chapter

16 Pointers and Arrays

16.1 Introduction

In this chapter, we introduce (actually, reintroduce) two simple but powerful programming constructs: pointers and arrays. We used pointers and arrays when writing LC-3 assembly code. Now, we examine them in the context of C.

A pointer is simply the address of a memory object, such as a variable. With pointers, we can *indirectly* access these objects, which provides some very useful capabilities. For example, with pointers, we can create functions that modify the arguments passed by the caller. With pointers, we can create sophisticated data organizations that grow and shrink (like the run-time stack) during a program's execution.

An array is a list of data arranged sequentially in memory. For example, in a few of the LC-3 examples from the first half of the book, we represented a file of characters as a sequence of characters arranged sequentially in memory. This sequential arrangement of characters is known as an *array* of characters. To access a particular item in an array, we need to specify which element we want. As we'll see, an expression like `a[4]` will access the fifth element in the array named `a`—it is the fifth element because we start numbering the array at element 0. Arrays are useful because they allow us to conveniently process groups of data such as vectors, matrices, lists, and character strings, which are naturally representative of certain objects in the real world.

16.2 Pointers

We begin our discussion of pointers with a classic example of their utility. In the C program in Figure 16.1, the function `Swap` is designed to switch the value of its two arguments. The function `Swap` is called from `main` with the arguments `valueA`, which in this case equals 3, and `valueB`, which equals 4. Once `Swap` returns control to `main`, we expect `valueA` and `valueB` to have their values swapped. However, compile and execute the code and you will notice that the arguments passed to `Swap` remain the same.

Let's examine the run-time stack during the execution of `Swap` to analyze why. Figure 16.2 shows the state of the run-time stack just prior to the completion of the function, just after the statement on line 25 has executed but before control returns to function `main`. Notice that the function `Swap` has modified the local copies of the parameters `firstVal` and `secondVal` within its own activation record. When `Swap` finishes and control returns to `main`, these modified values are lost when the activation record for `Swap` is popped off the stack. The values from `main`'s perspective have not been swapped. We have a buggy program.

In C, arguments are always passed from the caller function to the callee *by value*. C evaluates each argument that appears in a function call as an expression and pushes the value of the expression onto the run-time stack in order to pass them to the function being called. For `Swap` to modify the arguments that the caller

```
 1  #include <stdio.h>
 2
 3  void Swap(int firstVal, int secondVal);
 4
 5  int main()
 6  {
 7    int valueA = 3;
 8    int valueB = 4;
 9
10    printf("Before Swap ");
11    printf("valueA = %d and valueB = %d\n", valueA, valueB);
12
13    Swap(valueA, valueB);
14
15    printf("After Swap ");
16    printf("valueA = %d and valueB = %d\n", valueA, valueB);
17  }
18
19  void Swap(int firstVal, int secondVal)
20  {
21    int tempVal;           /* Holds firstVal when swapping */
22
23    tempVal = firstVal;
24    firstVal = secondVal;
25    secondVal = tempVal;
26  }
```

Figure 16.1 The function `Swap` attempts to swap the values of its two parameters

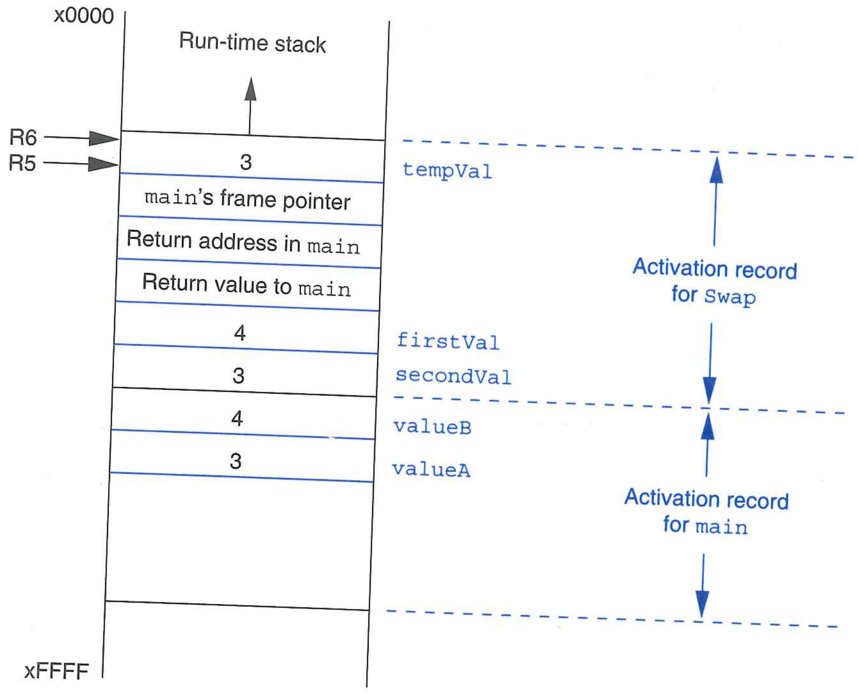

Figure 16.2 A snapshot of the run-time stack when the function Swap is about to return control to main

passes to it, it must have access to the caller function's activation record—it must access the locations at which the arguments are stored in order to modify their values. The function `Swap` needs the *addresses* of `valueA` and `valueB` in `main` in order to change their values. As we shall see in the next few sections, pointers and their associated operators enable this to happen.

16.2.1 Declaring Pointer Variables

A pointer variable contains the address of a memory object, such as a variable. A pointer is said to *point* to the variable whose address it contains. Associated with a pointer variable is the *type* of object to which it points. So, for instance, an integer pointer variable points to an integer variable. To declare a pointer variable in C, we use the following syntax:

```
int *ptr;
```

Here we have declared a variable named `ptr` that points to an integer. The asterisk (`*`) indicates that the identifier that follows is a pointer variable. C programmers will often say that `ptr` is of type `int` *star*. Similarly, we can declare

```
char *cp;
double *dp;
```

The variable `cp` points to a character and `dp` points to a double-precision floating point number. Pointer variables are initialized in a manner similar to all other variables. If a pointer variable is declared as a local variable, it will not be initialized automatically.

The syntax of declaring a pointer variable using `*` may seem a bit odd at first, but once we have gone through the pointer operators, the rationale behind the syntax will be more clear.

16.2.2 Pointer Operators

C has two operators for pointer-related manipulations, the address operator `&` and the indirection operator `*`.

The Address Operator &

The address operator, whose symbol is an ampersand, `&`, generates the memory address of its operand, which must be a memory object such as a variable. In the following code sequence, the pointer variable `ptr` will point to the integer variable `object`. The expression on the right-hand side of the second assignment statement generates the memory address of `object`.

```
int object;
int *ptr;

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

Let's examine the LC-3 code for this sequence. Both declared variables are locals and are allocated on the stack. Recall that R5, the base pointer, points to the first declared local variable, or `object` in this case.

```
AND   R0, R0, #0    ;   Clear R0
ADD   R0, R0, #4    ;   R0 = 4
STR   R0, R5, #0    ;   Object = 4;

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

Figure 16.3 shows the activation record of the function containing this code after the statement `ptr = &object;` has executed. In order to make things more concrete, each memory location is labeled with an address, which we've arbitrarily selected to be in the xEFF0 range. The base pointer R5 currently points to xEFF2. Notice that `object` contains the integer value 4 and `ptr` contains the memory address of `object`.

The Indirection Operator *

The second pointer operator is called the *indirection*, or *dereference*, operator, and its symbol is the asterisk, `*` (pronounced *star* in this context). This operator allows us to indirectly manipulate the value of a memory object. For example, the

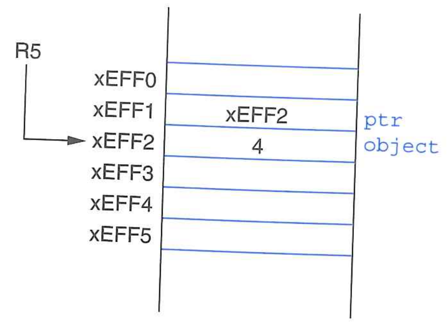

Figure 16.3 The run-time stack frame containing `object` and `ptr` after the statement `ptr = &object` has executed

expression `*ptr` refers to the value pointed to by the pointer variable `ptr`. Recall the previous example: `*ptr` refers to the value stored in variable `object`. Here, `*ptr` and `object` can be used interchangeably. Adding to the previous C code example,

```
int object;
int *ptr;

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

Essentially, `*ptr = *ptr + 1;` is another way of saying `object = object + 1;`. Just as with other types of variables we have seen, the `*ptr` means different things depending on which side of the assignment operator it appears on. On the right-hand side of the assignment operator, it refers to the value that appears at that location (in this case the value 4). On the left-hand side, it specifies the location that gets modified (in this case, the address of `object`). Let's examine the LC-3 code for the last statement in the preceding code.

```
LDR  R0, R5, #-1   ;   R0 contains the value of ptr
LDR  R1, R0, #0    ;   R1 <- *ptr
ADD  R1, R1, #1    ;   *ptr + 1
STR  R1, R0, #0    ;   *ptr = *ptr + 1;
```

Notice that this code is different from what would get generated if the final C statement had been `object = object + 1;`. With the pointer dereference, the compiler generates two `LDR` instructions for the indirection operator on the right-hand side, one to load the memory address contained in `ptr` and another to get the value stored at that address. With the dereference on the left-hand side, the compiler generates a `STR R1, R0, #0`. Had the statement been `object = *ptr + 1;`, the compiler would have generated `STR R1, R5, #0`.

16.2.3 Passing a Reference Using Pointers

Using the address and indirection operator, we can repair the `Swap` function from Figure 16.1 that did not quite accomplish the swap of its two input parameters. Figure 16.4 lists the same program with a revised version of `Swap` called `NewSwap`.

The first modification we've made is that the parameters of `NewSwap` are no longer integers but are now pointers to integers (`int *`). These two parameters are the memory addresses of the two variables that are to be swapped. Within the function body of `NewSwap`, we use the indirection operator `*` to obtain the values that these pointers point to.

Now when we call `NewSwap` from `main`, we need to supply the *memory addresses* for the two variables we want swapped, rather than the *values* of the variables as we did in the previous version of the code. For this, the `&` operator does the trick. Figure 16.5 shows the run-time stack when various statements of the function `NewSwap` are executed. The three subfigures (A–C) correspond to the run-time stack after lines 23, 24, and 25 execute.

By design, C passes information from the caller function to the callee by value: that is, each argument expression in the call statement is evaluated, and the resulting value is passed to the callee via the run-time stack. However, in `NewSwap` we created a *call by reference* for the two arguments by using the address

```
 1  #include <stdio.h>
 2
 3  void NewSwap(int *firstVal, int *secondVal);
 4
 5  int main()
 6  {
 7    int valueA = 3;
 8    int valueB = 4;
 9
10    printf("Before Swap ");
11    printf("valueA = %d and valueB = %d\n", valueA, valueB);
12
13    NewSwap(&valueA, &valueB);
14
15    printf("After Swap ");
16    printf("valueA = %d and valueB = %d\n", valueA, valueB);
17  }
18
19  void NewSwap(int *firstVal, int *secondVal)
20  {
21    int tempVal;            /* Holds firstVal when swapping */
22
23    tempVal = *firstVal;
24    *firstVal = *secondVal;
25    *secondVal = tempVal;
26  }
```

Figure 16.4 The function `NewSwap` swaps the values of its two parameters

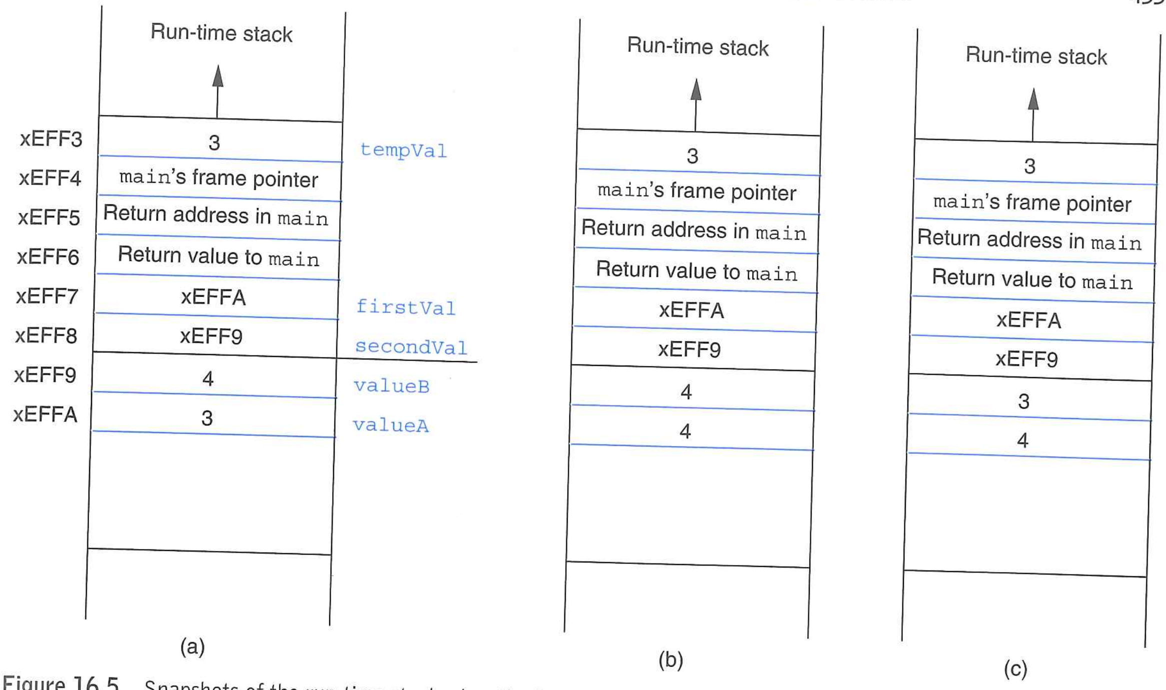

Figure 16.5 Snapshots of the run-time stack when the function `NewSwap` executes the statements in (a) line 23, (b) line 24, (c) line 25.

operator `&`. When an argument is passed as a reference, its **address** is passed to the callee function—for this to be valid, the argument must be a variable or other memory object (i.e., it must have an address). The callee function then can use the indirection operator `*` to access (and modify) the original value of the object.

16.2.4 Null Pointers

Sometimes it is convenient for us to say that a pointer points to nothing. Why such a concept is useful will be eminently clear to you when we discuss dynamic data structures such as linked lists in Chapter 19. For now, let us say that a pointer that points to nothing is a *null* pointer. In C, we make this designation with the following assignment:

```
int *ptr;

ptr = NULL;
```

Here, we are assigning the value of `NULL` to the pointer variable `ptr`. In C, `NULL` is a specially defined preprocessor macro that contains a value that no pointer should ever hold unless it is null. For example, `NULL` might equal 0 on a particular system because no valid memory object can exist at location 0.

16.2.5 Demystifying the Syntax

It is now time to revisit some notation that we introduced in Chapter 11. Now that we know how to pass a reference, let's reexamine the I/O library function `scanf`:

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

Since function `scanf` needs to update the variable `input` with the decimal value read from the keyboard, `scanf` needs the address of `input` and not its value. Thus, the address operator `&` is required. If we omit the address operator, the program terminates with an error. Can you come up with a plausible reason why this happens? Why is it not possible for `scanf` to work correctly without the use of a reference?

Before we complete our introduction to pointers, let's attempt to make sense of the pointer declaration syntax. To declare a pointer variable, we use a declaration of the following form:

```
    type *ptr;
```

where `type` can be any of the predefined (or programmer-defined) types such as `int`, `char`, `double`, and so forth. The name `ptr` is simply any legal variable identifier. With this declaration, we are declaring a variable that, when the `*` (dereference) operator is applied to it, generates a variable of type `type`. That is, `*ptr` is of type `type`.

We can also declare functions to return a pointer type (why we would want to do so will be more apparent in later chapters). For example, we can declare a function using a declaration of the form `int *MaxSwap()`.

As with all other operators, the address and indirection operator are evaluated according to the C precedence and associativity rules. The precedence and associativity of these and all other operators is listed in Table 12.5. Notice that both of the pointer operators have very high precedence.

16.2.6 An Example Problem Involving Pointers

Let's examine an example problem involving pointers. Say we want to develop a program that calculates the quotient and remainder given an integer dividend and integer divisor. That is, the program will calculate *dividend / divisor* and *dividend % divisor* where both values are integers. The structure of this program is very simple and requires only sequential constructs—that is, iteration is not required. The twist, however, is that we want the calculation of quotient and remainder to be performed by a single C function.

We can easily construct a function to generate a single output value (say, quotient) that we can pass back to the caller using the return value mechanism. A function that calculates only the quotient, for example, could consist of the single statement `return dividend / divisor;`. To provide the caller with multiple values, however, we will make use of the call by reference mechanism using pointer variables.

The code in Figure 16.6 contains a function that does just so. The function `IntDivide` takes four parameters, two of which are integers and two of which

```
#include <stdio.h>

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

int main()
{
  int dividend;          /* The number to be divided      */
  int divisor;           /* The number to divide by       */
  int quotient;          /* Integer result of division    */
  int remainder;         /* Integer remainder of division */
  int error;             /* Did something go wrong?       */

  printf("Input dividend: ");
  scanf("%d", &dividend);
  printf("Input divisor: ");
  scanf("%d", &divisor);

  error = IntDivide(dividend,divisor,&quotient,&remainder);

  if (!error)            /* !error indicates no error     */
    printf("Answer: %d remainder %d\n", quotient, remainder);
  else
    printf("IntDivide failed.\n");
}

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

Figure 16.6 The function `IntDivide` calculates the integer portion and remainder of an integer divide; it returns a −1 if the divisor is 0

are pointers to integers. The function divides the first parameter `x` by the second parameter `y`. The integer portion of the result is assigned to the memory location pointed to by `quoPtr`, and the integer remainder is assigned to the memory location pointed to by `remPtr`.

Notice that the function `IntDivide` also returns a value to indicate its status: It returns a −1 if the `divisor` is zero, indicating to the caller that an error has occurred. It returns a zero otherwise, indicating to the caller that the computation proceeded without a hitch. The function `main`, upon return, checks the return value to determine if the values in quotient and remainder are correct. Using the return value to signal a problem during a function call between caller and callee is an excellent defensive programming practice for conveying error conditions across a call.

16.3 Arrays

Consider a program that keeps track of the final exam scores for each of the 50 students in a computer engineering course. The most convenient way to store this data would be to declare a single object, say `examScore`, in which we can store 50 different integer values. We can access a particular exam score within this object using an *index* that is an offset from the beginning of the object. For example, `examScore[32]` provides the exam score for the 33rd student (the very first student's score stored in `examScore[0]`). The object `examScore` in this example is an *array* of integers. An array is a collection of similar data items that are stored sequentially in memory. Specifically, all the elements in the array are of the same type (e.g., `int`, `char`, etc.).

Arrays are most useful when the data upon which the program operates is naturally expressed as a contiguous sequence of values. Because a lot of real-world data falls into this category (such as exam scores for students in a course), arrays are incredibly useful data structures. For instance, if we wanted to write a program to take a sequence of 100 numbers entered from the keyboard and *sort* them into ascending order, then an array would be the natural choice for storing these numbers in memory. The program would be almost impossible to write using the simple variables we have been using thus far.

16.3.1 Declaring and Using Arrays

First, let's examine how to declare an array in a C program. Like all other variables, arrays must have a type associated with them. The type indicates the properties of the values stored in the array. Following is a declaration for an array of 10 integers:

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

The keyword `int` indicates that we are declaring something of type integer. The name of the array is `grid`. The brackets indicate we are declaring an array and the 10 indicates that the array is to contain 10 integers, all of which will be sequentially located in memory. Figure 16.7 shows a pictorial representation of how `grid` is allocated. The first element, `grid[0]`, is allocated in the lowest memory address and the last element, `grid[9]`, in the highest address. If the array `grid` were a local variable, then its memory space would be allocated on the run-time stack.

Let's examine how to access different values in this array. Notice in Figure 16.7 that the array's first element is actually numbered 0, which means the last element is numbered 9. To access a particular element, we provide an *index* within brackets. For example,

```
grid[6] = grid[3] + 1;
```

The statement reads the value stored in the fourth (remember, we start numbering with 0) element of `grid`, adds 1 to it, and stores the result into the seventh element of `grid`. Let's look at the LC-3 code for this example. Let's say that `grid` is the only local variable allocated on the run-time stack. This means that the base pointer R5 will point to `grid[9]`.

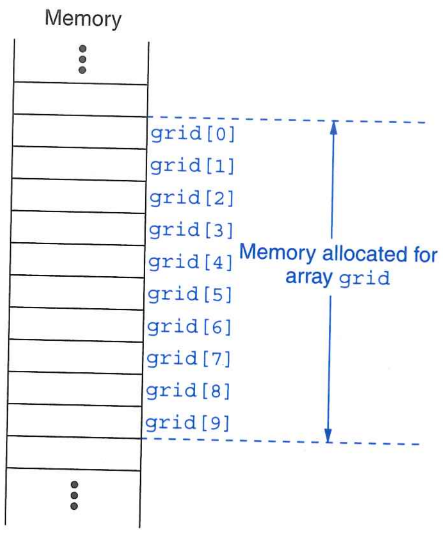

Figure 16.7 The array grid allocated in memory

```
ADD  R0, R5, #-9  ;  Put the base address of grid into R0
LDR  R1, R0, #3   ;  R1 <-- grid[3]
ADD  R1, R1, #1   ;  R1 <-- grid[3] + 1
STR  R1, R0, #6   ;  grid[6] = grid[3] + 1;
```

Notice that the first instruction calculates the base address of the array, which is the address of grid[0], and puts it into R0. The base address of an array in general is the address of the first element of the array. We can access any element in the array by adding the index of the desired element to the base address.

The power of arrays comes from the fact that an array's index can be any legal C expression of integer type. The following example demonstrates:

```
grid[x+1] = grid[x] + 2;
```

Let's look at the LC-3 code for this statement. Assume x is another local variable allocated on the run-time stack directly on top of the array grid.

```
LDR  R0, R5, #-10 ;  Load the value of x
ADD  R1, R5, #-9  ;  Put the base address of grid into R1
ADD  R1, R0, R1   ;  Calculate address of grid[x]
LDR  R2, R1, #0   ;  R2 <-- grid[x]
ADD  R2, R2, #2   ;  R2 <-- grid[x] + 2

LDR  R0, R5, #-10 ;  Load the value of x
ADD  R0, R0, #1   ;  R0 <-- x + 1
ADD  R1, R5, #-9  ;  Put the base address of grid into R1
ADD  R1, R0, R1   ;  Calculate address of grid[x+1]
STR  R2, R1, #0   ;  grid[x+1] = grid[x] + 2;
```

16.3.2 Examples Using Arrays

We start off with a simple C program that adds two arrays together by adding the corresponding elements from each array to form the sum. Each array represents a list of exam scores for students in a course. Each array contains an element for each student's score. To generate the cumulative points for each student, we effectively want to perform `Total[i] = Exam1[i] + Exam2[i]`. Figure 16.8 contains the C code to read in two 10-element integer arrays, add them together into another 10-element array, and print out the sum.

A style note: Notice the use of the preprocessor macro `NUM_STUDENTS` to represent a constant value of the size of the input set. This is a common use for preprocessor macros, which are usually found at the beginning of the source file (or within C header files). Now, if we want to increase the size of the array, for example if the student enrollment changes, we simply change the definition of

```c
#include <stdio.h>
#define NUM_STUDENTS 10

int main()
{
  int i;
  int Exam1[NUM_STUDENTS];
  int Exam2[NUM_STUDENTS];
  int Total[NUM_STUDENTS];

  /* Input Exam 1 scores */
  for (i = 0; i < NUM_STUDENTS; i++) {
    printf("Input Exam 1 score for student %d : ", i);
    scanf("%d", &Exam1[i]);
  }
  printf("\n");

  /* Input Exam 2 scores */
  for (i = 0; i < NUM_STUDENTS; i++) {
    printf("Input Exam 2 score for student %d : ", i);
    scanf("%d", &Exam2[i]);
  }
  printf("\n");

  /* Calculate Total Points */
  for (i = 0; i < NUM_STUDENTS; i++) {
    Total[i] = Exam1[i] + Exam2[i];
  }

  /* Output the Total Points */
  for (i = 0; i < NUM_STUDENTS; i++) {
    printf("Total for Student %d = %d\n", i, Total[i]);
  }
}
```

Figure 16.8 A C program that calculates the sum of two 10-element arrays

the macro (one change) and recompile the program. If we did not use the macro, changing the array size would require changes to the code in multiple places. The changes could be potentially difficult to track down, and forgetting to do one would likely result in a program that did not work correctly. Using preprocessor macros for the size of an array is good programming practice.

Now onto a slightly more complex example involving arrays. Figure 16.9 lists a C program that reads in a sequence of decimal numbers (in total MAX_NUMS of them) from the keyboard and determines the number of times each input number is repeated within the sequence. The program then prints out each number, along with the number of times it repeats.

In this program, we use two arrays, numbers and repeats. Both are declared to contain MAX_NUMS integer values. The array numbers stores the input sequence. The array repeats is calculated by the program to contain the number of times the corresponding element in numbers is repeated in the input sequence. For example, if numbers[3] equals 115, and there are a total of four 115s in the input

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

int main()
{
  int index;                 /* Loop iteration variable     */
  int repIndex;              /* Loop variable for rep loop  */
  int numbers[MAX_NUMS];     /* Original input numbers      */
  int repeats[MAX_NUMS];     /* Number of repeats           */

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

  /* Scan through entire array, counting number of          */
  /* repeats per element within the original array          */
  for (index = 0; index < MAX_NUMS; index++) {
    repeats[index] = 0;
    for (repIndex = 0; repIndex < MAX_NUMS; repIndex++) {
      if (numbers[repIndex] == numbers[index])
        repeats[index]++;
    }
  }

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

Figure 16.9 A C program that determines the number of repeated values in an array

sequence (i.e., there are four 115s in the array `numbers`), then `repeats[3]` will equal 4.

This program consists of three outer loops, of which the middle loop is actually a *nested loop* (see Section 13.3.2) consisting of two loops. The first and last `for` loops are simple loops that get keyboard input and produce program output.

The middle `for` loop contains the nested loop. This body of code determines how many copies of each element exist within the entire array. The outer loop iterates the variable `index` from 0 through `MAX_NUMS`; we use `index` to scan through the array from the first element `numbers[0]` through the last element `numbers[MAX_NUMS]`. The inner loop also iterates from 0 through `MAX_NUMS`; we use this loop to scan through the array again, this time determining how many of the elements match the element selected by the outer loop (i.e., `numbers[index]`). Each time a copy is detected (i.e., `numbers[repIndex] == numbers[index]`), the corresponding element in the `repeats` array is incremented (i.e., `repeats[index]++`).

16.3.3 Arrays as Parameters

Passing arrays between functions is a useful thing because it allows us to create functions that operate on arrays. Say we want to create a set of functions that calculates the mean and median on an array of integers. We would need either (1) to pass the entire array of values from one function to another or (2) to pass a reference to the array. If the array contains a large number of elements, copying each element from one activation record onto another could be very costly in execution time. Fortunately, C naturally passes arrays by reference. Figure 16.10 is a C program that contains a function `Average` whose single parameter is an array of integers.

When calling the function `Average` from `main`, we pass to it the value associated with the array identifier `numbers`. Notice that here we are not using the standard notation involving brackets `[ ]` that we normally use for arrays. In C, an array's name refers to the address of the base element of the array. The name `numbers` is equivalent to `&numbers[0]`. The type `numbers` is similar to `int *`. It is an address of memory location containing an integer.

In using `numbers` as the argument to the function `Average`, we are causing the address of the array `numbers` to be pushed onto the stack and passed to the function `Average`. Within the function `Average`, the parameter `inputValues` is assigned the address of the array. Within `Average` we can access the elements of the original array using standard array notation. Figure 16.11 shows the run-time stack just prior to the execution of the `return` from `Average` (line 34 of the program).

Notice how the input parameter `inputValues` is specified in the declaration of the function `Average`. The brackets `[ ]` indicate to the compiler that the corresponding parameter will be the base address to an array of the specified type, in this case an array of integers.

Since arrays are passed by reference in C, any modifications to the array values made by the called function will be visible to the caller once control returns to it. How would we go about passing only a single element of an array by value? How about by reference?

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

int Average(int input_values[]);

int main()
{
  int index;                    /* Loop iteration variable  */
  int mean;                     /* Average of numbers       */
  int numbers[MAX_NUMS];        /* Original input numbers   */


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

  mean = Average(numbers);

  printf("The average of these numbers is %d\n", mean);
}

int Average(int inputValues[])
{
  int index;
  int sum = 0;

  for (index = 0; index < MAX_NUMS; index++) {
    sum = sum + inputValues[index];
  }

  return (sum / MAX_NUMS);
}
```

Figure 16.10 An example of an array as a parameter to a function

16.3.4 Strings in C

A very common use for arrays in C is for *strings*. Strings are sequences of characters that represent text. Strings are simply character arrays, with each subsequent element containing the next character of the string. For example,

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

declares an array that can store a string of up to 10 characters. Longer strings require a larger array. What if the string is shorter than 10 characters? In C and many other modern programming languages, the end of a string is denoted by the null character whose ASCII value is 0. It is a sentinel that identifies the end of the string. Such strings are also called *null-terminated strings*. `'\0'` is the special

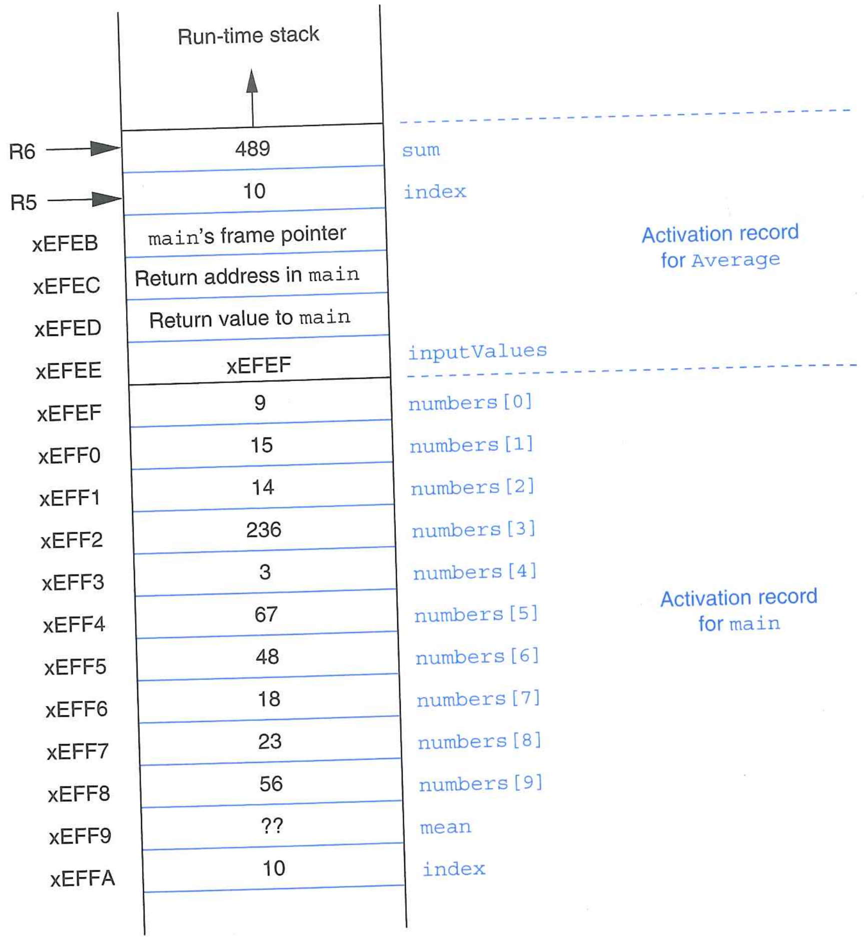

Figure 16.11 The run-time stack prior to the execution of the return from `Average`

sequence that corresponds to the null character. Continuing with our previous declaration,

```
char word[10];

word[0]  = 'H';
word[1]  = 'e';
word[2]  = 'l';
word[3]  = 'l';
word[4]  = 'o';
word[5]  = '\0';

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

Here, we are assigning each element of the array individually. The array will contain the string "Hello." Notice that the end-of-string character itself is a character that occupies an element of the array. Even though the array is declared for 10 elements, we must reserve one element for the null character, and therefore strings that are longer than nine characters cannot be stored in this array.

We have also used a new `printf` format specification `%s` in this example. This specification prints out a string of characters, starting with the character pointed to by the corresponding parameter and ending at the end-of-string character `'\0'`.

ANSI C compilers also allow strings to be initialized within their declarations. For instance, the preceding example can be rewritten to the following.

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

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

Make note of two things here: First, character strings are distinguished from single characters with double quotes, `" "`. Single quotes are used for single characters, such as `'A'`. Second, notice that the compiler automatically adds the null character to the end of the string.

Examples of Strings

Figure 16.12 contains a program that performs a very simple and useful primitive operation on strings: it calculates the length of a string. Since the size of the array that contains the string does not indicate the actual length of the string (it does, however, tell us the maximum length of the string), we need to examine the string itself to calculate its length.

The algorithm for determining string length is easy. Starting with the first element, we count the number of characters before we encounter the null character. The function `StringLength` in the code in Figure 16.12 performs this calculation.

Notice that we are using the format specification `%s` in the `scanf` statement. This specification causes `scanf` to read in a string of characters from the keyboard until the first *white space* character. In C, any space, tab, new line, carriage return, vertical tab, or form-feed character is considered white space. So if the user types (from *The New Colossus*, by Emma Lazarus)

```
Not like the brazen giant of Greek fame,
With conquering limbs astride from land to land;
```

only the word *Not* is stored in the array `input`. The remainder of the text line is reserved for subsequent `scanf` calls to read. So if we performed another `scanf("%s", input)`, the word *like* will be stored in the array `input`. Notice that the white space is automatically discarded by this `%s` specification. We examine this I/O behavior more closely in Chapter 18 when we take a deeper look into I/O in C.

Notice that the maximum word size is 20 characters. What happens if the first word is longer? The `scanf` function has no information on the size of the array `input` and will keep storing characters to the array address it was provided until white space is encountered. So what then happens if the first word is longer

```
#include <stdio.h>
#define MAX_STRING 20

int StringLength(char string[]);

int main()
{
  char input[MAX_STRING];     /* Input string */
  int  length = 0;

  printf("Input a word (less than 20 characters): ");
  scanf("%s", input);

  length = StringLength(input);
  printf("The word contains %d characters\n", length);
}

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

  while (string[index] != '\0')
    index = index + 1;

  return index;
}
```

Figure 16.12 A program that calculates the length of a string

than 20 characters? Any local variables that are allocated after the array `input` in the function `main` will be overwritten. Draw out the activation record before and after the call to `scanf` to see why. In the exercises at the end of this chapter, we provide a problem where you need to modify this program in order to catch the scenario where the user enters a word longer than what fits into the `input` array.

Let's examine a slightly more complex example that uses the `StringLength` function from the previous code example. In this example, listed in Figure 16.13, we read an input string from the keyboard using `scanf`, then call a function to reverse the string. The reversed string is then displayed on the output device.

The function `Reverse` performs two tasks in order to reverse the string properly. First it determines the length of the string to reverse using the `StringLength` function from the previous code example. Then it performs the reversal by swapping the first character with the last, the second character with the second to last, the third character with the third to last, and so on.

To perform the swap, it uses a modified version of the `NewSwap` function from Figure 16.4. The reversal loop calls the function `CharSwap` on pairs of characters within the string. First, `CharSwap` is called on the first and last character, then on the second and second to last character, and so forth.

The C standard library provides many prewritten functions for strings. For example, functions to copy strings, merge strings together, compare them, or

```
#include <stdio.h>
#define MAX_STRING 20

int StringLength(char string[]);
void CharSwap(char *firstVal, char *secondVal);
void Reverse(char string[]);

int main()
{
  char input[MAX_STRING];                    /* Input string */

  printf("Input a word (less than 20 characters): ");
  scanf("%s", input);

  Reverse(input);
  printf("The word reversed is %s.\n", input);
}

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

  while (string[index] != '\0')
    index = index + 1;

  return index;
}

void CharSwap(char *firstVal, char *secondVal)
{
  char tempVal;    /* Temporary location for swapping */

  tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}

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

  length = StringLength(string);

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

Figure 16.13 A program that reverses a string

Table 16.1 The Relationship Between Pointers and Arrays

`cptr`	`word`	`&word[0]`
`(cptr + n)`	`word + n`	`&word[n]`
`*cptr`	`*word`	`word[0]`
`*(cptr + n)`	`*(word + n)`	`word[n]`

calculate their length can be found in the C standard library, and the declarations for these functions can be included via the `<string.h>` header file. More information on some of these string functions can be found in Appendix D.9.2.

16.3.5 The Relationship Between Arrays and Pointers in C

You might have noticed that there is a similarity between an array's name and a pointer variable to an element of the same type as the array. For instance,

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

cptr = word;
```

is a legal, and sometimes useful, sequence of code. Here, we have assigned the pointer variable `cptr` to point to the base address of the array `word`. Because they are both pointers to characters, `cptr` and `word` can be used interchangeably. For example, we can access the fourth character within the string either by using `word[3]` or `*(cptr + 3)`.

One difference between the two, though, is that `cptr` is a variable and can be reassigned. The array identifier `word`, on the other hand, cannot be. For example, the following statement is illegal: `word = newArray`. The identifier always points to a fixed spot in memory where the compiler has placed the array. Once it has been allocated, it cannot be moved.

Table 16.1 shows the equivalence of several expressions involving pointer and array notation. Rows in the table are expressions with the same meaning.

16.3.6 Problem Solving: Insertion Sort

With this initial exposure to arrays under our belt, we can now attempt an interesting and sizeable (and useful!) problem: we will write C code to *sort* an array of integers into ascending order. That is, the code arranges the array `a[]` such that `a[0]` $\le$ `a[1]` $\le$ `a[2]` `....`

To accomplish this, we will use an algorithm for sorting called Insertion Sort. Sorting is an important primitive operation, and people in computing have devoted considerable time to understanding, analyzing, and refining the sorting process. As a result, there are many algorithms for sorting, and you will gain exposure to some basic techniques in subsequent computing courses. We use insertion sort here because it parallels how we might sort items in the real world. It is quite straightforward.

Insertion sort is best described by an example. Say you want to sort your compact disc collection into alphabetical order by artist. If you were sorting your compact discs using insertion sort, you would split the CDs into two groups, the sorted group and the unsorted group. Initially, the sorted group would be empty as all your CDs would be yet unsorted. The sorting process proceeds by taking a CD from the unsorted group and *inserting* it into the proper position among the sorted CDs. For example, if the sorted group contained three CDs, one by John Coltrane, one by Charles Mingus, and one by Thelonious Monk, then inserting the Miles Davis CD would mean inserting it between the Coltrane CD and the Mingus CD. You keep doing this until all CDs in the unsorted group have been inserted into the sorted group. This is insertion sort.

How would we go about applying this same technique to sort an array of integers? Applying systematic decomposition to the preceding algorithm, we see that the core of the program involves iterating through the elements of the array, inserting each element into the proper spot in a new array where all items are in ascending order. This process continues until all elements of the original array have been inserted into the new array. Once done, the new array will contain the same elements as the first array, except in sorted order.

For this technique we basically need to represent two groups of items, the original unsorted elements and the sorted elements. And for this we could use two separate arrays. It turns out, however, that we can represent both groups of elements within the original array. Doing so results in code that requires less memory and is more compact, though slightly more complex upon first glance. The initial part of the array contains the sorted elements and the remainder of the array contains the unsorted elements. We pick the next unsorted item and insert it into the sorted part at the correct point. We keep doing this until we have gone through the entire array.

The actual `InsertionSort` routine (shown in Figure 16.14) contains a nested loop. The outer loop scans through all the unsorted items (analogous to going through the unsorted CDs, one by one). The inner loop scans through the already sorted items, scanning for the place at which to insert the new item. Once we detect an already sorted element that is larger than the one we are inserting, we insert the new element between the larger and the one before it.

Let's take a closer look by examining what happens during a pass of the insertion sort. Say we examine the insertion sort process (lines 33–43) when the variable `unsorted` is equal to 4. The array `list` contains the following 10 elements:

```
2 16 69 92 15 37 92 38 82 19
```

During this pass, the code inserts `list[4]`, or 15, into the already sorted portion of the array, elements `list[0]` through `list[3]`.

The inner loop iterates the variable `sorted` through the list of already sorted elements. It does this from the highest numbered element down to 0 (i.e., starting at 3 down to 0). Notice that the condition on the `for` loop terminates the loop once a list item *less* than the current item, 15, is found.

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

void InsertionSort(int list[]);

int main()
{
  int index;                /* Iteration variable           */
  int numbers[MAX_NUMS];    /* List of numbers to be sorted */

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

  InsertionSort(numbers);  /* Call sorting routine        */

  /* Print sorted list */
  printf("\nThe input set, in ascending order:\n");
  for (index = 0; index < MAX_NUMS; index++)
    printf("%d\n", numbers[index]);
}

void InsertionSort(int list[])
{
  int unsorted;             /* Index for unsorted list items */
  int sorted;               /* Index for sorted items        */
  int unsortedItem;         /* Current item to be sorted     */

  /* This loop iterates from 1 thru MAX_NUMS  */
  for (unsorted = 1; unsorted < MAX_NUMS; unsorted++) {
    unsortedItem = list[unsorted];

    /* This loop iterates from unsorted thru 0, unless
       we hit an element smaller than current item */
    for (sorted = unsorted - 1;
         (sorted >= 0) && (list[sorted] > unsortedItem);
         sorted--)
      list[sorted + 1] = list[sorted];

    list[sorted + 1] = unsortedItem; /* Insert item      */
  }
}
```

Figure 16.14 Insertion sort program

In each iteration of this inner loop (lines 38–41), an element in the sorted part of the array is copied to the next position in the array. In the first iteration, `list[3]` is copied to `list[4]`. So after the first iteration of the inner loop, the array `list` contains

```
2 16 69 92 92 37 92 38 82 19
```

Notice that we have overwritten 15 (`list[4]`). This is OK because we have a copy of its value in the variable `unsortedItem` (from line 34). The second iteration performs the same operation on `list[2]`. After the second iteration, `list` contains

```
2 16 69 69 92 37 92 38 82 19
```

After the third iteration, `list` contains:

```
2 16 16 69 92 37 92 38 82 19
```

Now the `for` loop terminates because the evaluation condition is no longer true. More specifically, `list[sorted] > unsortedItem` is not true. The current sorted list item `list[0]`, which is 2, is not larger than the current unsorted item `unsortedItem`, which is 15. Now the inner loop terminates, and the statement following it, `list[sorted + 1] = unsortedItem;` executes. Now `list` contains, and the sorted part of the array contains, one more element.

```
2 15 16 69 92 37 92 38 82 19
```

This process continues until all items have been sorted, meaning the outer loop has iterated through all elements of the array `list`.

16.3.7 Common Pitfalls with Arrays in C

Unlike some other modern programming languages, C does not provide protection against exceeding the size (or bounds) of an array. It is a common error made with arrays in C programming. C provides no support for ensuring that an array index is actually within an array. The compiler blindly generates code for the expression `a[i]`, even if the index `i` accesses a memory location beyond the end of the array. To demonstrate, the code in Figure 16.15 lists an example of how exceeding the array bounds can lead to a serious debugging effort. Enter a number larger than the array size and this program exhibits some peculiar behavior.[1]

Analyze this program by drawing out the run-time stack and you will see more clearly why this bug causes the behavior it does.

C does not perform bounds checking on array accesses. C code tends to be faster because array accesses incur less overhead. This is yet another manner in

[1] Depending on the compiler you are using, you might need to enter a number larger than 16, or you might need to declare `index` after `array` in order to observe the problem.

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

int main()
{
  int index;
  int array[MAX_SIZE];
  int limit;

  printf("Enter limit (integer): ");
  scanf("%d", &limit);

  for(index = 0; index < limit; index++) {
    array[index] = 0;
    printf("array[%d] is set to 0\n", index);
  }
}
```

Figure 16.15 This C program has peculiar behavior if the user enters a number that is too large

which C provides more control to the programmer than other languages. If you are not careful in your coding, this bare-bones philosophy can, however, lead to undue debugging effort. To counter this, experienced C programmers often use some specific defensive programming techniques when it comes to arrays.

Another common pitfall with arrays in C revolves around the fact that arrays (in particular, statically declared arrays such as the ones we've seen) must be of a fixed size. We must know the size of the array when we compile the program. C does not support array declarations with variable expressions. The following code in C is illegal. The size of array `temp` must be known when the compiler analyzes the source code.

```
void SomeFunction(int num_elements)
{
  int temp[num_elements];   /* Generates a syntax error */

  :
}
```

To deal with this limitation, experienced C programmers carefully analyze the situations in which their code will be used and then allocate arrays with ample space. To supplement this built-in assumption in their code, bounds checks are added to warn if the size of the array is not sufficient. Another option is to use dynamic memory allocation to allocate the array at run-time. More on this in Chapter 19.

16.4 Summary

In this chapter we covered two important high-level programming constructs: pointers and arrays. Both constructs enable us to access memory *indirectly*. The key notions we covered in this chapter are:

- **Pointers.** Pointers are variables that contain addresses of other memory objects (such as other variables). With pointers we can indirectly access and manipulate these other objects. A very simple application of pointers is to use them to pass parameters by reference. Pointers have more substantial applications, and we will see them in subsequent chapters.

- **Arrays.** An array is a collection of elements of the same type arranged sequentially in memory. We can access a particular element within an array by providing an index to the element that is its offset from the beginning of the array. Many real-world objects are best represented within a computer program as an array of items, thus making the array a significant structure for organizing data. With arrays, we can represent character strings that hold text data, for example. We examine several important array operations, including the sorting operation via insertion sort.

Exercises

16.1 Write a C function that takes as a parameter a character string of unknown length, containing a single word. Your function should translate this string from English into Pig Latin. This translation is performed by removing the first letter of the string, appending it onto the end, and concatenating the letters *ay*. You can assume that the array contains enough space for you to add the extra characters.

For example, if your function is passed the string "Hello," after your function returns, the string should have the value "elloHay." The first character of the string should be "e."

16.2 Write a C program that accepts a list of numbers from the user until a number is repeated (i.e., is the same as the number preceding it). The program then prints out the number of numbers entered (excluding the last) and their sum. When the program is run, the prompts and responses will look like the following:

```
Number: 5
Number: -6
Number: 0
Number: 45
Number: 45
4 numbers were entered and their sum is 44
```

16.3 What is the output when the following code is compiled and run?

```
int x;

int main()
{
   int *px = &x;
   int x = 7;

   *px = 4;
   printf("x = %d\n", x);
}
```

16.4 Create a string function that takes two input strings, `stringA` and `stringB`, and returns a 0 if both strings are the same, a 1 if `stringA` appears before `stringB` in the sorted order of a dictionary, or a 2 if `stringB` appears before `stringA`.

16.5 Using the function developed for Exercise 16.4, modify the Insertion Sort program so that it operates upon strings instead of integers.

16.6 Translate the following C function into LC-3 assembly language.

```
int main()
{
   int a[5], i;

   i = 4;
   while (i >= 0) {
      a[i] = i;
      i--;
   }
}
```

16.7 For this question, examine the following program. Notice that the variable `ind` is a pointer variable that points to another pointer variable. Such a construction is legal in C.

```
#include <stdio.h>

int main()
{
   int apple;
   int *ptr;
   int **ind;
   ind = &ptr;
   *ind = &apple;
   **ind = 123;

   ind++;
   *ptr++;
   apple++;

   printf("%x %x %d\n", ind, ptr, apple);
}
```

Analyze what this program performs by drawing out the run-time stack at the point just after the statement `apple++;` executes.

16.8 The following code contains a call to the function `triple`. What is the minimum size of the activation record of `triple`?

```
int main()
{
   int array[3];

   array[0] = 1;
   array[1] = 2;
   array[2] = 3;

   triple(array);
}
```

16.9 Write a program to remove any duplicates from a sequence of numbers. For example, if the list consisted of the numbers 5, 4, 5, 5, and 3, the program would output 5, 4, 3.

16.10 Write a program to find the median of a set of numbers. Recall that the median is a number within the set in which half the numbers are larger and half are smaller. *Hint:* To perform this, you may need to sort the list first.

16.11 For this question, refer to the following C program:

```
int FindLen(char *);

int main()
{
   char str[10];

   printf("Enter a string : ");
   scanf("%s", str);
   printf("%s has %d characters\n", str, FindLen(str));
}

int FindLen(char * s)
{
   int len=0;

   while (*s != '\0') {
      len++;
      s++;
   }

   return len;
}
```

a. For the preceding C program, what is the size of the activation record for the functions `main` and `FindLen`?

b. Show the contents of the stack just before the function `FindLen` returns if the input string is `apple`.

c. What would the activation record look like if the program were run and the user typed a string of length greater than 10 characters? What would happen to the program?

16.12 The following code reads a string from the keyboard and prints out a version with any uppercase characters converted to lowercase. However, it has a flaw. Identify it.

```
#include <stdio.h>
#define MAX_LEN  10
char *LowerCase(char *s);

int main()
{
   char str[MAX_LEN];

   printf("Enter a string : ");
   scanf("%s", str);

   printf("Lowercase: %s \n", LowerCase(str));
}

char *LowerCase(char *s) {
  char newStr[MAX_LEN];
  int index;

  for (index = 0; index < MAX_LEN; index++) {
     if ('A' <= s[index] && s[index] <= 'Z')
       newStr[index] = s[index] + ('a' - 'A');
     else
       newStr[index] = s[index];
  }

  return newStr;
}
```

16.13 Consider the following declarations.

```
#define STACK_SIZE 100

int  stack[STACK_SIZE];
int  topOfStack;

int  Push(int item);
```

a. Write a funtion `Push` (the declaration is provided) that will push the value of `item` onto the top of the stack. If the stack is full and the item cannot be added, the function should return a 1. If the item is successfully pushed, the function should return a 0.

b. Write a function `Pop` that will pop an item from the top of the stack. Like `Push`, this function will return a 1 if the operation is unsuccessful. That is, a `Pop` was attempted on an empty stack. It should return a 0 if successful. Consider carefully how the popped value can be returned to the caller.