## 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:

```
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:

```
#define  N   20

int   myarray[ N ];
int   evensum;

evensum = 0;
for(int i=0 ; i < N ; ++i) {
    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 **int**eger 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 indicies start "counting" from 0 (not from 1).
- **Because our array consists of *N* integers, and indicies 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) {
        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 };
```

## Variable-length arrays

All of the examples of 1-dimensional arrays we've seen have the array's size defined *at compile time*:

```
#define  N   5

int   global_array[ N ];
....

    for(int i=0 ; i < N ; ++i) {
        int   array_in_block[ 100 ];
        ....
    }
```

As the compiler knows the exact amount of memory required, it may generate more efficient code (in both space and time) and more secure code.

More generally, an array's size may not be known *until run-time*. These arrays are termed *variable-length arrays*, or *variable-sized arrays*. However, once defined, their size cannot be changed.

In all cases, variable-length arrays may be defined in a function and passed to another. However, because the size is not known until run-time, the array's size must be passed as well. It is **not possible** to determine an array's size from its name.

```
void function2(int array_size, char vla[ ])
{
    for(int i=0 ; i < array_size ; ++i) {
        // access vla[i] ...
        ....
    }
}

void function1(void)
{
    int size = read an integer from keyboard or a file;

    char vla[ size ];

    function2(size, vla);
}
```

Variable-length arrays were first defined in the C99 standard, but then made optional in C11 - primarily because of their inefficient implementation on embedded devices. Modern Linux operating system kernels are now free of variable-length arrays.

## 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:

```
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:

```
#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:

```
char greeting[5] = { 'h', 'e', 'l', 'l', 'o' };

char today[6]    = "Monday";

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:

```
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 |

Of note, when dealing with strings:

- 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:

```
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:

```
// 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) {
        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) {
        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.

```
#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:

```
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);
}
```

## Identifying related data

Let's consider the 2012 1st project for CITS1002.

The goal of the project was to manage the statistics of AFL teams throughout the season, calculating their positions on the premiership ladder at the end of each week.

Let's consider the significant global variables in its sample solution:

```
//  DEFINE THE LIMITS ON PROGRAM'S DATA-STRUCTURES
#define MAX_TEAMS                24
#define MAX_TEAMNAME_LEN         30
....

//  DEFINE A 2-DIMENSIONAL ARRAY HOLDING OUR UNIQUE TEAMNAMES
char    teamname[MAX_TEAMS][MAX_TEAMNAME_LEN+1];        // +1 for null-byte

//  STATISTICS FOR EACH TEAM, INDEXED BY EACH TEAM'S 'TEAM NUMBER'
int     played  [MAX_TEAMS];
int     won     [MAX_TEAMS];
int     lost    [MAX_TEAMS];
int     drawn   [MAX_TEAMS];
int     bfor    [MAX_TEAMS];
int     bagainst[MAX_TEAMS];
int     points  [MAX_TEAMS];
....

//  PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
    for(int t=0 ; t<nteams ; ++t) {
        printf("%s %i %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
                teamname[t],
                played[t], won[t], lost[t], drawn[t],
                bfor[t], bagainst[t],
                (100.0 * bfor[t] / bagainst[t]),      // calculate percentage
                points[t]);
    }
```

It's clear that the variables are all *strongly related*, but that we're naming and accessing them as if they are independent.

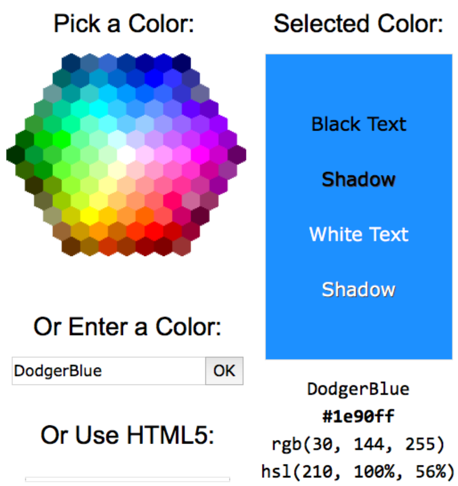
## Defining structures

Instead of storing and identifying related data as independent variables, we prefer to "collect" it all into a single structure.

C provides a mechanism to bring related data together, **structures**, using the **struct** keyword.

We can now define and gather together our related data with:

```
//  DEFINE AND INITIALIZE ONE VARIABLE THAT IS A STRUCTURE
struct {
    char     *name;    // a pointer to a sequence of characters
    int      red;      // in the range 0..255
    int      green;
    int      blue;
} rgb_colour = {
    "DodgerBlue",
     30,
    144,
    255
};
```




We now have a *single* variable (named *rgb_colour*) that is a **structure**, and at its point of definition we have initialised each of its 4 fields.

## Defining an array of structures

Returning to our AFL project example, we can now define and gather together its related data with:

```
//  DEFINE THE LIMITS ON PROGRAM'S DATA-STRUCTURES
#define MAX_TEAMS                24
#define MAX_TEAMNAME_LEN         30
....

struct {
    char     teamname[MAX_TEAMNAME_LEN+1];        // +1 for null-byte

//  STATISTICS FOR THIS TEAM, INDEXED BY EACH TEAM'S 'TEAM NUMBER'
    int      played;
    int      won;
    int      lost;
    int      drawn;
    int      bfor;
    int      bagainst;
    int      points;
} team[MAX_TEAMS];    //  DEFINE A 1-DIMENSIONAL ARRAY NAMED team
```

We now have a *single* (1-dimensional) array, *each element of which* is a **structure**.
We often term this *an array of structures*.

Each element of the array has a number of *fields*, such as its teamname (a whole array of characters) and an integer number of points.

## Accessing the fields of a structure

Now, when referring to individual items of data, we need to first specify which *team* we're interested in, and then which *field* of that team's structure.

We use a single dot ('.' or fullstop) to separate the variable name from the field name.

The old way, with independent variables:

```
//  PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<nteams ; ++t) {
    printf("%s %i %i %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
            teamname[t],
            played[t], won[t], lost[t], drawn[t],
            bfor[t], bagainst[t],
            (100.0 * bfor[t] / bagainst[t]),      // calculate percentage
            points[t]);
}
```

The new way, accessing fields within each structure:

```
//  PRINT EACH TEAM'S RESULTS, ONE-PER-LINE, IN NO SPECIFIC ORDER
for(int t=0 ; t<nteams ; ++t) {
    printf("%s %i %i %i %i %i %i %.2f %i\n", // %age to 2 decimal-places
            team[t].teamname,
            team[t].played, team[t].won, team[t].lost, team[t].drawn,
            team[t].bfor, team[t].bagainst,
            (100.0 * team[t].bfor / team[t].bagainst),      // calculate percentage
            team[t].points);
}
```

While it requires more typing(!), it's clear that the fields all belong to the same structure (and thus team). Moreover, the names *teamname*, *played*, .... may now be used as "other" variables elsewhere.

## Accessing system information using structures

Operating systems (naturally) maintain a lot of (related) information, and keep that information in structures.

So that the information about the structures (the datatypes and names of the structure's fields) can be known by both the operating system and users' programs, these structures are defined in *system-wide header files* - typically in `/usr/include` and `/usr/include/sys`.

For example, consider how an operating system may represent *time* on a computer:

```
#include <stdio.h>
#include <sys/time.h>

// A value accurate to the nearest microsecond but also has a range of years
struct timeval {
    int   tv_sec;        // Seconds
    int   tv_usec;       // Microseconds
};
```

Note that the structure has now been given a *name*, and we can now define multiple variables having this named datatype (in our previous example, the structure would be described as *anonymous*).

We can now request information from the operating system, with the information returned to us in structures:

```
#include <stdio.h>
#include <sys/time.h>

    struct timeval  start_time;
    struct timeval  stop_time;

    gettimeofday( &start_time, NULL );
    printf("program started at %i.06%i\n",
                    (int)start_time.tv_sec,  (int)start_time.tv_usec);

    ....
    perform_work();
    ....

    gettimeofday( &stop_time, NULL );
    printf("program stopped at %i.06%i\n",
                    (int)stop_time.tv_sec,  (int)stop_time.tv_usec);
```

Here we are *passing the structure by address*, with the **&** operator, so that the *gettimeofday()* function can modify the fields of our structure.

(we're not passing a meaningful pointer as the second parameter to gettimeofday(), as we're not interested in timezone information)