Chapter 10: Arrays

Arrays are derived data types, representing an ordered collection of values ("elements") of another type. Most arrays in C have a fixed number of elements of any one type, and its representation stores the elements contiguously in memory without gaps or padding. C allows multidimensional arrays whose elements are other arrays, and also arrays of pointers.

C supports dynamically allocated arrays whose size is determined at run time. C99 and later supports variable length arrays or VLAs.

Section 10.1: Declaring and initializing an array

The general syntax for declaring a one-dimensional array is

```
type arrName[size];
```

where `type` could be any built-in type or user-defined types such as structures, `arrName` is a user-defined identifier, and `size` is an integer constant.

Declaring an array (an array of 10 int variables in this case) is done like this:

```
int array[10];
```

it now holds indeterminate values. To ensure it holds zero values while declaring, you can do this:

```
int array[10] = {0};
```

Arrays can also have initializers, this example declares an array of 10 `int`'s, where the first 3 `int`'s will contain the values `1`, `2`, `3`, all other values will be zero:

```
int array[10] = {1, 2, 3};
```

In the above method of initialization, the first value in the list will be assigned to the first member of the array, the second value will be assigned to the second member of the array and so on. If the list size is smaller than the array size, then as in the above example, the remaining members of the array will be initialized to zeros. With designated list initialization (ISO C99), explicit initialization of the array members is possible. For example,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

In most cases, the compiler can deduce the length of the array for you, this can be achieved by leaving the square brackets empty:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Declaring an array of zero length is not allowed.

`Version ≥ C99` `Version < C11`

Variable Length Arrays (VLA for short) were added in C99, and made optional in C11. They are equal to normal arrays, with one, important, difference: The length doesn't have to be known at compile time. VLA's have automatic storage duration. Only pointers to VLA's can have static storage duration.

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Important:

VLA's are potentially dangerous. If the array `vla` in the example above requires more space on the stack than available, the stack will overflow. Usage of VLA's is therefore often discouraged in style guides and by books and exercises.

Section 10.2: Iterating through an array efficiently and row-major order

Arrays in C can be seen as a contiguous chunk of memory. More precisely, the last dimension of the array is the contiguous part. We call this the *row-major order*. Understanding this and the fact that a cache fault loads a complete cache line into the cache when accessing uncached data to prevent subsequent cache faults, we can see why accessing an array of dimension 10000x10000 with `array[0][0]` would **potentially** load `array[0][1]` in cache, but accessing `array[1][0]` right after would generate a second cache fault, since it is `sizeof(type)*10000` bytes away from `array[0][0]`, and therefore certainly not on the same cache line. Which is why iterating like this is inefficient:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

And iterating like this is more efficient:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

In the same vein, this is why when dealing with an array with one dimension and multiple indexes (let's say 2 dimensions here for simplicity with indexes i and j) it is important to iterate through the array like this:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
```

```
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

Or with 3 dimensions and indexes i,j and k:

```
#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

Or in a more generic way, when we have an array with **N1 x N2 x ... x Nd** elements, **d** dimensions and indices noted as **n1,n2,...,nd** the offset is calculated like this

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1) \cdots)) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Picture/formula taken from: https://en.wikipedia.org/wiki/Row-major_order

Section 10.3: Array length

Arrays have fixed lengths that are known within the scope of their declarations. Nevertheless, it is possible and sometimes convenient to calculate array lengths. In particular, this can make code more flexible when the array length is determined automatically from an initializer:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

However, in most contexts where an array appears in an expression, it is automatically converted to ("decays to") a pointer to its first element. The case where an array is the operand of the `sizeof` operator is one of a small number of exceptions. The resulting pointer is not itself an array, and it does not carry any information about the length of the array from which it was derived. Therefore, if that length is needed in conjunction with the pointer, such as when the pointer is passed to a function, then it must be conveyed separately.

For example, suppose we want to write a function to return the last element of an array of `int`. Continuing from the

above, we might call it like so:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

The function could be implemented like this:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Note in particular that although the declaration of parameter `input` resembles that of an array, **it in fact declares `input` as a pointer** (to `int`). It is exactly equivalent to declaring `input` as `int *input`. The same would be true even if a dimension were given. This is possible because arrays cannot ever be actual arguments to functions (they decay to pointers when they appear in function call expressions), and it can be viewed as mnemonic.

It is a very common error to attempt to determine array size from a pointer, which cannot work. DO NOT DO THIS:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

In fact, that particular error is so common that some compilers recognize it and warn about it. `clang`, for instance, will emit the following warning:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                            ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Section 10.4: Passing multidimensional arrays to a function

Multidimensional arrays follow the same rules as single-dimensional arrays when passing them to a function. However the combination of decay-to-pointer, operator precedence, and the two different ways to declare a multidimensional array (array of arrays vs array of pointers) may make the declaration of such functions non-intuitive. The following example shows the correct ways to pass multidimensional arrays.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
```

```
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

See also

Passing in Arrays to Functions

Section 10.5: Multi-dimensional arrays

The C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration –

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional (5 x 10 x 4) integer array:

```
int arr[5][10][4];
```

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of dimensions m x n, we can write as follows:

```
type arrayName[m][n];
```

Where `type` can be any valid C data type (`int`, `float`, etc.) and `arrayName` can be any valid C identifier. A two-dimensional array can be visualized as a table with `m` rows and `n` columns. **Note**: The order *does* matter in C. The array `int a[4][3]` is not the same as the array `int a[3][4]`. The number of rows comes first as C is a *row*-major language.

A two-dimensional array `a`, which contains three rows and four columns can be shown as follows:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array `a` is identified by an element name of the form `a[i][j]`, where `a` is the name of the array, `i` represents which row, and `j` represents which column. Recall that rows and columns are zero indexed. This is very similar to mathematical notation for subscripting 2-D matrices.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The following define an array with 3 rows where each row has 4 columns.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

While the method of creating arrays with nested braces is optional, it is strongly encouraged as it is more readable and clearer.

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. Let us check the following program where we have used a nested loop to handle a two-dimensional array:

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
```

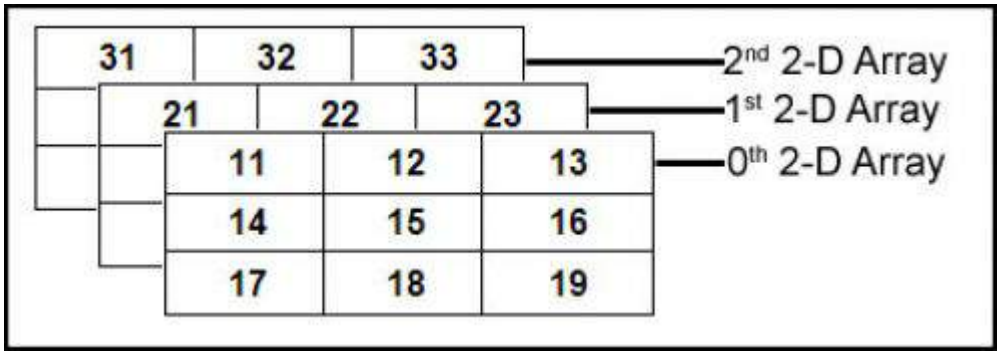
```
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

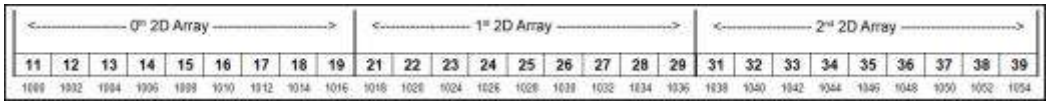
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Three-Dimensional array:

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D arrays.



3D array memory map:



Initializing a 3D Array:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

We can have arrays with any number of dimensions, although it is likely that most of the arrays that are created will be of one or two dimensions.

Section 10.6: Define array and access array element

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to
                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
   for ( i = 0; i < ARRLEN ; i++ )
   {
      n[ i ] = i + 100; /* Set element at location i to i + 100. */
   }

   /* Output each array element's value. */
   for (j = 0; j < ARRLEN ; j++ )
   {
      printf("Element[%zu] = %d\n", j, n[j] );
   }

   return 0;
}
```

Section 10.7: Clearing array contents (zeroing)

Sometimes it's necessary to set an array to zero, after the initialization has been done.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */

  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

An common short cut to the above loop is to use `memset()` from **`<string.h>`**. Passing `array` as shown below makes it decay to a pointer to its 1st element.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

or

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

As in this example `array` *is* an array and not just a pointer to an array's 1st element (see Array length on why this is important) a third option to 0-out the array is possible:

```
memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Section 10.8: Setting values in arrays

Accessing array values is generally done through square brackets:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

As a side effect of the operands to the `+` operator being exchangeable (--> commutative law) the following is equivalent:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

so as well the next statements are equivalent:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

and those two as well:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C doesn't perform any boundary checks, accessing contents outside of the declared array is undefined (Accessing memory beyond allocated chunk):

```
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Section 10.9: Allocate and zero-initialize an array with user defined size

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

```c
int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

This program tries to scan in an unsigned integer value from standard input, allocate a block of memory for an array of `n` elements of type `int` by calling the `calloc()` function. The memory is initialized to all zeros by the latter.

In case of success the memory is releases by the call to `free()`.

Section 10.10: Iterating through an array using pointers

```c
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Here, in the initialization of `p` in the first `for` loop condition, the array a *decays* to a pointer to its first element, as it would in almost all places where such an array variable is used.

Then, the `++p` performs pointer arithmetic on the pointer `p` and walks one by one through the elements of the

array, and refers to them by dereferencing them with `*p`.