

And we refer to a specific element using two indices instead of one:

```
image[38][283] = 0xFFFF;
// row 38, col 283
```



Access the text alternative for slide images.

Memory Allocation of a Two-Dimensional Array

Even if we think of this a two dimensions, it still must be mapped into a one-dimensional memory.

C uses **row-major** order, in which elements of a column are contiguous in memory. We can think of this as an "array of column arrays".

To calculate the offset of element `[i][j]`:

offset = (i × column_size) + j

This array requires over two million elements, which is too large to fit in the LC-3 memory! In the example to the right, the memory space is extended to 32 bits just to illustrate the concept.

Access the text alternative for slide images.

More than Two Dimensions

We can extend the concept beyond two dimensions:

```
int dataVolume[40][50][60];  // 120,000 elements
```

The memory allocation scheme follows the same pattern.

- Elements which differ by one in the rightmost dimension are in consecutive memory locations.
- Move through indices in right-to-left order.

To calculate the offset of `[i][j][k]` :

offset = $(i \times dim_0 \times dim_1) + (j \times dim_0) + k$

Because learning changes everything.®

www.mheducation.com