Introducing arrays

As programs become more complex, we notice that they require more variables, and thus more variable names to hold all necessary values. We could define:

```c
int x1, x2, x3, x4, x5.....;
```

but referring to them in our programs will quickly become unwieldy, and their actual names may be trying to tell us something.

In particular, our variables are often related to one another - they hold data having a physical significance, and the data value held in one variable is related to the data in another variable.

For example, consider a 2-dimensional field, where each square metre of the field may be identified by its rows and column coordinates. We may record each square's altitude, or temperature, or its number of ants:

```
(0,0) (1,0) (2,0) (3,0)
(0,1) (1,1) (2,1) (3,1)
(0,2) (1,2) (2,2) (3,2)
```

Like most languages, C provides a simple data structure, termed an *array*, to store and access data where the data items themselves are closely related.

Depending on the context, different problem domains will describe different kinds of arrays with different names:

- 1-dimensional arrays are often termed *vectors*.
- 2-dimensional arrays are often termed *matrices* (as in our example, above).
- 3-dimensional arrays are often termed *volumes*, and so on.

We'll start with the simple 1-dimensional arrays.
1-dimensional arrays

C provides support for 1-dimensional arrays by allowing us to identify the required data using a single index into the array.

Syntactically, we use square-brackets to identify that the variable is an array, and use an integer expression inside the array to identify which "part" of it we're requiring.

In all cases, an array is only a single variable, with one or more elements.

Consider the following code:

```c
#define N 20
int myarray[ N ];
int evensum;
evensum = 0;
for(int i=0 ; i < N ; i=i+1)
  {  
    myarray[ i ] = i * 2;
    evensum = evensum + myarray[ i ];
  }
```

What do we learn from this example?

- We declare our 1-dimensional arrays with square brackets, and indicate the maximum number of elements within those brackets.
- A fixed, known value (here N, with the value 20) is used to specify the number of elements of the array.
- We access elements of the array by providing the array's name, and an integer index into the array.
- Elements of an array may be used in the same contexts as basic (scalar) variables. Here myarray is used on both the left-hand and right-hand sides of assignment statements.

We may also pass array elements as arguments to functions, and return their values from functions.

- Array indices start "counting" from 0 (not from 1).
- Because our array consists of N integers, and indices begin at zero, the highest valid index is actually N-1.
Initializing 1-dimensional arrays

Like all variables, arrays should be initialized before we try to access their elements. We can:

- initialize the elements at run-time, by executing statements to assign values to the elements:

```
#define N 5
int myarray[ N ];
....
for(int i=0 ; i < N ; i=i+1)
{    
    myarray[ i ] = i;
}
```

- we may initialize the values at compile-time, by telling the compiler what values to initially store in the memory represented by the array. We use curly-brackets (braces) to provide the initial values:

```
#define N 5
int myarray[ N ] = { 0, 1, 2, 3, 4 };
```

- we may initialize the values at compile-time, by telling the compiler what values to initially store in the memory represented by the array, and having the compiler determine the number of elements in the array(!):

```
int myarray[ ] = { 0, 1, 2, 3, 4 }; #define N (sizeof(myarray) / sizeof(myarray[0]))
```

- or, we may initialize just the first few values at compile-time, and have the compiler initialize the rest with zeroes:

```
#define HUGE 10000
int myarray[ HUGE ] = { 4, 5 };
```
Strings are 1-dimensional arrays of characters

In contrast to some other programming languages, C does not have a basic datatype for strings.

However, C compilers provide some basic support for strings by considering strings to simply be arrays of characters.

We've already seen this support when calling the `printf()` function:

```c
printf("I'm afraid I can't do that Dave\n");
```

The double quotation characters simply envelope the characters to be treated as a sequence of characters.

In addition, a standards-conforming C compiler is required to also provide a large number of string-handling functions in its standard C library. Examples include:

```c
#include <string.h>

// which declares many functions, including:

int strlen( char string[] );          // to determine the length of a string
int strcmp( char str1[], char str2[] );  // to determine if two strings are equal
char *strcpy( char destination[], char source[] );  // to make a copy of a string
```

In reality these functions are not "really" managing strings as a basic datatype, but are just managing arrays of characters.
Initializing character arrays

As we've just seen with 1-dimensional arrays of integers, C also provides facility to initialize character arrays.

All of the following examples are valid:

```c
char greeting[5] = { 'h', 'e', 'l', 'l', 'o' };
char today[7] = "Tuesday";
char month[] = "August";
```

The 3rd of these is the most interesting. We have not specified the size of the array `month` ourselves, but have permitted the compiler to count and allocate the required size.
Strings are terminated by a special character

Unlike other arrays in C, the support for character arrays is extended by treating one character, the null byte, as having special significance. We may specify the null byte, as in the example:

```c
array[3] = '\0';
```

The null byte is used to indicate the end of a character sequence, and it exists at the end of all strings that are defined within double-quotes.

Inside the computer's memory we have:

```
  h e l l o  \0
```

- the string requires 6 bytes of memory to be stored correctly, but
- functions such as strlen(), which calculate the string's length, will report it as 5.

There is no inconsistency here - just something to watch out for.

Because the null byte has special significance, and because we may think of strings and character arrays as the same thing, we can manipulate the contents of strings by changing the array elements. Consider:

```
  h e l l o  w o r l d  \0
```

If we execute the statement:

```c
array[5] = '\0';
```

the space between the two words is replaced by the null byte.

The result is that the array still occupies 12 bytes of storage, but if we tried to print it out, we would only get hello.
Copying strings

As strings are so important, the standard C library provides many functions to examine and manipulate strings. However, C provides no basic string datatype, so we often need to treat strings as array of characters.

Consider these implementations of functions to copy one string into another:

```c
// DETERMINE THE STRING LENGTH, THEN USE A BOUNDED LOOP
void my_strcpy(char destination[], char source[]) {  
    int length = strlen(source);  
    for(int i = 0 ; i < length ; i = i+1)  
    {  
        destination[i] = source[i];  
    }  
    destination[length] = '\0';
}

// DO NOT WRITE STRING-PROCESSING LOOPS THIS WAY
void my_strcpy(char destination[], char source[]) {  
    int i;  
    for(i = 0 ; i < strlen(source) ; i = i+1)  
    {  
        destination[i] = source[i];  
    }  
    destination[i] = '\0';
}

// USE AN UNBOUNDED LOOP, COPYING UNTIL THE NULL-BYTE
void my_strcpy(char destination[], char source[]) {  
    int i = 0;  
    while(source[i] != '\0')  
    {  
        destination[i] = source[i];  
        i = i+1;  
    }  
    destination[i] = '\0';
}

// USE AN UNBOUNDED LOOP, COPYING UNTIL THE NULL-BYTE
void my_strcpy(char destination[], char source[]) {  
    int i = 0;  
    do  
    {  
        destination[i] = source[i];  
        i = i+1;  
    }  
    while(source[i-1] != '\0');
}
```
Formatting our results into character arrays

There are many occasions when we wish our “output” to be written to a character array, rather than to the screen.

Fortunately, we need to learn very little - we now call standard function `sprintf`, rather than `printf`, to perform our formatting.

```c
#include <stdio.h>
char chess_outcome[64];
if(winner == WHITE) { 
sprintf(chess_outcome, "WHITE with %i", nwhite_pieces);
} else { 
sprintf(chess_outcome, "BLACK with %i", nblack_pieces);
} printf("The winner: %s\n", chess_outcome);
```

We must be careful, now, not to exceed the maximum length of the array receiving the formatted printing. Thus, we prefer functions which ensure that not too many characters are copied:

```c
char chess_outcome[64];
// FORMAT, AT MOST, A KNOWN NUMBER OF CHARACTERS
if(winner == WHITE) { 
  snprintf(chess_outcome, 64, "WHITE with %i", nwhite_pieces);
} // OR, GREATLY PREFERRED:
if(winner == WHITE) { 
  snprintf(chess_outcome, sizeof(chess_outcome), "WHITE with %i", nwhite_pieces);
}
```

---

Pre- and post-, increment and decrement
To date, we've always employed the "traditional" mechanism of incrementing integer values, in both assignment statements and in for loops:

```c
int value = 0;
....
value = value + 1;
....
for(int i=0 ; i < MAXVALUE ; i=i+1)
{    ....
}
```

C provides a shorthand notation for incrementing and decrementing scalar variables, by one:

```c
int value = 0;
char ch = 'z';
++value;   // value is now 1
--ch;  // ch is now 'y'
....
for(int i=0 ; i < MAXVALUE ; ++i)
{    ....
}
for(char letter='a' ; letter <= 'z' ; ++letter)
{    ....
}
```

The notation used above is always used to increment or decrement by one, and the 2 statements:

```c
++value ;   // pre-increment value
value++ ;   // post-increment value
```

produce the exact same result.
Pre- and post-, increment and decrement, continued

While pre- and post-incrementing (and decrementing) initially appears simple, we must be careful when using modified variables in expressions.

Consider these results (all statements executed top-to-bottom):

```
int x = 0;
int y = 0;

// ------------------ what --- X --- Y ---
what = ++x;  // 1 1 0
what = y++;  // 0 1 1
what = y++;  // 1 1 2
what = ++y;  // 3 1 3
```

Shorthand arithmetic

A similar notation may be used to perform any standard arithmetic operation on a variable. For example, assuming the correct declarations:

```
value += 2;     // equivalent to  value = value + 2;
value -= y;     // equivalent to  value = value - y;
total *= x;      // equivalent to  total = total * x;
half /= 2;       // equivalent to  half = half / 2;
poly += x*y;     // equivalent to  poly = poly + (x*y);
```