5.1 Array concept

Note_language_neutral

A typical variable stores one data item, like the number 59 or the character 'a'. Instead, sometimes a *list* of data items should be stored. Ex: A program recording points scored in each quarter of a basketball game needs a list of 4 numbers. Requiring a programmer to declare 4 variables is annoying; 200 variables would be ridiculous. An ***array*** is a special variable having one name, but storing a list of data items, with each item directly accessible. Some languages use a construct similar to an array called a ***vector***. Each item in an array is known as an ***element***.

PARTICIPATION ACTIVITY 5.1.1: Sometimes a variable should store a list, or array, of data items.

Animation captions:

1. A variable usually stores just one data item.
2. Some variables should store a list of data items, like variable pointsPerQuarter that stores 4 items.
3. Each element is accessible, like the element numbered 3.

You might think of a normal variable as a truck, and an array variable as a train. A truck has just one car for carrying "data", but a train has many cars each of which can carry data.

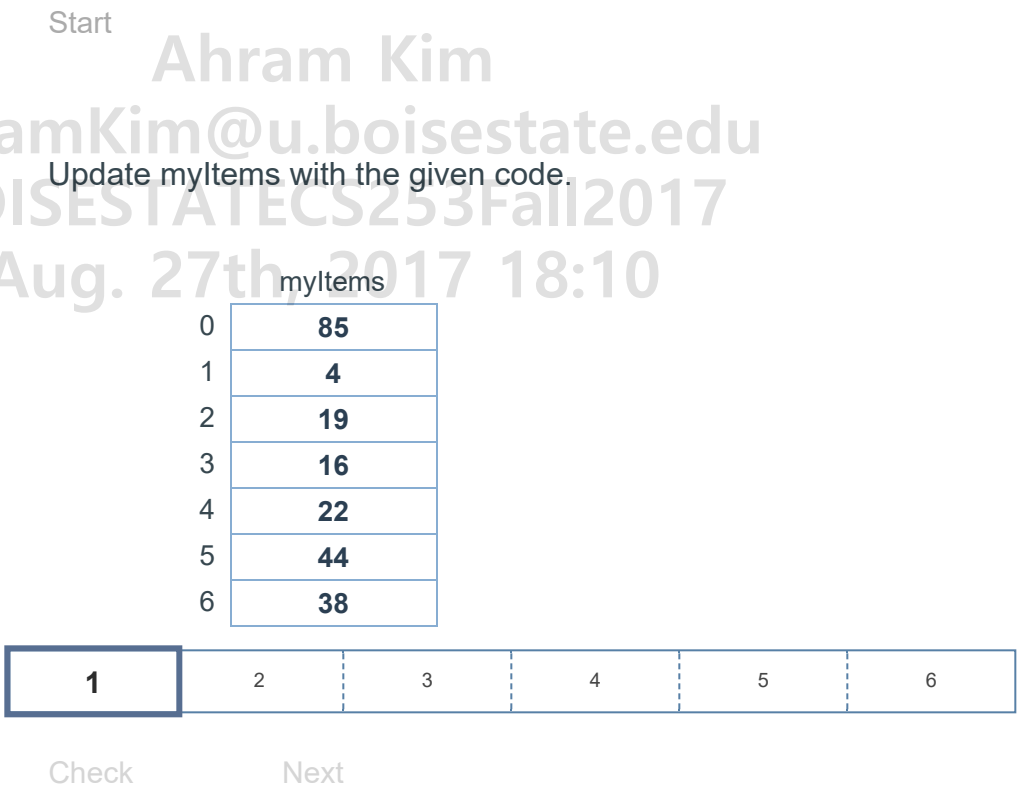
Figure 5.1.1: A normal variable is like a truck, whereas an array variable is like a train.

(Source for above images: Truck, Train)

In an array, each element's location number is called the ***index***; myArray[2] has index 2. An array's key feature is that the index enables direct access to any element, as in myArray[2]; different languages

may use different syntax, like myArray(3) or myVector.at(3). In many languages, indices start with 0 rather than 1, so an array with 4 elements has indices 0, 1, 2, and 3.

PARTICIPATION ACTIVITY 5.1.2: Update the array's data values.



PARTICIPATION ACTIVITY 5.1.3: Array basics.

Array peoplePerDay has 365 elements, one for each day of the year. Valid accesses are peoplePerDay[0], [1], ..., [364].

1) Which assigns element 0 with the value 250?

- ○ peoplePerDay[250] = 0
- ○ peoplePerDay[0] = 250
- ○ peoplePerDay = 250

2) Which assigns element 1 with the value 99?

- ○ peoplePerDay[1] = 99
- ○ peoplePerDay[99] = 1

3) Given the following statements:

```
peoplePerDay[9] = 5;
peoplePerDay[8] = peoplePerDay[9] - 3;
```

What is the value of peoplePerDay[8]?

- 8
- 5
- 2

4) Assume N is initially 1. Given the following:

```
peoplePerDay[N] = 15;
N = N + 1;
peoplePerDay[N] = peoplePerDay[N - 1] * 3;
```

What is the value of peoplePerDay[2]?

- 15
- 2
- 45

PARTICIPATION ACTIVITY 5.1.4: Arrays with element numbering starting with 0.

Array scoresList has 10 elements with indices 0 to 9, accessed as scoresList[0] to scoresList[9].

1) Assign the first element in scoresList with 77.

Check Show answer

2) Assign the second element in scoresList with 77.

Check Show answer

3) Assign the last element with 77.

Check Show answer

4) If that array instead has 100 elements, what is the last element's index?

Check Show answer

5) If the array's last index was 499, how many elements does the array have?

Check Show answer

(*Note_language_neutral) This section is mostly language neutral

5.2 Arrays

Previously-introduced variables could each only store a single item. Just as people often maintain lists of items like a grocery list or a course roster, a programmer commonly needs to maintain a list of items. A construct known as an array can be used for this purpose. An ***array*** is an ordered list of items of a given data type. Each item in an array is called an ***element***.

Construct 5.2.1: Array declaration.

```
dataType identifier[numElements];
```


This statement declares an array having the specified number of elements in memory, each element of the specified data type. The desired number of elements are specified in [ ] symbols.

Terminology note: [ ] are ***brackets***, { } are ***braces***.

The following shows how to read and assign values within an array. The program creates a variable named vals with 3 elements, each of data type int. Those three elements are in fact each a separate variable that is accessed using the syntax vals[0], vals[1], and vals[2]. Note that the 3 elements are (some might say unfortunately) numbered 0 1 2 and not 1 2 3. In an array access, the number in brackets is called the ***index*** of the corresponding element.

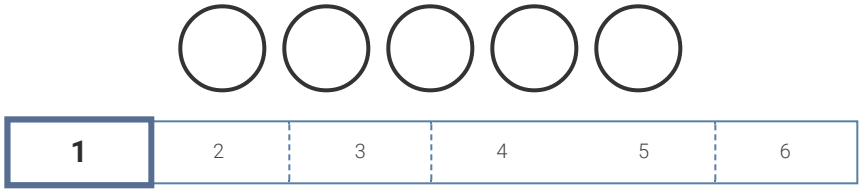
PARTICIPATION ACTIVITY 5.2.1: An array declaration creates multiple variables in memory, each accessible using [ ].

Animation captions:

1.

PARTICIPATION ACTIVITY 5.2.2: Select the index shown.

Start



Check Next

PARTICIPATION ACTIVITY 5.2.3: Array basics.

Given:

```
int yearsArr[4];

yearsArr[0] = 1999;
yearsArr[1] = 2012;
yearsArr[2] = 2025;
```


1) How many elements in memory does the array declaration create?

- ○ 0
- ○ 1
- ○ 3
- ○ 4

2) What value is stored in yearsArr[1]?

- ○ 1
- ○ 1999

- 2012

3) What value does `curr = yearsArr[2]` assign to curr?

- 2
- 2025
- Invalid index

4) What value does `curr = yearsArr[3]` assign to curr?

- 3
- 2025
- Invalid index
- Unknown

5) Recall that the array declaration was `int yearsArr[4]`. Is `curr = yearsArr[4]` a valid assignment?

- Yes, it accesses the fourth element.
- No, yearsArr[4] does not exist.

6) What is the proper way to access the *first* element in array yearsArr?

- yearsArr[1]
- yearsArr[0]

7) What are the contents of the array if the above code is followed by the statement: yearsArr[0] = yearsArr[2]?

- 1999, 2012, 1999, ?
- 2012, 2012, 2025, ?
- 2025, 2012, 2025, ?

8) What is the index of the *last* element for the following array: `int pricesArr[100];`

- 99
- 100
-

101

Besides reducing the number of variables a programmer must declare, a powerful aspect of arrays is that the index is an expression. Thus, an access could be written as userNums[i] where i is an int variable. As such, an array is useful to easily lookup the Nth item in a list. Consider the following program that allows a user to print the age of the Nth oldest known person to have ever lived.

Figure 5.2.1: Array's ith element can be directly accessed using [i]: Oldest people program.

```
#include <stdio.h>

int main(void) {
   int oldestPeople[5]; // Source: Wikipedia.org
   int nthPerson = 0;   // User input, Nth oldest person

   oldestPeople[0] = 122; // Died 1997 in France
   oldestPeople[1] = 119; // Died 1999 in U.S.
   oldestPeople[2] = 117; // Died 1993 in U.S.
   oldestPeople[3] = 117; // Died 1998 in Canada
   oldestPeople[4] = 116; // Died 2006 in Ecuador

   printf("Enter N (1-5): ");
   scanf("%d", &nthPerson);

   if ((nthPerson >= 1) && (nthPerson <= 5)) {
      printf("The %dth oldest person lived ", nthPerson);
      printf("%d years.Wn", oldestPeople[nthPerson-1]);
   }

   return 0;
}
```

```
Enter N (1-5): 1
The 1th oldest person lived 122 years.

...

Enter N (1-5): 4
The 4th oldest person lived 117 years.

...

Enter N (1-5): 9

...

Enter N (1-5): 0

...

Enter N (1-5): 5
The 5th oldest person lived 116 years.
```

The program can quickly access the Nth oldest person's age using `oldestPeople[nthPerson - 1]`. Note that the index is `nthPerson - 1` rather than just `nthPerson` because an array's indices start at 0, so the 1st age is at index 0, the 2nd at index 1, etc.

An array's index must be an integer type. The array index cannot be a floating-point type, even if the value is 0.0, 1.0, etc.

A key advantage of arrays becomes evident when used in conjunction with loops. To illustrate, the following program allows a user to enter 8 integer values, then prints those 8 values:

Figure 5.2.2: Arrays combined with loops are powerful together: User-entered numbers.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements in array
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index

   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   printf("You entered: ");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d ", userVals[i]);
   }
   printf("Wn");

   return 0;
}
```

```
Enter 8 integer values...
Value: 5
Value: 99
Value: -1
Value: -44
Value: 8
Value: 555555
Value: 0
Value: 2
You entered: 5 99 -1 -44 8 555555 0 2
```


Consider how the program would have had to be written if using 8 separate variables. That program would have repeated variable declarations, output statements, and input statements. Now consider that program for NUM_ELEMENTS equal to 100, 1000, or more. With arrays and loops, the code would be the same as above. Only the constant literal 8 would be changed.

Like other variables, an array's elements are not automatically initialized during the variable declaration and should be initialized before being read. A programmer may initialize an array's elements in an array variable declaration:

Construct 5.2.2: Array initialization.

```
type identifier[N] = {val0, val1, ..., valN - 1};
```

An example is: `int myArray[3] = {0, 0, 0};`. For larger arrays, a loop may be used for initialization.

Like other variables, the keyword ***const*** may be prepended to an array variable declaration to prevent changes to the array. Thus, `const int YEARS[3] = {1865, 1920, 1964};` followed by `YEARS[0] = 2000;` yields a compiler error.

PARTICIPATION ACTIVITY 5.2.4: Array declaration and use.

1) Declare an array named myVals that stores 10 items of type int.

Check Show answer

2) Assign the value stored at index 8 of array myVals to a variable x.

Check Show answer

3) Assign the value 555 to the element at index 2 of array myVals.

Check Show answer

4) Assign the value 777 to the second element of array myVals.

Check Show answer

5) Declare an array of ints named myVals with 4 elements each initialized to 10. The array declaration and initialization should be done in a single statement.

Check Show answer

CHALLENGE ACTIVITY 5.2.1: Enter the output for the array.

Start

Type the program's output.

```
2
5
7
```

```
#include <stdio.h>
int main(void) {
   const int NUM_ELEMENTS = 3;
   int userVals[NUM_ELEMENTS];
   int i = 0;

   userVals[0] = 2;
   userVals[1] = 5;
   userVals[2] = 7;

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d\n", userVals[i]);
   }

   return 0;
}
```

Check Next

CHALLENGE ACTIVITY 5.2.2: Printing array elements.

Write three statements to print the first three elements of array runTimes. Follow each statement with a newline. Ex: If runTime = {800, 775, 790, 805, 808}, print:

800
775
790

Note: These activities may test code with different test values. This activity will perform two tests, the first with a 5-element array (int runTimes[5]), the second with a 4-element array (int runTimes[4]). See How to Use zyBooks.

Also note: If the submitted code tries to access an invalid array element, such as runTime[9] for a 5-element array, the test may generate strange results. Or the test may crash and report "Program end never reached", in which case the system doesn't print the test case that caused the reported message.


```
1 #include <stdio.h>
2
3 int main(void) {
4    int runTimes[5];
5
6    // Populate array
7    runTimes[0] = 800;
8    runTimes[1] = 775;
9    runTimes[2] = 790;
```

```
    runTimes[3] = 805;
    runTimes[4] = 808;

    /* Your solution goes here  */

    return 0;
}
```


Run

CHALLENGE ACTIVITY 5.2.3: Printing array elements with a for loop.

Write a for loop to print all elements in courseGrades, following each element with a space (including the last). Print forwards, then backwards. End each loop with a newline. Ex: If courseGrades = {7, 9, 11, 10}, print:

```
7 9 11 10
10 11 9 7
```

Hint: Use two for loops. Second loop starts with i = NUM_VALS - 1. (Notes)

Note: These activities may test code with different test values. This activity will perform two tests, the first with a 4-element array (int courseGrades[4]), the second with a 2-element array (int courseGrades[2]). See How to Use zyBooks.

Also note: If the submitted code tries to access an invalid array element, such as courseGrades[9] for a 4-element array, the test may generate strange results. Or the test may crash and report "Program end never reached", in which case the system doesn't print the test case that caused the reported message.


```
#include <stdio.h>

int main(void) {
   const int NUM_VALS = 4;
   int courseGrades[NUM_VALS];
   int i = 0;

   courseGrades[0] = 7;
   courseGrades[1] = 9;
   courseGrades[2] = 11;
   courseGrades[3] = 10;

   /* Your solution goes here  */

```

```
15    return 0;
16 }
```

Run

5.3 Array iteration drill

The following activities can help one become comfortable with iterating through arrays or vectors, before learning to code such iteration.

PARTICIPATION ACTIVITY 5.3.1: Find the maximum value in the array.

Click "Store value" if a new maximum value is seen.

Start

X	X	X	X	X	X	X

Stored value
-1

Next value Store value

Time - Best time - Clear best

PARTICIPATION ACTIVITY 5.3.2: Negative value counting in array.

Click "Increment" if a negative value is seen.

Start

X	X	X	X	X	X	X

Counter
0

Next value Increment

Time - Best time - Clear best

PARTICIPATION ACTIVITY 5.3.3: Array sorting largest value.

Move the largest value to the right-most position. Click "Swap values" if the larger of the two current values is on the left.

Start

X	X	X	X	X	X	X

Next value Swap values

Time - Best time - Clear best

5.4 Iterating through arrays

Iterating through arrays using loops is commonplace and is an important programming skill to master.

Because array indices are numbered 0 to N - 1 rather than 1 to N, programmers commonly use this for loop structure:

Figure 5.4.1: Common for loop structure for iterating through an array.

```
// Iterating through myArray
for (i = 0; i < numElements; ++i) {
   // Loop body accessing myArray[i]
}
```


Note that index variable i is initialized to 0, and the loop expression is i < N rather than i <= N. If N were 5, the loop's iterations would set i to 0, 1, 2, 3, and 4, for a total of 5 iterations. The benefit of the loop structure is that each array element is accessed as myArray[i] rather than the more complex myArray[i - 1].

Programs commonly iterate through arrays to determine some quantity about the array's items. For example, the following program determines the maximum value in a user-entered list.

Figure 5.4.2: Iterating through an array example: Program that finds the max

item.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8;  // Number of elements
   int userVals[NUM_ELEMENTS];  // Array of user numbers
   int i = 0;                   // Loop index
   int maxVal = 0;              // Computed max

   // Prompt user to populate array
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Determine largest (max) number
   maxVal = userVals[0];        // Largest so far

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      if (userVals[i] > maxVal) {
         maxVal = userVals[i];
      }
   }
   printf("Max: %dWn", maxVal);

   return 0;
}
```

```
Enter 8 integer values...
Value: 3
Value: 5
Value: 23
Value: -1
Value: 456
Value: 1
Value: 6
Value: 83
Max: 456

...

Enter 8 integer values...
Value: -5
Value: -10
Value: -44
Value: -2
Value: -27
Value: -9
Value: -27
Value: -9
Max: -2
```


If the user enters numbers 7, -9, 55, 44, 20, -400, 0, 2, then the program will output "max: 55". The bottom part of the code iterates through the array to determine the maximum value. The main idea of that code is to use a variable maxVal to store the largest value seen "so far" as the program iterates through the array. During each iteration, if the array's current element value is larger than the max seen so far, the program writes that value to maxVal (akin to being able to carry only one item as you walk through a store, replacing the current item by a better item whenever you see one). Before entering the loop, maxVal must be initialized to some value because max will be compared with each array element's value. A logical error would be to initialize maxVal to 0, because 0 is not in fact the largest value seen so far, and would result in incorrect output (of 0) if the user entered all negative numbers. Instead, the program peeks at an array element (using the first element, though any element could have been used) and initializes maxVal to that element's value.

PARTICIPATION ACTIVITY 5.4.1: Array iteration.

Given an integer array myVals of size N_SIZE (i.e. int myVals[N_SIZE]), complete the code to achieve the stated goal.

1) Determine the minimum number in the array, using the same initialization as the maximum number example above.

```
minVal =
;

for (i = 0; i < N_SIZE; ++i) {

     if (myVals[i] < minVal) {

          minVal = myVals[i];

     }
}
```


Check Show answer

2) Count how many negative numbers exist in the array.

```
cntNeg = 0;

for (i = 0; i < N_SIZE; ++i) {

     if (
                              ) {

          ++cntNeg;

     }

}
```

Check Show answer

3) Count how many odd numbers exist in the array.

```
    cntOdd = 0;

    for (i = 0; i < N_SIZE; ++i) {

         if ( (myVals[i] % 2) == 1
    ) {

                                ;

    }
}
```


Check Show answer

A common error is to try to access an array with an index that is out of the array's index range, e.g., to try to access v[8] when v's valid indices are 0-7. Care should be taken whenever a user enters a number that is then used as an array index, and when using a loop index as an array index also, to ensure the index is within the array's valid index range. Checking whether an array index is in range is very important. Trying to access an array with an out-of-range index is not only a very common error, but is also one of the hardest errors to debug. The following animation shows what happens when a program writes to an out-of-range index using an array.

PARTICIPATION ACTIVITY 5.4.2: Writing to an out-of-range index using an array.

Animation captions:

1. int variable age is allocated to location in memory immediately after the array weights.
2.
3. Assigning to weights[3] will overwrite the memory location for variable age.
4. Incorrect value for age is now displayed.

A write to an array with an out-of-range index may simply write to a memory location of a different variable X residing next to the array in memory. Later, when the program tries to read X, the program encounters incorrect data. For example, a program may write X with the number 44, but when reading X later in the program X may be 2533, with X never (intentionally) written by any program statement in between.

Iterating through an array for various purposes is an important programming skill to master. Here is another example, computing the sum of an array of int variables:

Figure 5.4.3: Iterating through an array example: Program that finds the sum of

an array's elements.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index
   int sumVal = 0;             // For computing sum

   // Prompt user to populate array
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Determine sum
   sumVal = 0;

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      sumVal = sumVal + userVals[i];
   }

   printf("Sum: %dWn", sumVal);

   return 0;
}
```

```
Enter 8 integer values...
Value: 3
Value: 5
Value: 234
Value: 346
Value: 234
Value: 73
Value: 26
Value: -1
Sum: 920

...

Enter 8 integer values...
Value: 3
Value: 5
Value: 234
Value: 346
Value: 234
Value: 73
Value: 26
Value: 1
Sum: 922
```


Note that the code is somewhat different than the code computing the max. For computing the sum, the program initializes a variable sum to 0, then simply adds the current iteration's array element value to that sum.

PARTICIPATION ACTIVITY 5.4.3: Print the sum and average of an array's elements.

Modify the program to print the average (mean) as well as the sum. Hint: You don't actually have to change the loop, but rather change what you print.

Load default template...

3 5 234 346 234 73 26 -1

Run

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     const int NUM_ELEMENTS = 8; // Number of elements
6     int userVals[NUM_ELEMENTS]; // User numbers
7     int i = 0;                  // Loop index
8     int sumVal = 0;             // For computing sum
9
10    // Prompt user to populate array
11    printf("Enter %d integer values...\n", NUM_ELEMENTS);
12
13    for (i = 0; i < NUM_ELEMENTS; ++i) {
14       printf("Value: \n");
15       scanf("%d", &(userVals[i]));
16    }
17
```

```
18    // Determine sum
19    sumVal = 0;
20
21    for (i = 0; i < NUM_ELEMENTS; ++i) {
```

PARTICIPATION ACTIVITY 5.4.4: Print selected elements of an array.

Modify the program to instead just print each number that is greater than 21.

Load default template...

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     const int NUM_ELEMENTS = 8; // Number of elements
6     int userVals[NUM_ELEMENTS]; // User numbers
7     int i = 0;                  // Loop index
8     int sumVal = 0;             // For computing sum
9
10    // Prompt user to populate array
11    printf("Enter %d integer values...\n", NUM_ELEMENTS);
12
13    for (i = 0; i < NUM_ELEMENTS; ++i) {
14       printf("Value: \n");
15       scanf("%d", &(userVals[i]));
16    }
17
18    // Determine sum
19    sumVal = 0;
20
21    for (i = 0; i < NUM ELEMENTS; ++i) {
```

3 5 234 346 234 73 26 -1

Run

CHALLENGE ACTIVITY 5.4.1: Enter the output for the array.

Start

Type the program's output.

```
3
4
7
```

```
#include <stdio.h>
int main(void) {
   const int NUM_ELEMENTS = 3;
   int userVals[NUM_ELEMENTS];
   int i = 0;

   userVals[0] = 3;
   userVals[1] = 4;
   userVals[2] = 7;

   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d\n", userVals[i]);
   }

   return 0;
}
```


Check Next

CHALLENGE ACTIVITY 5.4.2: Finding values in arrays.

Set numMatches to the number of elements in userValues (having NUM_VALS elements) that equal matchValue. Ex: If matchValue = 2 and userValues = {2, 2, 1, 2}, then numMatches = 3. (Notes)

```
1  #include <stdio.h>
2
3  int main(void) {
4     const int NUM_VALS = 4;
5     int userValues[NUM_VALS];
6     int i = 0;
7     int matchValue = 0;
8     int numMatches = -99; // Assign numMatches with 0 before your for loop
9
10    userValues[0] = 2;
11    userValues[1] = 2;
12    userValues[2] = 1;
13    userValues[3] = 2;
14
15    matchValue = 2;
16
17    /* Your solution goes here  */
18
19    printf("matchValue: %d, numMatches: %d\n", matchValue, numMatches);
20
21    return 0;
```


Run

CHALLENGE ACTIVITY 5.4.3: Populating an array with a for loop.

Write a for loop to populate array userGuesses with NUM_GUESSES integers. Read integers using scanf. Ex: If NUM_GUESSES is 3 and user enters 9 5 2, then userGuesses is {9, 5, 2}.

```
#include <stdio.h>

int main(void) {
   const int NUM_GUESSES = 3;
   int userGuesses[NUM_GUESSES];
   int i = 0;

   /* Your solution goes here  */
   for (i = 0; i < NUM_GUESSES; ++i) {
      printf("%d ", userGuesses[i]);
   }

   return 0;
}
```


Run

CHALLENGE ACTIVITY 5.4.4: Array iteration: Sum of excess.

Array testGrades contains NUM_VALS test scores. Write a for loop that sets sumExtra to the total extra credit received. Full credit is 100, so anything over 100 is extra credit. Ex: If testGrades = {101, 83, 107, 90}, then sumExtra = 8, because 1 + 0 + 7 + 0 is 8.

```
#include <stdio.h>

int main(void) {
   const int NUM_VALS = 4;
   int testGrades[NUM_VALS];
   int i = 0;
   int sumExtra = -9999; // Assign sumExtra with 0 before your for loop

   testGrades[0] = 101;
   testGrades[1] = 83;
   testGrades[2] = 107;
   testGrades[3] = 90;

   /* Your solution goes here  */

   printf("sumExtra: %d\n", sumExtra);
   return 0;
}
```

Run

CHALLENGE ACTIVITY 5.4.5: Printing array elements separated by commas.

Write a for loop to print all NUM_VALS elements of array hourlyTemp. Separate elements with a comma and space. Ex: If hourlyTemp = {90, 92, 94, 95}, print:

90, 92, 94, 95

Note that the last element is not followed by a comma, space, or newline.

```
1  #include <stdio.h>
2
3  int main(void) {
4     const int NUM_VALS = 4;
5     int hourlyTemp[NUM_VALS];
6     int i = 0;
7
8     hourlyTemp[0] = 90;
9     hourlyTemp[1] = 92;
10    hourlyTemp[2] = 94;
11    hourlyTemp[3] = 95;
12
13    /* Your solution goes here  */
14
15    printf("\n");
16
17    return 0;
18 }
```

Run

5.5 Multiple arrays

Programmers commonly use multiple same-sized arrays to store related lists. For example, the following program maintains a list of letter weights in ounces, and another list indicating the corresponding postage cost for first class mail (usps.com).

Figure 5.5.1: Multiple array example: Letter postage cost program.

```
#include <stdio.h>
#include <stdbool.h>

int main (void) {
   const int NUM_ELEMENTS = 14;        // Number of elements
   double letterWeights[NUM_ELEMENTS]; // Weights in ounces
   int postageCosts[NUM_ELEMENTS];     // Costs in cents (usps.com 2013)
   double userLetterWeight = 0.0;      // Letter weight
   bool foundWeight = false;           // Found weight specified by user
   int i = 0;                          // Loop index


   // Populate letter weight/postage cost arrays
   letterWeights[i] = 1;   postageCosts[i] =  46; ++i;
   letterWeights[i] = 2;   postageCosts[i] =  66; ++i;
   letterWeights[i] = 3;   postageCosts[i] =  86; ++i;
   letterWeights[i] = 3.5; postageCosts[i] = 106; ++i;
   letterWeights[i] = 4;   postageCosts[i] = 152; ++i;
   letterWeights[i] = 5;   postageCosts[i] = 172; ++i;
   letterWeights[i] = 6;   postageCosts[i] = 192; ++i;
   letterWeights[i] = 7;   postageCosts[i] = 212; ++i;
   letterWeights[i] = 8;   postageCosts[i] = 232; ++i;
   letterWeights[i] = 9;   postageCosts[i] = 252; ++i;
   letterWeights[i] = 10;  postageCosts[i] = 272; ++i;
   letterWeights[i] = 11;  postageCosts[i] = 292; ++i;
   letterWeights[i] = 12;  postageCosts[i] = 312; ++i;
   letterWeights[i] = 13;  postageCosts[i] = 332; ++i;

   // Prompt user to enter letter weight
   printf("Enter letter weight (in ounces): ");
   scanf("%lf", &userLetterWeight);


   // Postage costs is based on smallest letter weight greater than
   // or equal to mailing letter weight
   foundWeight = false;

   for (i = 0; (i < NUM_ELEMENTS) && (!foundWeight); ++i) {
      if( userLetterWeight <= letterWeights[i] ) {
         foundWeight = true;
         printf("Postage for USPS first class mail is %d centsWn",
                postageCosts[i]);
      }
   }

   if( !foundWeight ) {
      printf("Letter is too heavy for USPS first class mail.Wn");
   }

   return 0;
}
```

```
Enter letter weight (in ounces): 3
Postage for USPS first class mail is 86 cents

...

Enter letter weight (in ounces): 9.5
Postage for USPS first class mail is 272 cents

...

Enter letter weight (in ounces): 15
Letter is too heavy for USPS first class mail.
```

Notice how the `if (userLetterWeight <= letterWeights[i])` statement compares the user-entered letter weight with the current element in the letterWeights array. If the entered weight is less than or equal to the current element in the letterWeights array, the program prints the element in postageCosts having that same index.

The loop's expression `(i < NUM_ELEMENTS) && (!foundWeight)` depends on the value of the variable foundWeight. This expression prevents the loop from iterating through the entire array once the correct letter weight has been found. Omitting the check for found from the loop expression would result in an incorrect output; the program would incorrectly print the postage cost for all letter weights greater than the user's letter weight.

Note that the array initialization uses [i] rather than [0], [1], etc. Such a technique is less prone to errors, and enables easy reordering or inserting of new letter weights and postage costs.

PARTICIPATION ACTIVITY 5.5.1: Multiple arrays in the above postage cost program.

1) letterWeights[0] is 1, meaning element 0 of letterWeights and postageCosts correspond to a weight of 1 ounce.
   - True
   - False

2) postageCosts[2] represents the cost for a weight of 2 ounces.
   - True
   - False

3) The program fails to provide a cost for a weight of 7.5.
   -

True

False

PARTICIPATION ACTIVITY 5.5.2: Postage calculation with negative weight error message.

Improve the program by also outputting "The next higher weight is ___ with a cost of ___ cents".

Load default template...

```
#include <stdio.h>
#include <stdbool.h>

int main (void) {
   const int NUM_ELEMENTS = 14;         // Number of element
   double letterWeights[NUM_ELEMENTS]; // Weights in ounces
   int postageCosts[NUM_ELEMENTS];      // Costs in cents (u
   double userLetterWeight = 0.0;       // Letter weight
   bool foundWeight = false;            // Found weight spec
   int i = 0;                           // Loop index


   // Populate letter weight/postage cost arrays
   letterWeights[i] = 1;    postageCosts[i] =  46; ++i;
   letterWeights[i] = 2;    postageCosts[i] =  66; ++i;
   letterWeights[i] = 3;    postageCosts[i] =  86; ++i;
   letterWeights[i] = 3.5;  postageCosts[i] = 106; ++i;
   letterWeights[i] = 4;    postageCosts[i] = 152; ++i;
```

3

Run

PARTICIPATION ACTIVITY 5.5.3: Multiple arrays.

1) Using two separate statements, declare two related integer arrays named seatPosition and testScore (in that order) each with 130 elements.

Check Show answer

2) How many total elements are stored within the two arrays int familyAges[50] and double familyHeights[50]?

Check Show answer

CHALLENGE ACTIVITY 5.5.1: Printing the sum of two array elements.

Add each element in origList with the corresponding value in offsetAmount. Print each sum followed by a space. Ex: If origList = {40, 50, 60, 70} and offsetAmount = {5, 7, 3, 0}, print:

45 57 63 70


```
#include <stdio.h>

int main(void) {
   const int NUM_VALS = 4;
   int origList[NUM_VALS];
   int offsetAmount[NUM_VALS];
   int i = 0;

   origList[0] = 40;
   origList[1] = 50;
   origList[2] = 60;
   origList[3] = 70;

   offsetAmount[0] = 5;
   offsetAmount[1] = 7;
   offsetAmount[2] = 3;
   offsetAmount[3] = 0;

   /* Your solution goes here  */

   printf("\n");
```

Run

CHALLENGE ACTIVITY 5.5.2: Multiple arrays: Key and value.

For any element in keysList with a value greater than 100, print the corresponding value in itemsList, followed by a space. Ex: If keysList = {42, 105, 101, 100} and itemsList = {10, 20, 30, 40}, print:

20 30

Since keysList[1] and keysList[2] have values greater than 100, the value of itemsList[1] and itemsList[2] are printed.

```
#include <stdio.h>
```

```
int main(void) {
   const int SIZE_LIST = 4;
   int keysList[SIZE_LIST];
   int itemsList[SIZE_LIST];
   int i = 0;

   keysList[0] = 42;
   keysList[1] = 105;
   keysList[2] = 101;
   keysList[3] = 100;

   itemsList[0] = 10;
   itemsList[1] = 20;
   itemsList[2] = 30;
   itemsList[3] = 40;

   /* Your solution goes here  */

   printf("\n");
```

Run

5.6 Swapping two variables

Note_language_neutral2

Sometimes a program must swap values among two variables. ***Swapping*** two variables x and y means to assign y's value to x, and x's value to y. If x is 33 and y is 55, then after swapping x is 55 and y is 33.

A common method for swapping uses a temporary third variable. To understand the intuition of such temporary storage, consider a person holding a book in one hand and a phone in the other, wishing to swap the items. The person can temporarily place the phone on a table, move the book to the other hand, then pick up the phone.

PARTICIPATION ACTIVITY 5.6.1: Swap idea: Use a temporary location.

Animation captions:

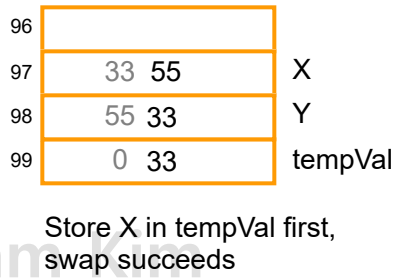
1. A swap between two hands requires a third, temporary place

Similarly, swapping two variables can use a third variable to temporarily hold one value while the other value is copied over.

PARTICIPATION ACTIVITY 5.6.2: Swapping two variables using a third temporary variable.

Start

```
int X = 33;
int Y = 55;
int tempVal = 0;

tempVal = X;
X = Y;
Y = tempVal;

// Print X and Y
```



Store X in tempVal first, swap succeeds

```
X: 55, Y: 33
```


PARTICIPATION ACTIVITY 5.6.3: Swap.

Given x = 22 and y = 99. What are x and y after the given code?

1)
```
x = y;
y = x;
```

- x is 99 and y is 22.
- x is 22 and y is 99.
- x is 99 and y is 99.

2)
```
x = y;
y = x;
x = y;
```

- x is 99 and y is 22.
- x is 99 and y is 99.
- x is 22 and y is 22.

3)
```
tempVal = x;
x = y;
y = x;
```

- x is 99 and y is 22.
- x is 99 and y is 99.

4)
```
tempVal = x;
x = y;
y = tempVal;
```

- x is 99 and y is 22.
- x is 99 and y is 99.

If you have studied arrays or vectors (or other kinds of lists), know that most swaps are actually performed between two list elements. For example, reversing a list with N elements can be achieved by swapping element 1 and N, element 2 and N-1, element 3 and N-2, etc. (stopping at the middle of the list).

PARTICIPATION ACTIVITY 5.6.4: Reversing a list using swaps.

Animation captions:

1. Swap outermost elements.
2. Swap next outermost elements, repeat until reach at middle.

PARTICIPATION ACTIVITY 5.6.5: Reversing a list using swaps.

1) Using the above approach, how many swaps are needed to reverse this list:
999 888 777 666 555 444 333 222

Check Show answer

(*Note_language_neutral2) This section is mostly language neutral

5.7 Loop-modifying or copying/comparing arrays

Sometimes a program changes some elements' values or moves elements while iterating through a array. The following uses a loop to convert any negative array element values to 0.

Figure 5.7.1: Modifying an array during iteration example: Converting negatives to 0 program.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8;  // Number of elements
   int userVals[NUM_ELEMENTS];  // User values
   int i = 0;                   // Loop index

   // Prompt user to input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Convert negatives to 0
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      if (userVals[i] < 0) {
         userVals[i] = 0;
      }
   }

   // Print numbers
   printf("New numbers: ");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d ", userVals[i]);
   }

   return 0;
}
```

```
Enter 8 integer values...
Value: 5
Value: 67
Value: -5
Value: -4
Value: 5
Value: 6
Value: 6
Value: 4
New numbers: 5 67 0 0 5 6 6 4
```


PARTICIPATION ACTIVITY 5.7.1: Modifying an array in a loop.

What is the resulting array contents, assuming each question starts with an array of size 4 having contents -55, -1, 0, 9?

1)

```
for (i = 0; i < 4; ++i) {
   itemsList[i] = i;
}
```

- -54, 0, 1, 10
- 0, 1, 2, 3
- 1, 2, 3, 4

2)

```
for (i = 0; i < 4; ++i) {
   if (itemsList[i] < 0) {
      itemsList[i] = itemsList[i] * -1;
   }
}
```

- -55, -1, 0, -9
- 55, 1, 0, -9
- 55, 1, 0, 9

3)

```
for (i = 0; i < 4; ++i) {
   itemsList[i] = itemsList[i+1];
}
```

- -1, 0, 9, 0
- 0, -55, -1, 0
- Error

4)

```
for (i = 0; i < 3; ++i) {
   itemsList[i] = itemsList[i+1];
}
```

- -1, 0, 9, 9
- Error
- -1, 0, 9, 0

5)

```
for (i = 0; i < 3; ++i) {
   itemsList[i+1] = itemsList[i];
}
```

- -55, -55, -55, -55
- 0, -55, -1, 0
- Error

PARTICIPATION ACTIVITY 5.7.2: Modifying an array during iteration example: Doubling element values.

Complete the following program to double each number in the array.

Load default template...

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     const int NUM_ELEMENTS = 8;  // Number of elements
6     int userVals[NUM_ELEMENTS];  // User values
7     int i = 0;                   // Loop index
8
9     // Prompt user to input values
10    printf("Enter %d integer values...\n", NUM_ELEMENTS);
11    for (i = 0; i < NUM_ELEMENTS; ++i) {
12       printf("Value: \n");
13       scanf("%d", &(userVals[i]));
14    }
15
16    // Double each element. FIXME write this loop
17
18    // Print numbers
19    printf("New numbers: ");
20    for (i = 0; i < NUM_ELEMENTS; ++i) {
21       printf("%d ", userVals[i]);
```

5 67 -5 -4 5 6 6 4

Run

Copying an array is a common task. Given a second array of the same size, a loop can copy each element one-by-one. Modifications to either array do not affect the other.

Figure 5.7.2: Array copying: Converting negatives to 0 program.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8;   // Number of elements
   int userVals[NUM_ELEMENTS];   // User numbers
   int copiedVals[NUM_ELEMENTS]; // Copied/modified user numbers
   int i = 0;                    // Loop index

   // Prompt user for input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Copy userNums to copiedNums array
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      copiedVals[i] =  userVals[i];
   }

   // Convert negatives to 0
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      if (copiedVals[i] < 0) {
         copiedVals[i] = 0;
      }
   }

   // Print numbers
   printf("WnOriginal and new values: Wn");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d became %dWn", userVals[i], copiedVals[i]);
   }
   printf("Wn");

   return 0;
}
```

```
Enter 8 integer values...
Value: 12
Value: -5
Value: 34
Value: 75
Value: -14
Value: 33
Value: 12
Value: -102

Original and new values:
12 became 12
-5 became 0
34 became 34
75 became 75
-14 became 0
33 became 33
12 became 12
-102 became 0
```

PARTICIPATION ACTIVITY 5.7.3: Array copying.

Given array firstList with size 4 and element values 33, 44, 55, 66, and array secondList with size 4 and elements values 0, 0, 0, 0.

1) firstList = secondList copies 0s into each firstList element.

   - True
   - False

2) This loop copies firstList to secondList, so that secondList becomes 33, 44, 55, 66:

```
for (i = 0; i < 4; ++i) {
   secondList[i] = firstList[i];
}
```

   - True
   - False

3) After a for loop copies firstList to secondList, the assignment secondList[0] = 99 will modify both arrays.

   - True
   - False

4) Given thirdList with size 5 and elements 22, 21, 20, 19, 18, the following causes firstList's values to be 22, 21, 20, 19, 18:

```
for (i = 0; i < 5; ++i) {
   firstList[i] = thirdList[i];
}
```

   - True
   - False

CHALLENGE ACTIVITY 5.7.1: Decrement array elements.

Write a loop that subtracts 1 from each element in lowerScores. If the element was already 0 or negative, assign 0 to the element. Ex: lowerScores = {5, 0, 2, -3} becomes {4, 0, 1, 0}.

```
#include <stdio.h>

int main(void) {
   const int SCORES_SIZE = 4;
   int lowerScores[SCORES_SIZE];
   int i = 0;

   lowerScores[0] = 5;
   lowerScores[1] = 0;
   lowerScores[2] = 2;
   lowerScores[3] = -3;

   /* Your solution goes here  */

   for (i = 0; i < SCORES_SIZE; ++i) {
      printf("%d ", lowerScores[i]);
   }
   printf("\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 5.7.2: Copy and modify array elements.

Write a loop that sets newScores to oldScores shifted once left, with element 0 copied to the end. Ex: If oldScores = {10, 20, 30, 40}, then newScores = {20, 30, 40, 10}.

Note: These activities may test code with different test values. This activity will perform two tests, the first with a 4-element array (newScores = {10, 20, 30, 40}), the second with a 1-element array (newScores = {199}). See How to Use zyBooks.

Also note: If the submitted code tries to access an invalid array element, such as newScores[9] for a 4-element array, the test may generate strange results. Or the test may crash and report "Program end never reached", in which case the system doesn't print the test case that caused the reported message.


```
#include <stdio.h>

int main(void) {
   const int SCORES_SIZE = 4;
   int oldScores[SCORES_SIZE];
   int newScores[SCORES_SIZE];
   int i = 0;

```

```
    oldScores[0] = 10;
    oldScores[1] = 20;
    oldScores[2] = 30;
    oldScores[3] = 40;

    /* Your solution goes here  */

    for (i = 0; i < SCORES_SIZE; ++i) {
       printf("%d ", newScores[i]);
    }
    printf("\n");

```

Run

CHALLENGE ACTIVITY 5.7.3: Modify array elements using other elements.

Write a loop that sets each array element to the sum of itself and the next element, except for the last element which stays the same. Be careful not to index beyond the last element. Ex:

```
Initial scores:        10, 20, 30, 40
Scores after the loop: 30, 50, 70, 40
```

The first element is 30 or 10 + 20, the second element is 50 or 20 + 30, and the third element is 70 or 30 + 40. The last element remains the same.

```
#include <stdio.h>

int main(void) {
   const int SCORES_SIZE = 4;
   int bonusScores[SCORES_SIZE];
   int i = 0;

   bonusScores[0] = 10;
   bonusScores[1] = 20;
   bonusScores[2] = 30;
   bonusScores[3] = 40;

   /* Your solution goes here  */

   for (i = 0; i < SCORES_SIZE; ++i) {
      printf("%d ", bonusScores[i]);
   }
   printf("\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 5.7.4: Modify an array's elements.

Double any element's value that is less than minVal. Ex: If minVal = 10, then dataPoints = {2, 12, 9, 20} becomes {4, 12, 18, 20}.

```
#include <stdio.h>

int main(void) {
   const int NUM_POINTS = 4;
   int dataPoints[NUM_POINTS];
   int minVal = 0;
   int i = 0;

   dataPoints[0] = 2;
   dataPoints[1] = 12;
   dataPoints[2] = 9;
   dataPoints[3] = 20;

   minVal = 10;

   /* Your solution goes here  */

   for (i = 0; i < NUM_POINTS; ++i) {
      printf("%d ", dataPoints[i]);
   }
   printf("\n");
```

Run

5.8 Debugging example: Reversing an array

A common array modification is to reverse an array's elements. One way to accomplish this goal is to perform a series of swaps. For example, starting with an array of numbers 10 20 30 40 50 60 70 80, we could first swap the first item with the last item, yielding 80 20 30 40 50 60 70 10. We could next swap the second item with the second-to-last item, yielding 80 70 30 40 50 60 20 10. The next swap would yield 80 70 60 40 50 30 20 10, and the last would yield 80 70 60 50 40 30 20 10.

With this basic idea of how to reverse an array, we can attempt to write a program to carry out such reversal. Below we develop such a program but we make common mistakes along the way, to aid learning from examples of what not to do.

A first attempt to write a program that reverses an array appears below:

Figure 5.8.1: First program attempt to reverse array: Invalid access out of array bounds.

```c
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index

   // Prompt user to input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Reverse array's elements
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      userVals[i] = userVals[NUM_ELEMENTS - i]; // Swap
   }

   // Print numbers
   printf("WnNew values: ");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d ", userVals[i]);
   }

   return 0;
}
```

```
Enter 8 integer values...
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80

New values: 0 80 70 60 50 60 70 80
```


Something went wrong: The program did not reverse the array, and the first element was set to 0. Let's try to find the code that caused the problem.

The first and third for loops are fairly standard, so let's initially focus attention on the middle for loop that does the reversing. The swap statement inside that loop is `userNums[i] = userNums[NUM_ELEMENTS - i]`. When i is 0, the statement will execute `userNums[0] = userNums[8];`. However, userNums has size 8 and thus valid indices are 0..7. userNums[8] does not exist. The program should actually swap elements 0 and 7, then 1 and 6, etc. Thus, let's change the right-side index to `NUM_VALUES - 1 - i`. The revised program is shown below.

Figure 5.8.2: Next program attempt to reverse an array: Doesn't reverse properly; we forgot to swap.


```
Enter 8 integer values...
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80

New values: 80 70 60 50 50 60 70 80
```

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index

   // Prompt user to input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
```


The last four elements are still wrong. To determine what went wrong, we can manually (i.e., on paper) trace the loop's execution.

- i is 0: userNums[0] = userNums[7]. Array now: 80 20 30 40 50 60 70 80.
- i is 1: userNums[1] = userNums[6]. Array now: 80 70 30 40 50 60 70 80.
- i is 2: userNums[2] = userNums[5]. Array now: 80 70 60 40 50 60 70 80.
- i is 3: userNums[3] = userNums[4]. Array now: 80 70 60 50 50 60 70 80.
- i is 4: userNums[4] = userNums[3]. Array now: 80 70 60 50 50 60 70 80. *Uh-oh, where did 40 go?*

We failed to actually swap the array elements, instead the code just copies values in one direction. We need to add code to properly swap. We add a variable tempVal to temporarily hold `userNums[NUM_VALUES - 1 - i]` so we don't lose that element's value.

Figure 5.8.3: Program with proper swap: However, the program's output shows the array doesn't change.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index
   int tempVal = 0;            // Temp variable for swapping

   // Prompt user to input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Reverse array's elements
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      tempVal = userVals[i];                          // Temp for swap
      userVals[i] = userVals[NUM_ELEMENTS - 1 - i]; // First part of swap
      userVals[NUM_ELEMENTS - 1 - i] = tempVal;     // Second complete
   }

   // Print numbers
   printf("WnNew values: ");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d ", userVals[i]);
   }

   return 0;
}
```

```
Enter 8 integer values...
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80
New values: 10 20 30 40 50 60 70 80
```

The new values are not reversed. Again, let's manually trace the loop iterations.

- i is 0: userNums[0] = userNums[7]. Array now: 80 20 30 40 50 60 70 10.
- i is 1: userNums[1] = userNums[6]. Array now: 80 70 30 40 50 60 20 10.
- i is 2: userNums[2] = userNums[5]. Array now: 80 70 60 40 50 30 20 10.
- i is 3: userNums[3] = userNums[4]. Array now: 80 70 60 50 40 30 20 10. *Looks reversed.*
- i is 4: userNums[4] = userNums[3]. Array now: 80 70 60 40 50 30 20 10. *Why are we still swapping?*

Tracing makes clear that the for loop should not iterate over the entire array. The reversal is completed halfway through the iterations. The solution is to set the loop expression to `i < (NUM_VALUES / 2)`.

Figure 5.8.4: Program with correct loop bound: Running the program yields the correct output.

```
#include <stdio.h>

int main(void) {
   const int NUM_ELEMENTS = 8; // Number of elements
   int userVals[NUM_ELEMENTS]; // User numbers
   int i = 0;                  // Loop index
   int tempVal = 0;            // Temp variable for swapping

   // Prompt user to input values
   printf("Enter %d integer values...Wn", NUM_ELEMENTS);
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   // Reverse array's elements
   for (i = 0; i < (NUM_ELEMENTS / 2); ++i) {
      tempVal = userVals[i];                        // Temp for swap
      userVals[i] = userVals[NUM_ELEMENTS - 1 - i]; // First part of swap
      userVals[NUM_ELEMENTS - 1 - i] = tempVal;     // Second complete
   }

   // Print numbers
   printf("WnNew values: ");
   for (i = 0; i < NUM_ELEMENTS; ++i) {
      printf("%d ", userVals[i]);
   }

   return 0;
}
```

```
Enter 8 integer values...
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
Value: 60
Value: 70
Value: 80

New values: 80 70 60 50 40 30 20 10
```

We should ensure the program works if the number of elements is odd rather than even. Suppose the array has 5 elements (0-4) with values 10 20 30 40 50. NUM_VALUES / 2 would be 5 / 2 = 2, meaning the loop expression would be i < 2. The iteration when i is 0 would swap elements 0 and 4 (5-1-0), yielding 50 20 30 40 10. The iteration for i=1 would swap elements 1 and 3, yielding 50 40 30 20 10. The loop would then not execute again because i is 2. So the results are correct for an odd number of elements, because the middle element will just not move.

The mistakes made above are each very common when dealing with loops and arrays, especially for beginning programmers. An incorrect (in this case out-of-range) index, an incorrect swap, and an incorrect loop expression. The lesson is that loops and arrays require attention to detail, greatly aided by manually executing the loop to determine what is happening on each iteration. Ideally, a programmer will take more care when writing the original program, but the above mistakes are quite common.

PARTICIPATION ACTIVITY 5.8.1: Array reversal example.

Questions refer to the problematic example in this section.

1) The first problem was trying to access a non-existent element.
   - ○ True
   - ○ False

2) The second problem was failing to properly swap, using just this statement: userNums[i] = userNums[NUM_ELEMENTS - 1 - i]; // Swap
   - ○ True
   - ○ False

3) The third problem was that the loop did not iterate over all the elements, but rather stopped one short.
   - ○ True
   - ○ False

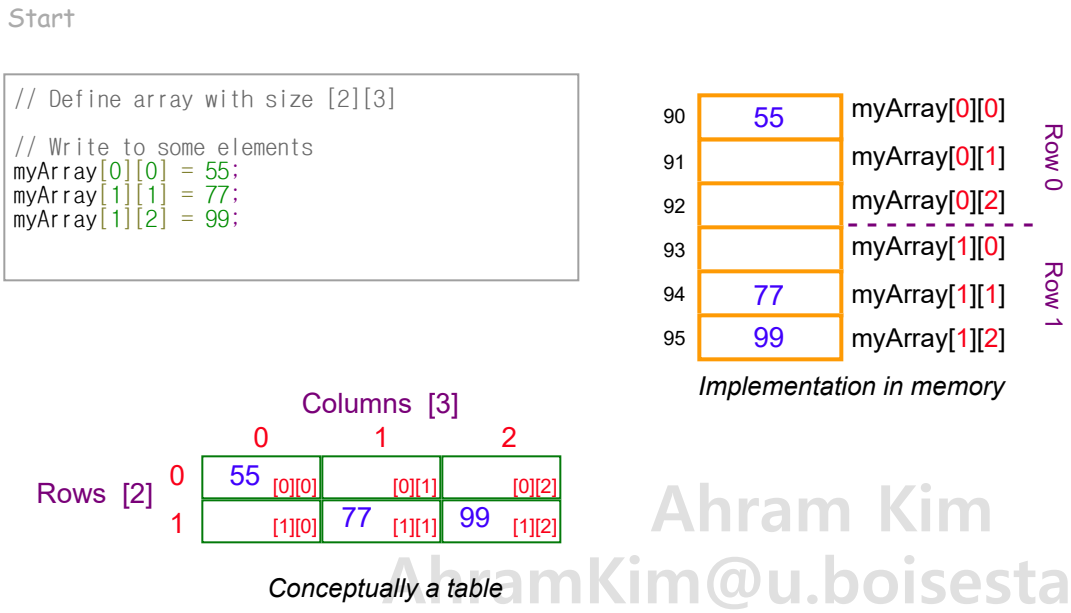
4) The programmer probably should have been more careful in creating the first version of the program.

- ○ True
- ○ False

5.9 Two-dimensional arrays

An array can be declared with two dimensions. `int myArray[R][C]` represents a table of int variables with R rows and C columns, so R*C elements total. For example, `int myArray[2][3]` creates a table with 2 rows and 3 columns, for 6 int variables total. Example accesses are `myArray[0][0] = 33;` or `num = myArray[1][2]`.

PARTICIPATION ACTIVITY 5.9.1: Two-dimensional array.



Conceptually, a two-dimensional array is a table with rows and columns. The compiler maps two-dimensional array elements to one-dimensional memory, each row following the previous row, known as ***row-major order***.

Figure 5.9.1: Using a two-dimensional array: A driving distance between cities example.

```c
#include <stdio.h>

/* Direct driving distances between cities, in miles */
/* 0: Boston  1: Chicago  2: Los Angeles */

int main(void) {
   int cityA = 0;              // Starting city
   int cityB = 0;              // Destination city
   int DrivingDistances[3][3]; // Driving distances

   // Initialize distances array
   DrivingDistances[0][0] = 0;
   DrivingDistances[0][1] = 960;  // Boston-Chicago
   DrivingDistances[0][2] = 2960; // Boston-Los Angeles
   DrivingDistances[1][0] = 960;  // Chicago-Boston
   DrivingDistances[1][1] = 0;
   DrivingDistances[1][2] = 2011; // Chicago-Los Angeles
   DrivingDistances[2][0] = 2960; // Los Angeles-Boston
   DrivingDistances[2][1] = 2011; // Los Angeles-Chicago
   DrivingDistances[2][2] = 0;

   printf("0: Boston  1: Chicago  2: Los AngelesWn");

   printf("Enter city pair (Ex: 1 2) -- ");
   scanf("%d %d", &cityA, &cityB);

   printf("Distance: %d milesWn", DrivingDistances[cityA][cityB]);

   return 0;
}
```

```
0: Boston  1: Chicago  2: Los Angeles
Enter city pair (Ex: 1 2) -- 1 2
Distance: 2011 miles

...

0: Boston  1: Chicago  2: Los Angeles
Enter city pair (Ex: 1 2) -- 2 0
Distance: 2960 miles

...

0: Boston  1: Chicago  2: Los Angeles
Enter city pair (Ex: 1 2) -- 1 1
Distance: 0 miles
```


A programmer can initialize a two-dimensional array's elements during declaration using nested braces, as below. Multiple lines make the rows and columns more visible.

Construct 5.9.1: Initializing a two-dimensional array during declaration.

```c
// Initializing a 2D array
int numVals[2][3] = { {22, 44, 66}, {97, 98, 99} };

// Use multiple lines to make rows more visible
int numVals[2][3] = {
   {22, 44, 66}, // Row 0
   {97, 98, 99}  // Row 1
};
```


Arrays of three or more dimensions can also be declared, as in `int myArray[2][3][5]`, which declares a total of 2*3*5 or 30 elements. Note the rapid growth in size -- an array declared as `int myArray[100][100][5][3]` would have 100*100*5*3 or 150,000 elements. A programmer should make sure not to unnecessarily occupy available memory with a large array.

PARTICIPATION ACTIVITY 5.9.2: Two-dimensional arrays.

1) Declare a two dimensional array of integers named dataVals with 4 rows and 7 columns.

Check Show answer

2) How many total elements are in an array with 4 rows and 7 columns?

Check Show answer

3) How many elements are in the array declared as: char streetNames[20][50];

Check Show answer

4) Write a statement that assigns 99 into the fifth row, third column of array dataVals. Note: the first row/column is at index 0, not 1.

Check Show answer

CHALLENGE ACTIVITY 5.9.1: Find 2D array max and min.

Find the maximum value and minimum value in milesTracker. Assign the maximum value to maxMiles, and the minimum value to minMiles. Sample output for the given program:

```
Min miles: -10
Max miles: 40
```

(Notes)

```
1 #include <stdio.h>
2
3 int main(void) {
```

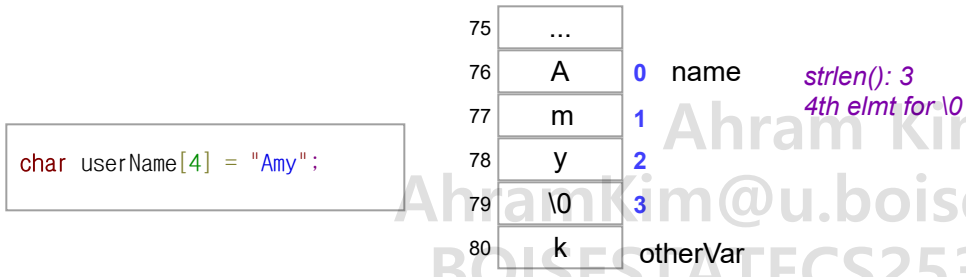
```
4    const int NUM_ROWS = 2;
5    const int NUM_COLS = 2;
6    int milesTracker[NUM_ROWS][NUM_COLS];
7    int i = 0;
8    int j = 0;
9    int maxMiles = -99; // Assign with first element in milesTracker before loop
10   int minMiles = -99; // Assign with first element in milesTracker before loop
11
12   milesTracker[0][0] = -10;
13   milesTracker[0][1] = 20;
14   milesTracker[1][0] = 30;
15   milesTracker[1][1] = 40;
16
17   /* Your solution goes here  */
18
19    printf("Min miles: %d\n", minMiles);
20    printf("Max miles: %d\n", maxMiles);
21
```

Run

5.10 Char arrays / C strings

A programmer can use an array to store a sequence of characters, known as a ***string***. An example is: `char movieTitle[20] = "Star Wars";`. Because a string can be shorter than the character array, a string in a char array must end with a special character known as a ***null character***, written as '\0'. Given a string literal like "Star Wars", the compiler automatically appends a null character.

PARTICIPATION ACTIVITY 5.10.1: A char array declaration with a compiler-added null character.



A char array of size 20 can store strings of lengths 0 to 19. The longest string is 19, not 20, since the null character must be stored.

If a char array is initialized when declared, then the char array's size may be omitted, as in `char userName[] = "Hellen";` . The compiler determines the size from the string literal, in this case 6

+ 1 (for the null character), or 7.

An array of characters ending with a null character is known as a ***null-terminated string***.

printf() automatically handle null-terminated strings, printing each character until reaching the null character that ends the string.

Figure 5.10.1: Printing stops when reaching the null character at each string's end.


```
#include <stdio.h>
int main(void) {
   char cityName[20] = "Forest Lake"; // Compiler appends null char

   // In each printf(), printing stops when reaching null char
   printf("%sWn", "City:");          // Compiler appends null char to "City:"
   printf("%sWn", cityName);

   return 0;
}
```

```
City:
Forest Lake
```

PARTICIPATION ACTIVITY 5.10.2: Char array strings.

Indicate whether the array declaration and initialization are appropriate.

1) `char firstName[10] = "Henry";`

- True
- False

2) `char lastName[10] = "Michelson";`

- True
- False

3) `char favoriteMuseum[10] = "Smithsonian";`

- True
- False

4) Given:

```
char catBreed[20] = "Persian";
```

Printing catBreed will print 19 characters.

- True
- False

After a string is declared, a programmer may not later assign the string as in `movieTitle = "Indiana Jones";`. That statement tries to assign a value to the char array variable itself, rather than copying each character from the string on the right into the array on the left. Functions exist to *copy* strings, such as strcpy(), discussed elsewhere.

A programmer can traverse a string using a loop that stops when reaching the null character.

A common error is to loop for the string's array size rather than stopping at the null character. Such looping visits unused array elements beyond the null character. An even worse common error is to loop beyond the last valid element, which visits memory locations that are not part of the array. These errors are illustrated below. Notice the strange characters that are output as the contents of other memory locations are printed out; the program may also crash.

Figure 5.10.2: Traversing a C string.

```
Enter string (<20 chars): test@gmail.com

test@gmail.com
Found '@'.

"test@gmail.com6789"
"test@gmail.com6789$W305W366;W226W333"
```

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:10

```
#include <stdio.h>

int main(void) {
   char  userStr[20] = "1234567890123456789"; // Input string
   int   i = 0;

   // Prompt user for string input
   printf("Enter string (<20 chars): ");
   scanf("%s", userStr);

   // Print string
   printf("Wn%sWn", userStr);

   // Look for '@'
   for (i = 0; userStr[i] != 'WO'; ++i) {
      if (userStr[i] == '@') {
         printf("Found '@'.Wn");
      }
   }

   // The following is an ERROR.
   // May print chars it shouldn't.
   // Problem: doesn't stop at null char.
```


The above output is machine and compiler dependent. Also, some values aren't printable so don't appear in the output.

PARTICIPATION ACTIVITY 5.10.3: C string errors.

Given the following char array declaration, which of the following code snippets are bad?

```
char userText[10] = "Car";
```

1)
```
for (i = 0; userText != 'WO'; ++i) {
   // Print userText[i]
}
```

- OK
- Bad

2)
```
for (i = 0; userText[i] != 'WO'; ++i) {
   // Print userText[i]
}
```

- OK
- Bad

3)
```
for (i = 0; i < 10; ++i) {
   // Print userText[i]
}
```

- OK
- Bad

4)

```
for (i = 0; i < 4; ++i) {
   // Print userText[i]
}
```

- OK
- Bad

5) `userText = "Bus";`

- OK
- Bad

Yet another common error with C strings is for the program user to enter a string larger than the character array. That may cause the input statement to write to memory locations outside the array's locations, which may corrupt other parts of program or data, and typically causes the program to crash.

PARTICIPATION ACTIVITY 5.10.4: Reading in a string too large for a C string.

Run the program, which simply reads an input string and prints it one character at a time. Then, lengthen the input string beyond 10 characters, and run again. The program *might* work, if the extra memory locations being assigned don't matter. Try larger and larger strings, and see if the program fails (be sure to scroll to the bottom of the output to look for erroneous output or an error message).

Load default template...

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     char  userStr[10]; // Input string
6     int   i = 0;
7
8     // Prompt user for string input
9     printf("Enter string (<10 chars): ");
10    scanf("%s", userStr);
11
12    // Print 1 char at a time
13    printf("\n");
14    for (i=0; userStr[i] != '\0'; ++i) {
15       printf("%c\n", userStr[i]);
16    }
17
18    return 0;
19 }
20
```

Hello

Run

The following program is for illustration, showing how a string is made up of individual character elements followed by a null character. Normally a programmer would not create a string that way.

Figure 5.10.3: A C string is an array of characters, ending with the null character.

```
#include <stdio.h>

int main(void) {
   char nameArr[9] = "";

   nameArr[0] = 'A';
   nameArr[1] = 'l';
   nameArr[2] = 'a';
   nameArr[3] = 'n';
   nameArr[4] = '\0'; // Null character

   printf("%s\n", nameArr);

   nameArr[4] = '!';  // Oops, overwrote null char
   printf("%s\n", nameArr); // *Might* still work

   return 0;
}
```

```
Alan
Alan!
```


When printing a string stored within a character array, each character within the array will be printed until the null character is reached. If the null character is omitted, the program would print whatever values are found in memory after the array, until a null character happens to be encountered. Omitting the null character is a serious logical error.

It just so happens that the null character '\0' has an ASCII encoding of 0. Many compilers initialize memory to 0s. As such, omitting the '\0' in the above program would not always cause erroneous execution. Like a nail in the road, that bug in your code is just waiting to wreak havoc.

PARTICIPATION ACTIVITY 5.10.5: C string without null character.

Given:

```
char userText[10];
userText[0] = 'C';
userText[1] = 'a';
userText[2] = 'r';
userText[3] = '\0';
...
userText[3] = 's';
```


1) The first four characters in userText are now: Cars.

   - ○ True
   - ○ False

2) The compiler generates an error, because element 3 is the null character and can't be overwritten.

   - ○ True

- False

3) Printing userText should work fine because the new string is 4 characters, which is still much less than the array size of 10.

- True
- False

5.11 String library functions

C provides common functions for working with C strings, presented in the ***string.h*** library. To use those functions, the programmer starts with: `#include <string.h>`.

Some C string functions for *modifying* strings are summarized below.

Table 5.11.1: Some C string modification functions.

Given:

```
char orgName[100] = "United Nations";
char userText[20] = "UNICEF";
char targetText[10];
```

strcpy()	`strcpy(destStr, sourceStr)`
	
	Copies sourceStr (up to and including null character) to destStr.
		`strcpy(targetText, userText); // Copies "UNICEF" + null char`
		`                               // to targetText`
		`strcpy(targetText, orgName);  // Error: "United Nations"`
		`                               // has > 10 chars`
		`targetText = orgName;         // Error: Strings can't be`
		`                               // copied this way`

strncpy()	`strncpy(destStr, sourceStr, numChars)` Copies up to numChars characters.	`strncpy(orgName, userText, 6); // orgName is "UNICEF Nations"`
strcat()	`strcat(destStr, sourceStr)` Copies sourceStr (up to and including null	`strcat(orgName, userText); // orgName is "United NationsUNICEF"`

	character) to *end* of destStr (starting at destStr's null character).	

strncat()	`strncat(destStr, sourceStr, numChars)`
	
	Copies up to numChars characters to destStr's end, then appends null character.
		`strcpy(targetText, "abc");  // targetText is "abc"`
		`strncat(targetText, "123456789", 3); // targetText is "abc123"`

For strcpy(), a common error is to copy a source string that is too large, causing an out-of-range access in the destination string. Another common error is to call strcpy with the source string first rather than the destination string, which copies in the wrong direction.

Note that string assignment, as in targetText = orgName, does not copy the string and should not be used. The exception is during initialization, as in `char userText[20] = "UNICEF";`, for which the compiler copies the string literal's characters into the array.

PARTICIPATION ACTIVITY 5.11.1: String modification functions.

Given: char userStr[5];
Do not type quotes in your answers.
If appropriate, type: Error

1) What is userStr after: strcpy(userStr, "Bye");

Check Show answer

2) If userStr is initially "Hi", what is userStr after: strcpy(userStr, "Bye");

Check Show answer

3) What is userStr after: strcpy(userStr, "Goodbye");

Check Show answer

4) What is userStr after: strncpy(userStr, "Goodbye", 4);

Check Show answer

5) If userStr is initially "Hi", what is userStr after: strcat(userStr, '!');

Check Show answer

6) If userStr is initially "Hi", what is userStr after: strcat(userStr, "!");

Check Show answer

7) If userStr is initially "Hi", what is userStr after: strncat(userStr, "?!$#@%", 2);

Check Show answer

Several C string functions that get *information* about strings are summarized below.

Table 5.11.2: Some C string information functions.

Given:

```
char orgName[100] = "United Nations";
char userText[20] = "UNICEF";
char targetText[10];
```

strchr()	`strchr(sourceStr, searchChar)`	

	Returns NULL if searchChar does not exist in sourceStr. (Else, returns address of first occurrence, discussed elsewhere).
	NULL is defined in the string.h library.
		`if (strchr(orgName, 'U') != NULL) { // 'U' exists in orgName?`
		`  ... // 'U' exists in "United Nations", branch taken`
		`}`
		`if (strchr(orgName, 'u') != NULL) { // 'u' exists in orgName?`
		`  ... // 'u' doesn't exist (case matters), branch not taken`
		`}`

strlen()	`size_t strlen(sourceStr)`
	Returns number of characters in sourceStr up to, but not including, first null character. size_t is integer type.
		`x = strlen(orgName);  // Assigns 14 to x`
		`x = strlen(userText);  // Assigns 6 to x`
		`x = strlen(targetText); // Error: targetText may lack null char`

strcmp()	`int strcmp(str1, str2)`
	Returns 0 if str1 and str2 are equal, non-zero if they differ.
		`if (strcmp(orgName, "United Nations") == 0) {`
		`  ... // Equal, branch taken`
		`}`
		`if (strcmp(orgName, userText) == 0) {`
		`  ... // Not equal, branch not taken`
		`}`

strcmp() is usually used to compare for equality, returning 0 if the strings are the same length and have identical characters. A common error is to use == when comparing C strings, which does not work. str1 == str2 compares the strings' addresses, not their contents. Because those addresses will usually be different, str1 == str2 will evaluate to 0. This is not a syntax error, but clearly a logic error. Another common error is to forget to compare the result of strcmp with 0, as in `if (strcmp(str1, str2)) {...}`. The code is not a syntax error, but is a logic error because the if condition will be false (0) when the strings are equal. The correct condition would instead be `if (strcmp(str1, str2) == 0) {...}`. Although strcmp returns 0, a good practice is to avoid using `if (!strcmp(str1,str2)) {...}` because that 0 does not represent "false" but rather is encoding a particular situation.

strcmp(str1, str2) returns a negative number if str1 is less than str2, and a positive number if str1 is greater than str2. Evaluation first compares the character pair at element 0, then at element 1, etc., returning as soon as a pair differs.

PARTICIPATION ACTIVITY 5.11.2: String comparison.

Start

	0	1	2	3	4	5	6	7
studentName	K	a	y	,	_	J	o	
teacherName	K	a	y	,	_	A	m	y

strcmp(studentName, teacherName) *evaluates to positive number 9*

Each comparison uses ASCII values

75	97	121	44	32	74
75	97	121	44	32	65
0	0	0	0	0	**9**

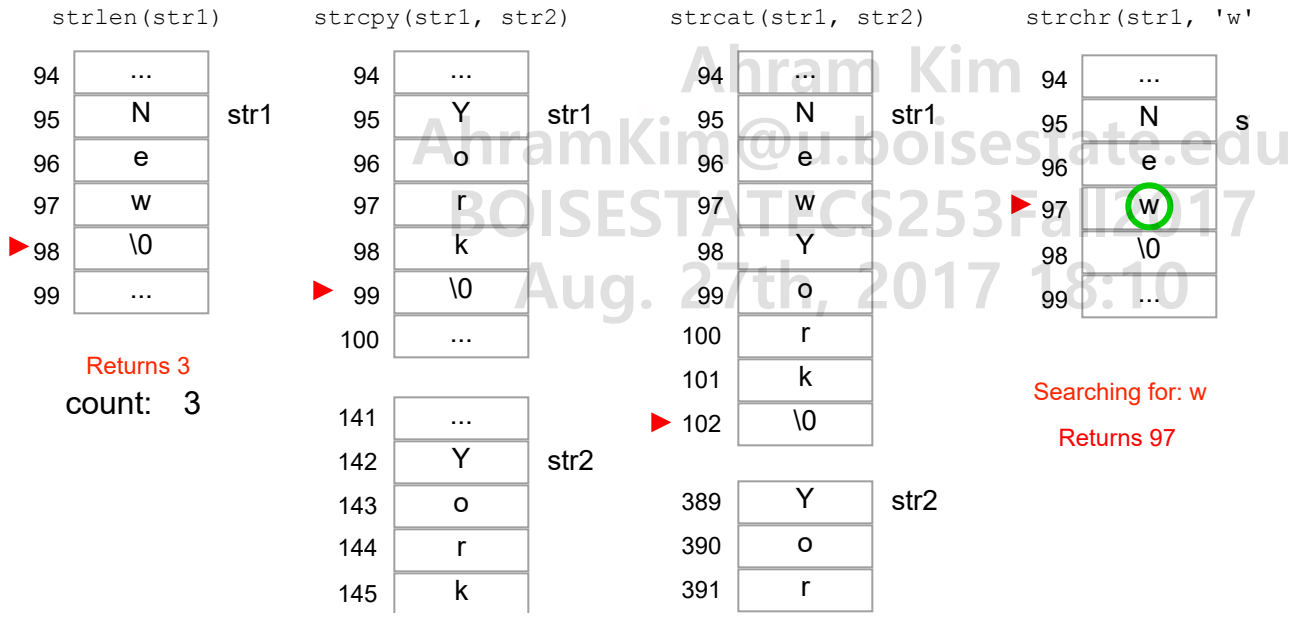
strlen is often used to iterate through each string character in a loop.

Figure 5.11.1: Iterating through a C string using strlen.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char userName[15] = "Alan Turing";
   int i = 0;

   printf("Before: %sWn", userName);

   for (i = 0; i < strlen(userName); ++i) {
      if (userName[i] == ' ') {
         userName[i] = '_';
      }
   }
   printf("After:  %sWn", userName);

   return 0;
}
```

```
Before: Alan Turing
After:  Alan_Turing
```

PARTICIPATION ACTIVITY 5.11.3: Some C string library functions.



▶ 146	\0

392	k
▶ 393	\0

PARTICIPATION ACTIVITY 5.11.4: String information functions.

Given:

```
char str1[10] = "Earth";
char str2[20] = "Earthlings";
char str3[15] = "Mars";
```

Answer the following questions. If appropriate, type: Error

1) What does strlen(str3) return?

Check Show answer

2) Is the branch taken? (Yes/No/Error)

```
if (strchr(str1, '@') != NULL) {
   // Print "Found @"
}
```

Check Show answer

3) Is the branch taken? (Yes/No/Error)

```
if (strchr(str1, 'E') != NULL) {
   // Print "Found E"
}
```

Check Show answer

4) Is the branch taken? (Yes/No/Error)

```
if (strchr(str2, "Earth") != NULL) {
   // Print "Found Earth"
}
```

Check Show answer

5) Is the branch taken? (Yes/No/Error)

```
if (strcmp(str1, str2) == 0) {
   // Print "strings are equal"
}
```

Check Show answer

6) Is the branch taken? (Yes/No/Error)

```
if (str1 == str3) {
   // Print "strings are equal"
}
```

Check Show answer

7) Finish the code to take the branch if str1 and str3 are equal.

```
if (strcmp(str1, str3)
) {
   // Strings are equal
}
```

Check Show answer

scanf() and fgets() in stdio.h

stdio.h has several functions to support C strings. ***scanf*** can be used to read a string from the user input. For example, `scanf("%s", myString)` reads a string from the user input into myString, where a string is a sequence of characters excluding spaces, tabs, or newline. If a user types "John Smith<enter>", then myString will just be "John" because the string ends at the space after the 'n'.

Allowing spaces in a user's string input can be accomplished using the ***fgets*** function. `fgets(str, num, stdin)` reads one line of characters from user input, ending with a newline, and writes those characters into the C string str. The read characters may include spaces and tabs. If a newline character is read from the user input before num characters are read, the newline character itself is also written into str, after which the function appends a null character. num is the maximum number of characters to be written into str. If num is 10 and the input line exceeds 10 characters, only

the first 9 characters will be written into str, followed by the null character; the remaining input characters will not be read and will remain in user input.

The following example asks the user to enter a name, and then creates and modifies a string involving that name and other text.

Figure 5.11.2: C string modification example.


```
#include <stdio.h>
#include <string.h>

int main(void) {
   char nameArr[10] = "";          // User specified name
   char greetingArr[17] = "";      // Output greeting and name

   // Prompt user to enter a name
   printf("Enter name: ");
   fgets(nameArr, 10, stdin);

   // Eliminate end-of-line char
   nameArr[strlen(nameArr)-1] = '\0';

   // Modify string, hello + user specified name
   strcpy(greetingArr, "Hello ");
   strcat(greetingArr, nameArr);
   strcat(greetingArr, ".");

   // Output greeting and name
   printf("%s\n", greetingArr);

   return 0;
}
```

```
Enter name: Al Smith
Hello Al Smith.

...

Enter name: Mary Johnson
Hello Mary Joh.
```

PARTICIPATION ACTIVITY 5.11.5: fgets and scanf.

1) scanf() reads an entire line of text, including spaces.
   - True
   - False

2) fgets' first parameter is the string into which a line of input text will be read.
   - True
   - False

3) fgets' second parameter is the number of characters to be written into the string parameter.
   - True

False

4) If a user types "Hi there" and presses enter, fgets' string parameter will contain the string "Hi there" ending with a null character.

True

False

Exploring further:

- More C string functions from cplusplus.com (applies to C)

5.12 Char library functions: ctype

C provides common functions for working with characters, presented in the ctype.h library, short for "character type". To use those functions, the programmer adds the following at the top of a file: `#include <ctype.h>`.

Commonly-used ctype.h functions are summarized below; a complete reference is found at http://www.cplusplus.com/reference/cctype/ (applies to C).

Character checking functions

The following functions check whether a character is of a given category, returning either false (0) or true (non-zero). The examples below assume the following string declaration.

```
char myString[30] = "Hey9! Go";
```


- ***isalpha(c)*** -- Returns true if c is alphabetic: a-z or A-Z.
  - `isalpha('A'); // Returns true`
  - `isalpha(myString[0]); // Returns true because 'H' is alphabetic`
  - `isalpha(myString[3]); // Returns false because '9' is not alphabetic`
- ***isdigit(c)*** -- Returns true if c is a numeric digit: 0-9.
  - `isdigit(myString[3]); // Returns true because '9' is numeric`
  - `isdigit(myString[4]); // Returns false because ! is not numeric`
- ***isalnum(c)*** -- Returns true if c is alphabetic or a numeric digit. Thus, returns true if either isalpha or isdigit would return true.

- ***isspace(c)*** -- Returns true if character c is a whitespace.
  - `isspace(myString[5]);` // Returns true because that character is a space ' '.
  - `isspace(myString[0]);` // Returns false because 'H' is not whitespace.

- ***islower(c)*** -- Returns true if character c is a lowercase letter a-z.
  - `islower(myString[0]);` // Returns false because 'H' is not lowercase.
  - `islower(myString[1]);` // Returns true because 'e' is lowercase.
  - `islower(myString[3]);` // Returns false because '9' is not a lowercase letter.

- ***isupper(c)*** -- Returns true if character c is an uppercase letter A-Z.

- ***isblank(c)*** -- Returns true if character c is a blank character. Blank characters include spaces and tabs.
  - `isblank(myString[5]);` // Returns true because that character is a space ' '.
  - `isblank(myString[0]);` // Returns false because 'H' is not blank.

- ***isxdigit(c)*** -- Returns true if c is a hexadecimal digit: 0-9, a-f, A-F.
  - `isxdigit(myString[3]);` // Returns true because '9' is a hexadecimal digit.
  - `isxdigit(myString[1]);` // Returns true because 'e' is a hexadecimal digit.
  - `isxdigit(myString[6]);` // Returns false because 'G' is not a hexadecimal digit.

- ***ispunct(c)*** -- Returns true if c is a punctuation character. Punctuation characters include: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
  - `ispunct(myString[4]);` // Returns true because '!' is a punctuation character.
  - `ispunct(myString[6]);` // Returns false because 'G' is not a punctuation character.

- ***isprint(c)*** -- Returns true if c is a printable character. Printable characters include alphanumeric, punctuation, and space characters.

- ***iscntrl(c)*** -- Returns true if c is a control character. Control characters are all characters that are not printable.

Character conversion functions

The following functions return a character representing a converted version of the input character. The examples below assume the following string declaration.

```
char myString[30] = "Hey9! Go";
```

- ***toupper(c)*** -- If c is a lowercase alphabetic character (a-z), returns the uppercase version (A-Z). If c is not a lowercase alphabetic character, just returns c.
  - `toupper(myString[0]);` // Returns 'H' (no change)
  - `toupper(myString[1]);` // Returns 'E' ('e' converted to 'E')
  - `toupper(myString[3]);` // Returns '9' (no change)
  - `toupper(myString[5]);` // Returns ' ' (no change)

- ***tolower(c)*** -- If c is an uppercase alphabetic character (A-Z), returns the lowercase version (a-z). If c is not an uppercase alphabetic character, just returns c.

The following example illustrates some of the ctype.h functions.

Figure 5.12.1: Use of some functions in ctype.h.

```
#include <stdio.h>
#include <ctype.h>

int main(void) {
   const int MAX_LEN = 30; // Max string length
   char userStr[MAX_LEN]; // User defined string
   int  i = 0;

   // Prompt user to enter string
   printf("Enter string (<%d chars): ", MAX_LEN);
   scanf("%s", userStr);

   printf("Original: %sWn", userStr);

   printf("isalpha:  ");
   for (i = 0; userStr[i] != 'WO'; ++i) {
      printf("%d", isalpha(userStr[i]));
   }
   printf("Wn");

   printf("isdigit:  ");
   for (i = 0; userStr[i] != 'WO'; ++i) {
      printf("%d", isdigit(userStr[i]));
   }
   printf("Wn");

   printf("isupper:  ");
   for (i = 0; userStr[i] != 'WO'; ++i) {
      printf("%d", isupper(userStr[i]));
   }
   printf("Wn");

   for (i = 0; userStr[i] != 'WO'; ++i) {
      userStr[i] = toupper(userStr[i]);
   }
   printf("After toupper: %sWn", userStr);

   return 0;
}
```

```
Enter string (<30 chars): ABC123$%&def
Original: ABC123$%&def
isalpha:  111000000111
isdigit:  000111000000
isupper:  111000000000
After toupper: ABC123$%&DEF
```


PARTICIPATION ACTIVITY 5.12.1: Character type functions.

Enter the value to which each function evaluates, using 1 for true, 0 for false. Assume str is "Hi 321!".

1) isalpha(str[0])

Check Show answer

2) isdigit(str[4])

Check Show answer

3) isalnum(str[2])

Check Show answer

4) isspace(str[2])

Check Show answer

5) islower(str[6])

Check Show answer

6) tolower(str[0])

Check Show answer

7) tolower(str[1])

Check Show answer

To compare two strings without paying attention to case, one technique is to first convert (a copy of) each string to lowercase (using a loop, discussed elsewhere) and then comparing.

5.13 Arrays and strings

Because C strings are stored using arrays of characters, an array of strings can be created using a two-dimensional char array. For example, `char studentNames[5][30];` creates an array of 5 strings, in which each string contains up to 30 characters. Strings within the array are accessed using only the first array index, e.g., `studentNames[0]` will access the first string in the array. The following example illustrates the use of an array of strings.

Figure 5.13.1: Array of strings example: Top 10 countries for most TV watched daily.


```
#include <stdio.h>
#include <string.h>

// Source: www.statista.com, 2011

int main(void) {
   const int NUM_COUNTRY = 10;                          // Number of countries supported
   const int MAX_COUNTRY_NAME_LENGTH = 50;              // Max length for names
   char ctryNames[NUM_COUNTRY][MAX_COUNTRY_NAME_LENGTH]; // 2D array of country tv stats
   int arrPosition = 0;                                 // User specified position
   int i = 0;

   // Populate array
   strcpy(ctryNames[i], "U.S.A.");      ++i;
   strcpy(ctryNames[i], "Italy");       ++i;
   strcpy(ctryNames[i], "Poland");      ++i;
   strcpy(ctryNames[i], "U.K.");        ++i;
   strcpy(ctryNames[i], "Canada");      ++i;
   strcpy(ctryNames[i], "Spain");       ++i;
   strcpy(ctryNames[i], "France");      ++i;
   strcpy(ctryNames[i], "Germany");     ++i;
   strcpy(ctryNames[i], "Brazil");      ++i;
   strcpy(ctryNames[i], "Russia");      ++i;

   // Prompt user to enter desired position
   printf("Enter desired position (1-10): ");
   scanf("%d", &arrPosition);

   // Print results
   printf("People in %s watch the %d", ctryNames[arrPosition-1], arrPosition);
   if( arrPosition == 1 ) {
      printf("st");
   }
   else if( arrPosition == 2 ) {
      printf("nd");
   }
   else if( arrPosition == 3 ) {
      printf("rd");
   }
   else {
      printf("th");
   }
   printf(" most TV per day.Wn");

   return 0;
}
```

```
Enter desired position (1-10): 1
People in U.S.A. watch the 1st most TV per day.

...

Enter desired position (1-10): 3
People in Poland watch the 3rd most TV per day.

...

Enter desired position (1-10): 10
People in Russia watch the 10th most TV per day.
```


Note that the initialization of the array uses strcpy() (e.g., `strcpy(ctryNames[i], "U.S.A.");`) to the copy the the string literal for the country names to the corresponding string within the array ctryNames.

PARTICIPATION ACTIVITY 5.13.1: Arrays of strings.

1) Declare an array named favoriteBooks that stores 20 strings each with a maximum of 75 characters.

Show answer

Check Show answer

2) Using strcpy(), write a statement to copy the string "Hamlet" into the second string of the array favoriteBooks.

Check Show answer

3) Write a statement to print the last string in the array.

Check Show answer

5.14 C example: Salary calculation with arrays

PARTICIPATION ACTIVITY 5.14.1: Various tax rates.

Arrays are useful to process tabular information. Income taxes are based on annual salary, usually with a tiered approach. Below is an example of a simple tax table:

Annual Salary	Tax Rate
0 to 20000	10%
Above 20000 to 50000	20%
Above 50000 to 100000	30%
Above 100000	40%

The below program uses an array salaryBase to hold the cutoffs for each salary level and a parallel array taxBase that has the corresponding tax rate.

1. Run the program and enter annual salaries of 40000 and 60000, then enter 0.
2. Modify the program to use two parallel arrays named annualSalaries and taxesToPay, each with 10 elements. Array annualSalaries holds up to 10 annual salaries entered; array taxesToPay holds up to 10 corresponding amounts of taxes to pay for those annual salaries. Print the total annual salaries and taxes to pay after all input has been processed.

3. Run the program again with the same annual salary numbers as above.
4. Challenge: Modify the program from the previous step to use a 2-dimensional array of 10 elements named salariesAndTaxes instead of two one-dimensional parallel arrays. The 2D array's first column will hold the salaries, the second the taxes to pay for each salary.

The following program calculates the tax rate and tax to pay based on annual income.


```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
   const int BASE_TABLE_ELEMENTS = 5;
   const int MAX_ELEMENTS = 10;
   int annualSalary       = 0;
   double taxRate         = 0.0;
   int taxToPay           = 0;
   int numSalaries        = 0;
   bool keepLooking       = true;
   int i = 0;

   int salaryBase[] = { 20000,     50000,     100000,    999999999 };
   double taxBase[] = {   .10,       .20,        .30,          .40 };
   // FIXME: Declare annualSalaries and taxesToPay arrays to hold 10 elements each.
   // FIXME: Use the final constant MAX_ELEMENTS to declare the arrays

   printf("\nEnter annual salary (0 to exit): \n");
   scanf("%d", &annualSalary);

```

```
40000 60000 0
```

Run

A solution to above problem follows.

PARTICIPATION ACTIVITY 5.14.2: Solution to salaries array.

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
   const int BASE_TABLE_ELEMENTS = 5;
   const int MAX_ELEMENTS = 10;
   int annualSalary       = 0;
```

```
    double taxRate         = 0.0;
    int taxToPay           = 0;
    int totalSalaries      = 0;
    int totalTaxes         = 0;
    int numSalaries        = 0;
    bool keepLooking       = true;
    int i = 0;

    int salaryBase[] = { 20000,     50000,     100000,    999999999 };
    double taxBase[] = {   .10,       .20,        .30,          .40 };
    int annualSalaries[MAX_ELEMENTS];
    int taxesToPay[MAX_ELEMENTS];

    printf("\nEnter annual salary (0 to exit): \n");
```

```
40000 60000 0
```


Run

5.15 C example: Domain name validation with arrays

PARTICIPATION ACTIVITY 5.15.1: Validate domain names with arrays.

Arrays are useful to process lists.

A ***top-level domain*** (TLD) name is the last part of an Internet domain name like .com in example.com. A ***core generic top-level domain*** (core gTLD) is a TLD that is either .com, .net, .org, or .info. A ***restricted top-level domain*** is a TLD that is either .biz, .name, or .pro. A ***second-level domain*** is a single name that precedes a TLD as in apple in apple.com.

The following program repeatedly prompts for a domain name, and indicates whether that domain name consists of a second-level domain followed by a core gTLD. Valid core gTLD's are stored in an array. For this program, a valid domain name must contain only one period, such as apple.com, but not support.apple.com. The program ends when the user presses just the Enter key in response to a prompt.

1. Run the program and enter domain names to validate.
2. Extend the program to also recognize restricted TLDs using an array, and statements to validate against that array. The program should also report whether the TLD is a core gTLD or a restricted gTLD. Run the program again.

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

int main() {

   // Define the list of valid core gTLDs
   const int NUM_ELEMENTS = 4;
   const int MAX_LENGTH = 5;
   char validCoreGtld[NUM_ELEMENTS][MAX_LENGTH];
   // FIXME: Declare an array named validRestrictedGtld that has the names
   //        of the restricted domains, .biz, .name, and .pro
   char inputName[50]     = "";
   char searchName[50]    = "";
   char theGtld[50]       = "";
   bool isValidDomainName = false;
   bool isCoreGtld        = false;
   bool isRestrictedGtld  = false;
   int periodCounter      = 0;
   int periodPosition     = 0;
```



```
apple.com
APPLE.com
apple.comm
```

Run

PARTICIPATION ACTIVITY 5.15.2: Validate domain names with arrays (solution).

A solution to the problem posed above follows.


```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

int main(void) {
   // Define the list of valid core gTLDs
   const int NUM_ELEMENTS_CORE = 4;
   const int MAX_SIZE     = 6;
   const int NUM_ELEMENTS_RSTR = 3;
   char  inputName[50]    = "";
   char  searchName[50]   = "";
   char  theGtld[50]      = "";
   bool isValidDomainName = false;
   bool isCoreGtld        = false;
   bool isRestrictedGtld  = false;
   int periodCounter      = 0;
```

```
18    int periodPosition     = 0;
19    char validCoreGtld[NUM_ELEMENTS_CORE][MAX_SIZE];
20    char validRestrictedGtld[NUM_ELEMENTS_RSTR][MAX_SIZE];
21    int i = 0;
```

```
apple.com
APPLE.com
apple.comm
```

Run

Ahram Kim
AhramKim@u.boisestate.edu
BOISESTATECS253Fall2017
Aug. 27th, 2017 18:10