1D Arrays

Arrays in C can be allocated on the stack - or they may be allocated on the heap

Or they may be declared as global variables

Arrays in C are indexed from 0, and use square brackets

C arrays don’t record their sizes

When you pass a C array into a function, the size information is lost, so **you pass the number of elements separately**

Arrays are declared with types and (constant) sizes:

```
double data[30];
```

It’s often a good idea, especially in larger programs, to replace such magic constants as 30 with named constants, e.g.:

```
#define MAXDATA 30
double data[MAXDATA];
```

It is possible (and excellent practice) to initialise an array during its declaration.

```
int a[3] = {1, 2, 3}; // All elements specified
int b[10] = {1, 2, 3, 4}; // Some elements unspecified
```

Like all other variables in C, the value of each array element is undefined, unless you initialise it either at declaration time (as above) or by an explicit for loop

An Example

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

#define MAXVALUES 200
```

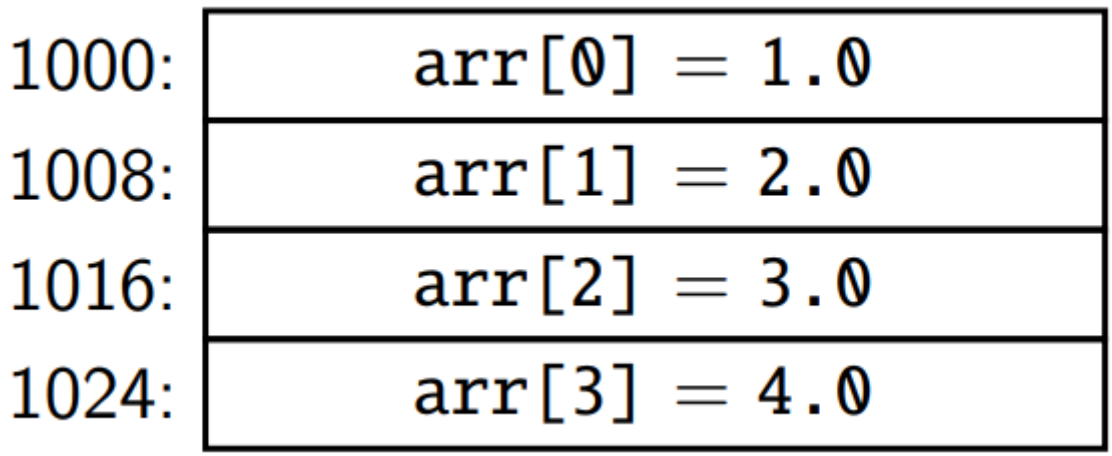
```
#define PI 3.1415926

int main( void ) {
        double data[MAXVALUES];
        for( int i=0; i<MAXVALUES; i++ ) {
                data[i] = 100 * sin(i*PI/100);
        }
        // now use data[] normally
        for( int i=0; i<MAXVALUES; i++ ) {
                printf( "%d %.4lf\n", i, data[i] );
        }
        return 0;
}
```

`%.4lf` = Round the doubles to 4dp

A simple array would be laid out in memory as follows (assuming that it starts at address 1000, and `sizeof (double) == 8`):

```
double arr[4] = { 1.0, 2.0, 3.0, 4.0 };
```



A local array is allocated on the stack. Various architectures have size limits - either on an individual stack frame, or on the whole stack. So a tremendously large local array may cause problems. To avoid this, you might consider making such arrays global, or heap allocated

You may omit the array size when it can be inferred from the initializer:

```
int arr[] = {10,20,30,40,50,60};
```

To write code that iterates over all elements of such an array, you need to know the number of elements (calculated by the compiler for you)

The number of elements is `sizeof(arr) / sizeof(arr[0])`

Personally I'd put this in a macro:

```
#define NELEMENTS(arr) (sizeof(arr) / sizeof(arr[0]))
```

To iterate over every element of an array, summing them up, we'd write:

```
int arr[] = {10,20,30,40,50,60};
int nel = NELEMENTS(arr);
int total = 0;
for( int i = 0; i < nel; i++ ) // for each element in arr:
{
        total += arr[i];
}
printf( "total = %d\n", total );
```

This only works when the full declaration of the array is available. It doesn't work when the array is passed as a parameter to a function - inside any function `sizeof(arr)` delivers the size of a pointer, not the array

Hence, when passing an array to a function, you **pass a second parameter - the number of elements in the array**

C99 introduced variable length stack-allocated arrays:

```
#include <stdio.h>
int main( void ) {
        int n = -1;
        printf( "Enter n: " ); scanf( "%d", &n );
        double arr[n];
        //...
        return 0;
}
```

Given a suitably large value of n, or a negative value, this program will **crash**

Note that the array size still needs to be maintained separately by the programmer (and passed separately into functions)

Don't use variable length arrays, instead, heap allocate a block of memory to use as a dynamic array