Chapter 6
Arrays

6.1 Chapter Overview

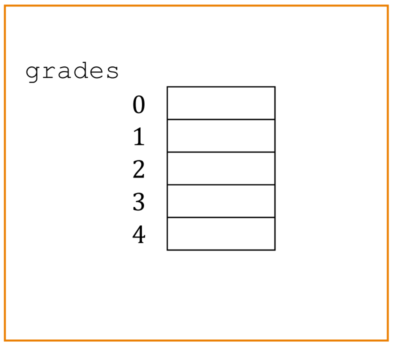
Arrays are data structures that store sets of data items under a single name declaration. Each element of the array must be of the same type. When allocated, an array takes up a contiguous block of memory. The elements can be accessed via an **index** (a non-negative integer).

6.2 Declaring Arrays

An array is declared like any other variable (`<type>` then `<variable name>`) but also included is the number of items in the array (the size must be specified when declaring an array). When an array is declared, memory is allocated for the array, and it remains that size. Each element is accessed by an **index** (or **subscript**), beginning with the zero-ith element. So, for example, if you need to store a set of 5 integer grades, you can declare an array like the following:

```
    int grades[5];
```

where the array will have space for 5 integers indexed from 0-4: grades[0], grades[1], grades[2], grades[3], & grades[4] – each one can contain an integer. If you need to access the 3rd item, you would access `grades[2]` or `grades[n-1]` where n is the total number of elements in the array. In memory, you can picture it looking something like the following:



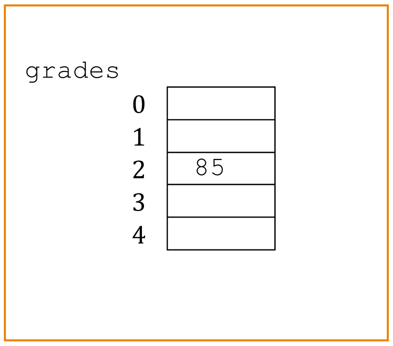
Once allocated, this allows you to have one variable name for a set of same-type items.

You can use the entire array as a single object, or each item in the array individually, as shown below.

An individual array element can be used anywhere that a normal variable can be used. For example, if you want to assign the 3rd element the value of 85, then you would write

```
    grades[2] = 85;
```

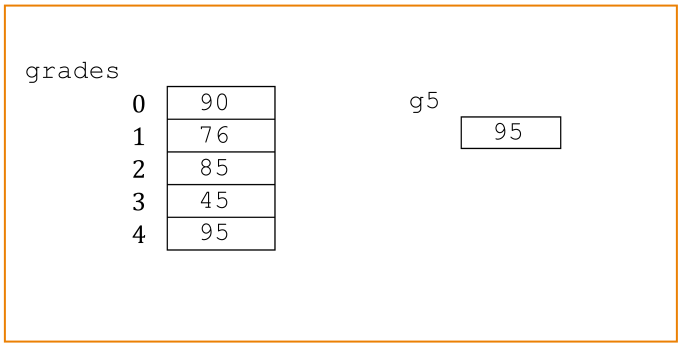
Now our array would look something like this:



Or, if you have an integer variable called `g5` and you want to assign to it the value of the 5th element (assuming it has a value), you would write

```
    g5 = grades[4];
```

which then gives `g5` the value of whatever that 5th element is in the array. In memory, with the array already filled in with values and after the above assignment statement, you can picture something like the following:



This construct is a powerful storage mechanism. Arrays make it much easier for handling large data sets without having to declare each of the items as individual variables. They also allow you to perform subtasks, like filling the elements of an array from user input or a file, displaying all of the elements of an array in forward or reverse order, sorting the values of an array in ascending or descending order, determining the sum, average, or some other statistic from the array values, and finding the highest or lowest value.

With a `for` loop, for example, you can easily access each item in a large array. Finding the lowest value, or adding up all the values is much easier to implement using an array.

The following loop will sum up all the values in an array of 100 items:

```
    for ( i = 0; i< 100; ++i)
        sum += grades[i];
```

Remember to start with zero or you will miss the first item.

WARNING – MAJOR ARRAY PITFALL: C will let you use a subscript that is outside of the bounds of the array. You will not get a syntax error and C does not catch it for you during execution. If you go outside of the bounds of the array, you may read or write other storage in your program, or you may get a segmentation fault (segfault) or bus error – i.e. tried to read or write on someone else's address space. Or you may read or write other variables in your program. The results are unpredictable. It's up to you to keep your subscript value within the bounds of the array.

TIP: When specifying the size of an array, you can use a named constant or #defined value, such as with the following:

```
const int NUMBER_OF_STUDENTS = 50;

// ...

int scores[NUMBER_OF_STUDENTS];
```

OR

```
#define NUMBER_OF_STUDENTS 50

// ...

int scores[NUMBER_OF_STUDENTS];
```

This is helpful for a couple of reasons. First, it improves program readability. It also makes the program more versatile and maintainable because you can change any reference to the size of the array in one easy to find location.

6.3 Initializing Arrays

If the initial values of an array are known, the elements in the array can be initialized when declared, such as:

```
int counters[5] = { 0, 0, 0, 1, 5 };
```

The above line of code sets the value of `counters[0]` to 0, `counters[1]` to 0, `counters[2]` to 0, `counters[3]` to 1, and `counters[4]` to 5. This would be equivalent to the following 6 lines of code:

```
int counters[5];
counters[0] = 0;
counters[1] = 0;
counters[2] = 0;
counters[3] = 1;
counters[4] = 5;
```

An array of characters can be initialized in a similar manner:

```
char vowels[5] = { 'a', 'e', 'i', 'o', 'u' };
```

It's not necessary to completely initialize the entire array. If the first few are initialized, the remaining are set to zero (0). Consider the following:

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

The first 3 elements are initialized to the values provided; the remaining 497 elements will be set to zero (0). However, no assumptions may be made about the values of the elements of an uninitialized array (i.e. they are NOT zero (0) by default).

If all the elements need to be initialized to something other than zero (0), the best way to do that is to use a for loop.

```
int array_values[10];
int i;

for  ( i = 0; i < 10; ++i )
     array_values[i] = i * i;
```

This for loop initializes each element to the square of the element number (subscript number), so the array will contain 0, 1, 4, 9, 16 25, 36, 47, 64, 81 after initialization.

C allows you to declare an array without specifying the number of elements only if you initialize every element of the array when it is declared. The following:

```
int counters[] = { 0, 0, 0, 1, 5 };
```

will implicitly dimension the array to 5 elements.

You can also use the index numbers when initializing. So if you know specific elements need to be initialized to a certain value, you can do the following:

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

Because the largest index number specified above is 99, `sample_data` will be set to contain 100 elements; the remaining uninitialized elements initialized to zero (0).

6.4 Multidimensional Arrays

The C language allows arrays of any dimension to be defined. For example, here is a 2-dimensional array, a matrix:

columns j

rows i	0	1	2	3	4
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	32	1	0	14
3	0	0	8	7	6

In this matrix `M`, `M[i][j]` refers to the element in the *i*th row and *j*th column. So, `M[2][1]` refers to the element with the value of 32; `M[0][4]` refers to the element with the value of 82; `M[2][3]` refers to the element with the value 0, and so on.

The statement `int sum = M[0][2] + M[2][4];` would result in `sum` being equal to `11` (-3 + 14).

Two-dimensional arrays are declared the same way as one-dimensional arrays. So, `int M[4][5];` would declare a 2-dimensional array consisting of 4 rows and 5 columns, for a total of 20 elements. Each position in the array is defined to contain an integer value.

Initializing a 2-dimensional array, like the one above, can be done like the following:

```
int M[4][5] = {
                   { 10,  5, -3, 17, 82 },
                   {  9,  0,  0,  8, -7 },
                   { 32, 20,  1,  0, 14 },
                   {  0,  0,  8,  7,  6 )
              };
```

or, it could have been initialized this way:

```
int M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, 7, -7, 32, 20, 1, 0, 14, 0, 0, 8, 7, 6 }
```

As with 1-dimensional arrays, it is not required that the entire array be initialized. A statement such as the following only initializes the first 3 elements of each row; the remaining values are initialized to zero (0) (the inner pairs of braces *are* required in this case or else these values would fill up the first couple two rows + the first 2 values of the 3rd row, and the remaining elements would be set to zero (0)).

```
int M[4][5] = {
                   { 10,  5, -3 },
                   {  9,  0,  0 },
                   { 32, 20,  1 },
                   {  0,  0,  8 )
              };
```

Subscripts can also be used to initialize specific elements (the remaining are set to zero (0)):

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

6.5 Variable-length Arrays

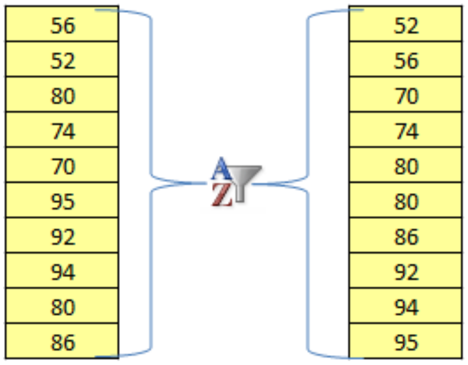
The C language allows you to declare arrays of a variable size. The following example prompts the user for the number of values, and that number is used to declare an array of that length.

```
printf (" How many Fibonacci numbers do you want ( between 1 and 75)? ");
scanf ("%i", &numFibs);

unsigned long long int Fibonacci[numFibs];
```

This is called a variable length array because the size is specified by a variable and not by a constant expression. In this case, the array cannot be declared at the top of the function with all the other variables; it cannot be declared until that value is entered by the user.

6.6 Sorting



Sorting is the process of arranging data into some type of order, generally ascending or descending. If not specified, ascending is assumed. Sorted ascending means that the lowest value is first and each successive value is greater than or equal to the previous value. Sorted descending means that the highest value is first and each successive value is less than or equal to the previous value.

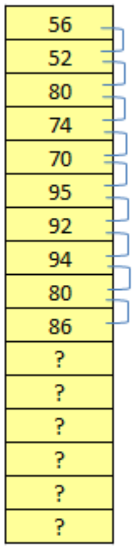
There are many different types of sorts. Some are more efficient than others, and some are better for different types of data or conditions. Some examples include: bubble, exchange, selection, insertion, shell, comb, merge, heap, quick, counting, bucket, radix, distribution, timsort, gnome, cocktail, library, cycle, binary tree, bogo, pigeonhole, spread, bead, pancake, . . .

Sorting algorithms have always received much attention from computer scientists because it is an operation that is so commonly performed. There are many sophisticated algorithms that sort in the least amount of time using as little of memory as possible.

Sorts often involve swapping the values of two variables. How do you swap two variables' values?

NOTE: If you are a visual learner, Google "YouTube sorting with Hungarian dancers" to view some of the different types of sorts in action.

A. **Bubble Sort** is probably one of the oldest, easiest, straight-forward, but inefficient sorting algorithms. It is the algorithm introduced as a sorting routine in most introductory courses on Algorithms. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list (if ascending) whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list. An implementation in C:



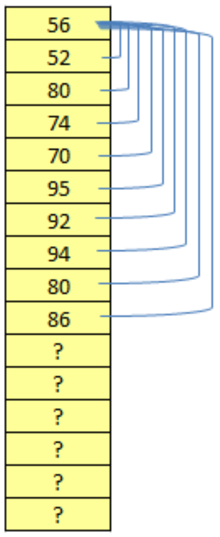
```
void bubbleSort(int a[], int array_size)
{
     int i, j, temp;
     for (i = 0; i < (array_size - 1); ++i)
     {
          for (j = 0; j < array_size - 1 - i; ++j )
          {
               if (a[j] > a[j+1])
               {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
               }
          }
     }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8 6 10 3 1 2 5 4	*pass 0*
6 8 3 1 2 5 4 10	*pass 1*
6 3 1 2 5 4 8 10	*pass 2*
3 1 2 5 4 6 8 10	*pass 3*
1 2 3 4 5 6 8 10	*pass 4*
1 2 3 4 5 6 8 10	*pass 5*
1 2 3 4 5 6 8 10	*pass 6*
1 2 3 4 5 6 8 10	*pass 7*

The table to the left clearly depicts how the bubble sort works. Note that each pass results in one number being bubbled to the end of the list.

B. The **Exchange Sort** , is similar to its cousin, the bubble sort, in that it compares two elements, swapping them if necessary. The difference is the exchange sort compares the first item in the array with each of the remaining elements, making any necessary swaps. After the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array, swapping when necessary. An implementation in C:



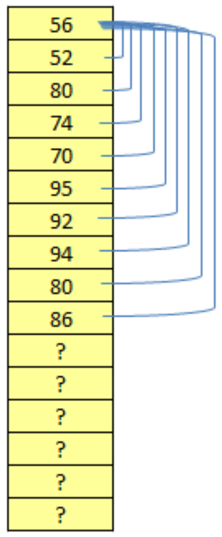
```
void exchangeSort(int a[], int array_size)
{
     int i, j, temp;
     for (i = 0; i < (array_size - 1); ++i)
     {
          for (j = i + 1; j < array_size; ++j )
          {
               if (a[i] > a[j])
               {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
               }
          }
     }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting.)

8 6 10 3 1 2 5 4	*pass 0*
1 8 10 6 3 2 5 4	*pass 1*
1 2 10 8 6 3 5 4	*pass 2*
1 2 3 10 8 6 5 4	*pass 3*
1 2 3 4 10 8 6 5	*pass 4*
1 2 3 4 5 10 8 6	*pass 5*
1 2 3 4 5 6 10 8	*pass 6*
1 2 3 4 5 6 8 10	*pass 7*

The table to the left clearly depicts how each exchange sort works. Note that each pass results in the lowest number being put in the first element, then the next lowest in the second element, etc.

C.   The idea of **Selection Sort** is rather simple. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where its supposed to be. The process is repeated such that the *n*th minimum (or maximum) element is swapped with the element at the *n-1*th index of the list. Below is an implementation of the algorithm in C:



```
void selectionSort(int a[], int array_size)
{
     int i;
     for (i = 0; i < array_size - 1; ++i)
     {
          int j, min, temp;
          min = i;
          for (j = i+1; j < array_size; ++j)
          {
               if (a[j] < a[min])
                    min = j;
          }

          temp = a[i];
          a[i] = a[min];
          a[min] = temp;
     }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting.)

8 6 10 3 1 2 5 4	*pass 0*
1 6 10 3 8 2 5 4	*pass 1*
1 2 10 3 8 6 5 4	*pass 2*
1 2 3 10 8 6 5 4	*pass 3*
1 2 3 4 8 6 5 10	*pass 4*
1 2 3 4 5 6 8 10	*pass 5*
1 2 3 4 5 6 8 10	*pass 6*
1 2 3 4 5 6 8 10	*pass 7*

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).

D. The **Insertion Sort** algorithm is a commonly used algorithm. Even if you haven't been a programmer or a student of computer science, you may have used this algorithm. Try recalling how you sort a deck of cards. You start from the beginning, traverse through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following are two implementations in C.

```
void insertionSort(int a[], int array_size)
{
     int i, j, temp;
     for (i = 1; i < array_size; ++i)
     {
          temp = a[i];
          for (j = i; j > 0 && a[j-1] > temp; j--)
               a[j] = a[j-1];

          a[j] = temp;
     }
}
```

```
void insertionSort(int a[], int array_size)
{
     int i, j, temp;
     for (i = 1; i < array_size; ++i)
     {
            if (a[i] < a[i - 1] {
                  temp = a[i];
                  j = i;

                  while (temp < a[j-1]) {
                        a[j] = a[j - 1];
                        j--;
                  }  // end while

                  a[j] = temp;

            }  // end if
      }  // end for
}
```

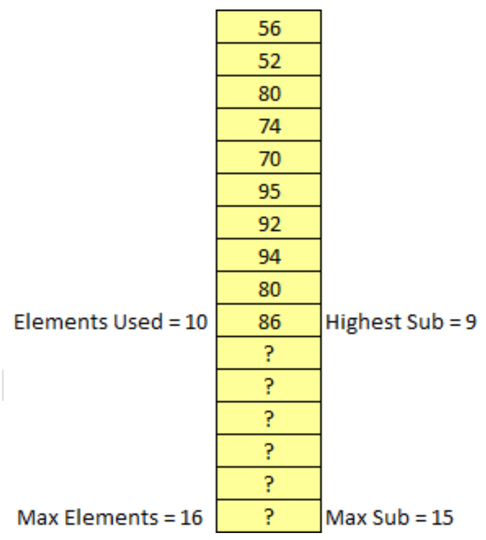
Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting.)

8 6 10 3 1 2 5 4	*pass 0*
6 8 10 3 1 2 5 4	*pass 1*
6 8 10 3 1 2 5 4	*pass 2*
3 6 8 10 1 2 5 4	*pass 3*
1 3 6 8 10 2 5 4	*pass 4*
1 2 3 6 8 10 5 4	*pass 5*
1 2 3 5 6 8 10 4	*pass 6*
1 2 3 4 5 6 8 10	*pass 7*

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. Now try out the deck-of-cards-sorting algorithm with this list and see if it matches with the tabulated data. For example, you start from 8 and the next card you see is 6. Hence you remove 6 from its current position and "insert" it back to the top. That constituted pass 1. Repeat the same process and you'll do the same thing for 3, which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its < (6,8,10) but > 1. As you carry on till you reach the end of the list you'll find that the list has been sorted.

6.7 Other Array Subtasks

A. Partially-filled Arrays



It is very common that you will not place values in all of the array elements but will instead partially fill the array. You must declare the array, so you want to pick some maximum size that would most commonly hold all the values that you wish to hold. If you make the array way too large, you will be wasting memory. If you make it too small, there will be cases where it will not hold all of your values. Given an array of a declared size, you know the number of elements and the highest subscript. If you partially fill the array, you must also keep track of the elements that you have filled, the count and/or the highest subscript.

B. `sizeof` and Arrays

```
const int MAX_SCORES = 20;

// ...

int scores[MAX_SCORES];
int scoresBytes = sizeof(scores);  // MAX_SCORES * 4
int scoresElements = sizeof(scores) / sizeof(int);  // MAX_SCORES
```

You should note that the sizeof operator returns the total amount of memory (in Bytes) allocated for its argument. Passing an array to the sizeof operator will result in the total Bytes in the array. You can determine the number of elements by dividing the total size by the size of each element. To determine the size of one element, you can pass sizeof one subscripted variable or you can pass it the array data type.

You cannot use sizeof to determine the number of elements that you are currently using in a partially filled array, only the total elements.

C. Loading an Array

Be careful not to overflow the array; do not read directly into array elements.

```
// load an array of scores, checking for overfill

const int MAX_SCORES = 20;
int scores[MAX_SCORES];
int score, scoreCount;

// load into array, check for too many
for (scoreCount = 0; scanf("%d", &score) == 1; scoreCount++) {
    // scoreCount here is one less than actual scores read
    if (scoreCount >= MAX_SCORES) {
        printf("Unable to store more than %d scores.\n", MAX_SCORES);
        exit(1);
    }
    scores[scoreCount] = score;
}
```

Let's say you want to read some test scores into an array. We decide that 20 elements should be large enough; however, we want to be careful not to overfill or overflow the array, addressing memory outside the bounds of the array. We declare the array using a constant, a variable to read into, and a variable to keep count of the number of items filled in the array.

The loop is going to start the count out at zero and increment it each time through the loop. We want to execute the loop as long as we have not reached end of file, so as long as our `scanf` returns a 1. As we enter the loop, the count is one less than the number of scores that we have successfully read, but this is what we need to use as a subscript for the next element to fill.

Before placing the score in the array, we check to see if the subscript is OK. If the subscript is too high we need to stop loading; so in this case we print an error message and use the `exit( )` function from `stdlib`. If the subscript is OK, we place the value read into the array. At the bottom of the loop, the `for` statement increments the count so count now has the number of values placed into the array.

D. Loading a Two-dimensional Array

```
int table[ROWS][COLS];
int row, col, value;

for (row = 0; row < ROWS; row++) {
    for (col = 0; col < COLS; col++) {
        scanf("%d", &value);
        table[row][col] = value;
    }
}
```

If you know the number of data items matches the size of an array, you can just use nested `for` loops to read each value and place it into the array.

E. Safer 2-D Load

```
int table[ROWS][COLS];
int row, col, value, match = 1;

scanf("%d", &value);

for (row = 0; !feof(stdin) && row < ROWS; row++) {
    for (col = 0; !feof(stdin) && col < COLS; col++) {
        table[row][col] = value;
        scanf("%d", &value);
    }

    if (!feof(stdin) || row != ROWS)
        match = 0;
}

// if after the for loop, match == 0, then that means
//     either too much data was in the file (!feof(stdin)) i.e. didn't reach eof
//     or not enough data in the file (row != ROWS)
```

F. Searching an Array

There are many reasons why you may wish to search through an array. You may wish to find if a target value exists in the array. You might wish to find where a value exists in an array.

With any array, you can do a ***linear search***. With this method, you go from element to element looking for a target value or values.

With a large sorted array, it is quicker to do a ***binary search*** where the data is continually divided into two sections and you use the value between the two to determine which half to search next.

The following code is an example of a ***linear search***:

```
int scores[MAX_SCORES];
int scoreCount, scoreNdx, targetScore;

// assume array has been loaded, count = scoreCount, and
// targetScore contains the search value

for (scoreNdx = 0;
     scoreNdx < scoreCount && scores[scoreNdx] != targetScore;
     scoreNdx++)   {
    ;  // empty loop body, by design
}

if (scoreNdx >= scoreCount) {
    // whatever you want to do if not found
}
else {
    // whatever you want to do if found
}
```

Notice that the body of the above `for` loop has only a semi-colon.   Why do you think that is?

G. Sum & Average Example

There are many types of statistics than can be gathered on the values in an array. Two of the most common are the sum and the average. The average is calculated by dividing the sum by the number of elements making up the sum. Before calculating an average, we should verify that the number of elements that we are going to divide by is positive so that we don't divide by zero.

In the example shown below, the count is checked first to see if we can compute the average. We don't even need to compute the sum if the count is not positive. If the count is positive, we initialize the sum, loop through the elements adding to the sum, and then calculate the average.

```
// calculate average score
int scores[MAX_SCORES];
int scoreCount, scoreNdx, sum;
float average;

// assume array has been loaded, count = scoreCount
if (scoreCount <= 0)
    printf("Unable to compute average, no scores.\n");
else {
    sum = 0;
    for (scoreNdx = 0; scoreNdx < scoreCount; scoreNdx++)
        sum += scores[scoreNdx];
    average = (float)sum / scoreCount;
    printf("Average score is %.2f\n", average);
}
```

Suppose we wanted an average of only the passing scores. You would then need to count and sum inside the loop and check the count after the loop to make sure that it was valid.

H. Extremes

The best method for determining extremes within an array is to assume that the first value is the current extreme. We then compare all the other values to the current extreme, replacing the current extreme if we find a more extreme value.

```
int scores[MAX_SCORES];
int scoreCount, scoreNdx, maxScore;
float average;

// assume array has been loaded, count = scoreCount
maxScore = score[0];    // assume first
for (scoreNdx = 1; scoreNdx < MAX_SCORES; scoreNdx++)
    if (scores[scoreNdx] > maxScore)    // check others
        maxScore = scores[scoreNdx];

printf("The highest score is %d\n", maxScore);
```