Arrays

The `sizeof` Operator

C has a an operator that returns the size (in bytes) of a value or type. It's a unary operator and it is listed in the precedence chart.

General form is:

```
sizeof(expression or type)
```

Examples:

Code	Output

`int i;`
`float f;`
`double d;`
`printf("sizeof(i) is %u\n", sizeof(i));`
`printf("sizeof(int) is %u\n", sizeof(int));`
`printf("sizeof(42) is %u\n", sizeof(42));`
`printf("\n");`
 	
`printf("sizeof(f) is %u\n", sizeof(f));`
`printf("sizeof(float) is %u\n", sizeof(float));`
`printf("sizeof(42.0F) is %u\n", sizeof(42.0F));`
`printf("\n");`
 	
`printf("sizeof(d) is %u\n", sizeof(d));`
`printf("sizeof(double) is %u\n", sizeof(double));`
`printf("sizeof(42.0) is %u\n", sizeof(42.0));`
`printf("\n");`
	`sizeof(i) is 4`
	`sizeof(int) is 4`
	`sizeof(42) is 4`
	
	`sizeof(f) is 4`
	`sizeof(float) is 4`
	`sizeof(42.0F) is 4`
	
	`sizeof(d) is 8`
	`sizeof(double) is 8`
	`sizeof(42.0) is 8`

The **`sizeof`** operator is also unique in that it can determine the value at *compile time*. The program does not need to be executed to obtain the results. Also, the operand to the **`sizeof`** operator can be an expression. For example, given these declarations:

```
int i;
int j;
double d;
```

This is the result of applying the **`sizeof`** operator to these expressions:

```
sizeof(i + j) is 4
sizeof(i * j - 20) is 4
sizeof(i * j * d) is 8
sizeof(i + 2.0) is 8
sizeof(10 + 3.2F - 5.4) is 8
```

One-Dimensional Arrays

An array is an *aggregate* data structure. This means that it consists of multiple values, all of which are the same type. Contrast this to *scalar* data types like **`float`** and **`int`**, which are single values. Each value in an array is called an *element*.

Because all of the values in an array have the same type, an array is called a *homogeneous* data structure. To declare an array, you must specify an additional piece of information: the size.

The general form is:

```
type identifier[integer_constant];
```

This declares an array of 10 integers. The array is called *a* and has room for 10 elements:

```
int a[10];
```

Visually, we can think of the array in memory like this:

Also, like local scalar variables, the values of the array are undefined:

Examples:

```
int touchdowns[10];     /* array of 10 integers, 40 bytes on 32-bit machines */
float distances[20];    /* array of 20 floats, 80 bytes on 32-bit machines   */
double temperatures[2]; /* array of 2 doubles, 16 bytes on 32-bit machines   */
```

Visually:

The **`sizeof`** operator will also determine number of bytes required for an array. Examples:

Code

```
int touchdowns[10];
float distances[20];
double temperatures[2];

printf("sizeof(touchdowns) is %u\n", sizeof(touchdowns));
printf("sizeof(distances) is %u\n", sizeof(distances));
printf("sizeof(temperatures) is %u\n", sizeof(temperatures));
```

Output

```
sizeof(touchdowns) is 40
sizeof(distances) is 80
sizeof(temperatures) is 16
```

Accessing Array Elements

- The name for the array applies to the *entire* array; all of the elements.
- Each individual element is anonymous, in that it doesn't have a name.
- So how do we access these elements if they don't have a name?
- We access them as *offsets* into the array.
- The first element has an offset of 0. (The offset is also called an *index*.)
- Arrays in C are known as zero-based arrays (since they start at index 0).

Note that the addresses shown below are completely arbitrary and for discussion purposes only.

Most work with arrays is done with some kind of looping construct:

Assigning values to each element

```
int a[10];
int i;
for (i = 0; i < 10; i++)
  a[i] = i * 2;
```

Printing out the values

```
for (i = 0; i < 10; i++)
  printf("%i  ", a[i]);
```

```
Output: 0  2  4  6  8  10  12  14  16  18
```

It is crucial that you understand that there is absolutely, positively no boundary checking when reading/writing an array. Your program is completely undefined in the event you read/write out of bounds (even if it *appears* to work correctly).

Writing past the end of the array:

```
int a[10];
int i;
for (i = 0; i < 15; i++)
  a[i] = i * 2;
```

The output at runtime:

```
    21 [main] a 3672 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
   546 [main] a 3672 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
    21 [main] a 3672 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
   546 [main] a 3672 open_stackdumpfile: Dumping stack trace to a.exe.stackdump
 63187 [main] a 3672 _cygtls::handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
 68303 [main] a 3672 _cygtls::handle_exceptions: Error while dumping state (probably corrupted stack)
```

Note that *undefined* means just that. This similar code may or may not result in an infinite loop:

Code **Possible memory layout**

```
int i;
int a[10];

  /* initialize elements WRONG */
for (i = 0; i <= 10; i++)
  a[i] = 0;
```

Arrays and looping go hand-in-hand.

for

```
int i;
int a[10];

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

while

```
int i;
int a[10];

i = 0;
while (i < 10)
{
  a[i] = 0;
  i++;
}
```

while (compact)

```
int i;
int a[10];

i = 0;
while (i < 10)
  a[i++] = 0;
```

Unlike scalar types, you can't assign one array to another. You must "manually" copy each element from one array to the other:

```
void assign1(void)
{
  #define SIZE 10

  int a[SIZE]; /* 10 integers */
  int b[SIZE]; /* 10 integers */
  int i;

    /* set elements to i squared */
    /* 0, 1, 4, 9, 16, etc...    */
  for (i = 0; i < SIZE; i++)
    b[i] = i * i;

    /* Assign elements of b to a */
    /* This is not legal         */
  a = b;

    /* Assign elements of b to a    */
    /* This is how to assign arrays */
  for (i = 0; i < SIZE; i++)
    a[i] = b[i];
}
```

Initializing Arrays

When we declare an array and provide its size, the array is called a *static array*. The size of the array is set in stone and will never change. Ever.

We can perform static initialization of an array:

```
void some_function(void)
{
  int array1[5] = {1, 2, 3, 4, 5};  /* All elements are initialized */
  ...
}
```

In partial initialization, if we "run out" of initializers, the remaining elements are set to 0:

```
int array3[5] = {1, 2, 3};          /* 1, 2, 3, 0, 0 */
```

It is an error to provide too many initializers:

```
int array4[5] = {1, 2, 3, 4, 5, 6}; /* error: too many initializers */
```

C has a convenient feature that allows us to leave the size of the array empty. The compiler automatically fills in the size based on the number of initializers provided:

```
int array5[] = {1, 2, 3};          /* array5 has size 3 */
int array6[] = {1, 2, 3, 4, 5, 6}; /* array6 has size 6 */
```

A very convenient way to initialize all elements to 0:

```
int array3[50] = {0};            /* All 50 elements are set to 0 */
```

Example: Given a date in the form of **month/day**, print out the day of the year. For example:

```
Day of Year for 1/1 is 1
Day of Year for 2/1 is 32
Day of Year for 5/13 is 133
Day of Year for 12/31 is 365
```

```
void DayOfYear(void)
{
    /* The number of days in each month */
  int jan = 31;  int feb = 28;  int mar = 31;
  int apr = 30;  int may = 31;  int jun = 30;
  int jul = 31;  int aug = 31;  int sep = 30;
  int oct = 31;  int nov = 30;  int dec = 31;

  int month, day; /* Current month and day */
  int count = 0;  /* Total count of days   */

    /* Prompt the user for month/day */
  printf("Enter a date (mm/dd): ");
  scanf("%d/%d", &month, &day);

    /* Add up the days in previous months     */
    /* break statement intentionally missing  */
  switch (month)
  {
    case 12: count += nov;
    case 11: count += oct;
    case 10: count += sep;
    case  9: count += aug;
    case  8: count += jul;
    case  7: count += jun;
    case  6: count += may;
    case  5: count += apr;
    case  4: count += mar;
    case  3: count += feb;
    case  2: count += jan;

    default: count += day; /* Add in this month's days */
  }

    /* Format and print out the results */
  printf("The date %i/%i is day number %i\n", month, day, count);
}
```

The same problem using an array:

```
void DayOfYear(void)
{
    /* The number of days in each month */
  int months[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

  int month, day; /* Current month and day */
  int count = 0;  /* Total count of days   */
  int i;          /* Loop counter          */

    /* Prompt the user for month/day */
  printf("Enter a date (mm/dd): ");
  scanf("%d/%d", &month, &day);

    /* Add up the days in previous months */
  for (i = month - 2; i >= 0; i--)
    count += months[i];

    /* Add in this month's days */
  count += day;

    /* Format and print out the results */
  printf("The date %i/%i is day number %i\n", month, day, count);
}
```

Multidimensional Arrays

An array with more than one *dimension* is called a *multidimensional array*.

```
int matrix[5][10];  /* array of 5 arrays of 10 int; a 5x10 array of int */
```

Building up multidimensional arrays:

```
int a;            /* int                                  */
int b[10];        /* array of 10 int                      */
int c[5][10];     /* array of 5 arrays of 10 int          */
```

```
int d[3][5][10];  /* array of 3 arrays of 5 arrays of 10 int */

int e[10][5][3];  /* array of 10 arrays of 5 arrays of 3 int */
```

Storage order
Arrays in C are stored in *row major* order. This means that the rightmost subscript varies the most rapidly.

Given the declaration of *points*:

```
double points[3][4];
```

- An array of 3 arrays of 4 doubles
- A 3x4 array of doubles

We could diagram the arrays like this:

With details:

Or draw it contiguously (as it really is in memory):

Or horizontally:

Giving concrete values to the 2D array of doubles will help visualize the arrays. Note how the initialization syntax helps us visualize the "array of arrays" notion:

```
double points[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

or even formatted as a 3x4 matrix:

```
double points[3][4] = {
                        {1.0,  2.0,  3.0,  4.0},
                        {5.0,  6.0,  7.0,  8.0},
                        {9.0, 10.0, 11.0, 12.0}
                      };
```

Diagram:

Some expressions involving *points*:

```
Addresses                                Type
---------------------------------------------------------------
points        = 0012F5A4    An array of 3 arrays of 4 doubles
&points       = 0012F5A4    A pointer to an array of 3 arrays of 4 doubles
points[0]     = 0012F5A4    An array of 4 doubles
&points[0]    = 0012F5A4    A pointer to an array of 4 doubles
*points       = 0012F5A4    An array of 4 doubles
&points[0][0] = 0012F5A4    A pointer to a double

Contents
------------------------
**points      = 1.000000
*points[0]    = 1.000000
points[0][0]  = 1.000000

Sizes
-------------------------
sizeof(points)       = 96
sizeof(*points)      = 32
sizeof(**points)     =  8
sizeof(points[0])    = 32
sizeof(points[0][0]) =  8
```

Code to display above tables.

Accessing Elements in a 2-D Array

```
short matrix[3][8]; /* 24 shorts, 3x8 array */
```

matrix

```
  matrix[0]
*(matrix + 0)
   *matrix

  matrix[1]
*(matrix + 1)

  matrix[2]
*(matrix + 2)

     matrix[1][2]
*(*(matrix + 1) + 2)
```

Remember the rule:

```
array[i] == *(array + i)
```

where:

- *array* is an array of any type
- *i* is any integer expression

With multidimensional arrays, the rule becomes:

```
array[i][j] == *(*(array + i) + j)
array[i][j][k] == *(*(*(array + i) + j) + k)
etc...
```

Pointer arithmetic is used to locate each element. (Base address + Offset)

Given this declaration:

```
short matrix[3][8];
```

The value of **sizeof** varies with the argument:

```
Sizes
-------------------------
sizeof(matrix)       = 48   ; entire matrix
sizeof(matrix[0])    = 16   ; first row
sizeof(matrix[1])    = 16   ; second row
sizeof(matrix[0][0]) = 2    ; first short element
```

Passing 2D Arrays to Functions

Putting values in the matrix and printing it:

```
Fill3x8Matrix(matrix);  /* Put values in the matrix */
Print3x8Matrix(matrix); /* Print the matrix         */
```

Implementations:

```
void Fill3x8Matrix(short matrix[][8])
{
  int i, j;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 8; j++)
      matrix[i][j] = i * 8 + j + 1;
}

void Print3x8Matrix(short matrix[][8])
{
  int i, j;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 8; j++)
      printf("%i ", matrix[i][j]);
  printf("\n");
}
```

These functions could have specified the parameters this way (see the precedence chart):

```
void Fill3x8Matrix(short (*matrix)[8])
void Print3x8Matrix(short (*matrix)[8])
```

or

```
void Fill3x8Matrix(short matrix[3][8]);
void Print3x8Matrix(short matrix[3][8]);
```

Why are they not declared like this?:

```
void Fill3x8Matrix(short matrix[][]);
void Print3x8Matrix(short matrix[][]);
```

The compiler needs to know the size of each *element* in each dimension. It doesn't need to (and can't) know the number of elements in the first dimension. The size of each element in the first dimension is determined by the other dimensions and the *type* of the elements.

```
void Test(int a[], int b[][6], int c[][3][5])
{
  printf("a = %p, b = %p, c = %p\n", a, b, c);
  a++;
  b++;
  c++;
  printf("a = %p, b = %p, c = %p\n", a, b, c);
}

Output:
a = 0012FEE8, b = 0012FF38, c = 0012FEFC
a = 0012FEEC, b = 0012FF50, c = 0012FF38
```

In decimal:

```
Output:
a = 1244904, b = 1244984, c = 1244924
a = 1244908, b = 1245008, c = 1244984
```

The function **Test** is equivalent to this:

```
void Test(int *a, int (*b)[6], int (*c)[3][5])
```

Other methods for filling the matrix use explicit pointer arithmetic:

```
void Fill3x8Matrix(short matrix[][8])
{
  int i, j;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 8; j++)
      *(*(matrix + i) + j) = i * 8 + j + 1;
}

void Fill3x8Matrix(short matrix[][8])
{
  int i, j;
  for (i = 0; i < 3; i++)
  {
    short *pmat = *(matrix + i);
    for (j = 0; j < 8; j++)
      *pmat++ = i * 8 + j + 1;
  }
}
```

How does the compiler calculate the address (offset) for the element below?

```
matrix[1][2];
```

Using address offsets we get:

```
&matrix[1][2] ==> &*(*(matrix + 1) + 2) ==> *(matrix + 1) + 2
```

1. First dimension - Each element of matrix is an array of 8 shorts, so each element is 16 bytes.
2. Second dimension - Each element of each element of matrix is a short, so it's 2 bytes.

Given these declarations:

```
short matrix[3][8]
short array[10]
```

We can calculate the size of any portion:

```
Expression             Meaning                 Size (bytes)
-----------------------------------------------------------
array                Entire array                  20
array[]              Element in 1st dimension       2
matrix               Entire array                  48
matrix[]             Element in 1st dimension      16
matrix[][]           Element in 2nd dimension       2
```

Recap:

- The compiler needs to know the size of each of the *elements*, in each dimension.
- Since the size of each dimension relies on the fundamental type (int, double, etc.) of the array(s), there is an implicit size

specified.

- In a two-dimensional array, knowing the size of the second dimension (number of columns) and the data type of the array is sufficient to perform pointer arithmetic on the first dimension.
- This seemingly convoluted way of locating array elements is required since memory is laid out in one dimension by the compiler. The multiple dimension syntax (e.g. `[][]`) is just a convenience for the programmer.

Dynamically Allocated 2D Arrays

Recall the 2D points static array and how a dynamically allocated array would look:

```
double points[3][4];
```

```
double *pd = malloc(3 * 4 * sizeof(double));
```

Given a row and column:

```
int row = 1, column = 2;
double value;
```

- The static 2D array can be accessed using subscripts, but the dynamic "2D array" can only be indexed with a single subscript.

```
value = points[row][column]; /* OK      */
value = pd[row][column];     /* ILLEGAL */
```

- We (the programmers) have to do all of the arithmetic to locate an element using two subscripts:

```
value = pd[row * 4 + column];
```

- The compiler is still doing some of the work for us:

```
value = *(address-of-pd + (row * 4 + column) * sizeof(double));
```

- What does the number **4** in the above calculations represent?

If we want to use two subscripts on a dynamic 2D array, we have to set things up a little differently.

Using these definitions from above:

```
#define ROWS 3
#define COLS 4
double *pd = malloc(ROWS * COLS * sizeof(double));
```

Create a variable that is a *pointer* to a *pointer* to a **double**

```
double **ppd;
```

Allocate an array of 3 (ROWS) pointers to doubles and point *ppd* at it:

```
ppd = malloc(ROWS * sizeof(double *));
```

Point each element of *ppd* at an array of 4 doubles:

```
ppd[0] = pd;
ppd[1] = pd + 4;
ppd[2] = pd + 8;
```

Of course, for a large array, or an array whose size is not known at compile time, you would want to set these in a loop:

```
int row;
for (row = 0; row < ROWS; row++)
  ppd[row] = pd + (COLS * row);
```

This yields the diagram:

Given a row and column, we can access elements through the single pointer or double pointer variable:

```
int row = 1, column = 3;
double value;

  /* Access via double pointer using subscripting */
value = ppd[row][column];
```

```
  /* Access via single pointer using pointer arithmetic        */
  /* and/or subscripting. These statements are all equivalent. */
value = pd[row * COLS + column];
value = *(pd + row * COLS + column);
value = (pd + row * COLS)[column];
```