Array Outline

1. Array Outline
2. Scalars
3. Another Scalar Example
4. The Same Program, With Multiplication
5. The Same Program, With a Twist
6. Arrays/
   Array Element Properties
7. Array Properties
8. Array Indices
9. Array Declarations
10. Assigning a Value to an Array Element
11. Using `scanf` to Obtain a Value of an Array Element
12. Using `for` Loops to Accomplish Tasks on Arrays
13. Using `for` Loops to Accomplish Tasks on Arrays (continued)
14. Reading in an Array’s Values Using a `for` Loop
15. `for` Loop Behaves the Same as Several Statements
16. `for` Loop Behaves the Same as Several Statements (continued)
17. Reading in an Array’s Values on a Single Line of Input Text
18. Aside: Why We Like Declaring Named Constants
19. Declaring Named Constants for Loop Bounds
20. Computing with Arrays
21. Computing with Arrays (continued)
22. Static Memory Allocation
23. Sometime Static Memory Allocation Just Isn’t Good Enough
24. Static Memory Allocation Can Be Wasteful
25. Dynamic Memory Allocation
26. Dynamic Memory Deallocation
27. Dynamic Memory Allocation Example
28. Exercise: Mean
 Scalars

Consider this program:

```c
#include <stdio.h>

int main ()
{ /* main */
    int b, c, d, e, f;
    b = 0;
    c = 2;
    d = 4;
    e = 6;
    f = 8;
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("d = %d\n", d);
    printf("e = %d\n", e);
    printf("f = %d\n", f);
    return 0;
} /* main */
```

Notice that all of the variables in the program are simple `int` variables. Each of the individual `int` variables has a single name, a single address, a single data type and a single value.

Such variables, whether their type is `int`, `float` or `char`, are referred to as `scalar` variables.
Another Scalar Example

Here’s another version of the same program:

```
% cat scalar_a.c
#include <stdio.h>

int main ()
{ /* main */
    int a0, a1, a2, a3, a4;

    a0 = 0;
    a1 = 2;
    a2 = 4;
    a3 = 6;
    a4 = 8;
    printf("a0 = %d\n", a0);
    printf("a1 = %d\n", a1);
    printf("a2 = %d\n", a2);
    printf("a3 = %d\n", a3);
    printf("a4 = %d\n", a4);
    return 0;
} /* main */

% gcc -o scalar_a scalar_a.c
% scalar_a
a0 = 0
a1 = 2
a2 = 4
a3 = 6
a4 = 8
```

The only difference between this program and the previous program is the names of the scalar variables (and therefore some of the output).
The Same Program, With Multiplication

Here’s another version of the same program:

```c
#include <stdio.h>

int main ()
{
    int a0, a1, a2, a3, a4;

    a0 = 2 * 0;
    a1 = 2 * 1;
    a2 = 2 * 2;
    a3 = 2 * 3;
    a4 = 2 * 4;
    printf("a0 = %d\n", a0);
    printf("a1 = %d\n", a1);
    printf("a2 = %d\n", a2);
    printf("a3 = %d\n", a3);
    printf("a4 = %d\n", a4);
    return 0;
}
```

Notice that, in this program, the values of the scalar variables are obtained by multiplying a constant by the number associated with the scalar variable.
The Same Program, With a Twist

Here’s a very strange version of the program:

```
% cat array_mult.c
#include <stdio.h>

int main ()
{
    const int a_length = 5;
    int a[a_length];

    a[0] = 2 * 0;
    a[1] = 2 * 1;
    a[2] = 2 * 2;
    a[3] = 2 * 3;
    a[4] = 2 * 4;
    printf("a[0] = %d\n", a[0]);
    printf("a[1] = %d\n", a[1]);
    printf("a[2] = %d\n", a[2]);
    printf("a[3] = %d\n", a[3]);
    printf("a[4] = %d\n", a[4]);
    return 0;
}
```

% gcc -o array_mult array_mult.c
% array_mult

a[0] = 0
a[1] = 2
a[2] = 4
a[3] = 6
a[4] = 8

Huh?
Arrays

int a[8];

An array is a special kind of variable. Like a scalar variable, it has:

- a name;
- an address;
- a data type.

But instead of an array having exactly one single value, it can have multiple values. Each of these values is referred to as an element of the array.

If you’re familiar with vectors, you can think of an array as the equivalent idea, but in computing instead of in mathematics.

Array Element Properties

Each of the elements of an array is almost exactly like a scalar variable.

An element of an array has:

1. a name, part of which it shares with all of the other elements of the array that it belongs to;
2. an address, which we’ll learn about shortly;
3. a data type, which it shares with all of the other elements of the array that it belongs to;
4. a single value.

But, an array element also has:

5. an index, which we’ll learn about shortly.
Array Properties

\[
\text{int } \text{a[8]};
\]

An array as a whole has the following properties:

1. It has a data type, which is the data type of each of its elements; e.g., \text{int}

2. It as a \textit{dimension} attribute, sometimes called its \textit{length}, which describes the \textbf{number of elements} in the array; e.g., \text{[8]}

3. It has exactly as many values as it has elements, and in fact each of its elements contains exactly one of its values.

4. Its elements are accessed via \textit{indexing} with respect to the variable name; e.g., \text{a[2] = 7}

5. Its elements are \textit{contiguous} in memory; e.g.,

\[
\begin{array}{cccccccc}
\text{a[0]} & \text{a[1]} & \text{a[2]} & \text{a[3]} & \text{a[4]} & \text{a[5]} & \text{a[6]} & \text{a[7]} \\
\end{array}
\]

\[
\begin{array}{c}
\text{a[0]}: \ \boxed{????????} \ \text{(Address 12345)} \\
\text{a[1]}: \ \boxed{????????} \ \text{(Address 12349)} \\
\text{a[2]}: \ \boxed{????????} \ \text{(Address 12353)} \\
\text{a[3]}: \ \boxed{????????} \ \text{(Address 12357)} \\
\text{a[4]}: \ \boxed{????????} \ \text{(Address 12361)} \\
\text{a[5]}: \ \boxed{????????} \ \text{(Address 12365)} \\
\text{a[6]}: \ \boxed{????????} \ \text{(Address 12369)} \\
\text{a[7]}: \ \boxed{????????} \ \text{(Address 12373)} \\
\end{array}
\]
Array Indices

```c
int a[8];
```

We access a particular element of an array using *index* notation:

```c
a[2]
```

This notation is pronounced “a of 2” or “a sub 2.”

The number in parentheses – e.g., the 2 in `a[2]` – is called the *index* or *subscript* of the array element.

Array indices are exactly analogous to subscript numbers in mathematics:

```c
a_0, a_1, a_2, \ldots, a_7
```

An individual element of an array — e.g., `a[2]` — has exactly the same properties as a scalar variable of the same data type — except for being accessed via indexing.

Notice that the elements of an array are numbered from 0 through `(length - 1)`; in the above example, the elements of `a` are `a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]`.

An array can have **multiple** dimensions:

```c
int array2d[8][5];
```

For now, we’re going to concentrate on arrays with only one dimension.

A one-dimensional array is sometimes called a *vector*, because of the close relationship between arrays in computing and vectors in mathematics.
Array Declarations

The general form of an array declaration is:

\[ \text{type } \text{arrayname1}[\text{dimension1}], \text{arrayname2}[\text{dimension2}], \ldots \]

For example:

\[ \text{int a}[8], b[4], c[9]; \]

causes the compiler to set up three arrays in memory like so:

\[
\begin{array}{cccccccc}
\end{array}
\]

\[
\begin{array}{cccc}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\end{array}
\]

In principle, these arrays could be remote from each other in memory (e.g., \(a\) could start at address 12345, \(b\) could start at address 67891 and \(c\) could start at address 981439291).

In practice, they are usually contiguous or almost contiguous in memory; that is, the last byte of array \(a\) will typically be contiguous with the first byte of array \(b\), and the last byte of array \(b\) will typically be contiguous with the first byte of array \(c\).

However, the compiler is **not required** to make the different arrays contiguous in memory. The only contiguity constraint is that, within **each array**, all of the elements are contiguous.
Assigning a Value to an Array Element

Because an individual array element is exactly analogous to a scalar variable, we can assign or input a value into it in exactly the same ways that we assign or input values into scalar variables. For example, we can use a scalar assignment for each individual element:

```c
#include <stdio.h>

int main ()
{
    int a[3];
    a[0] = 5;
    a[1] = 16;
    a[2] = -77;
    printf("a[0] = %d\n", a[0]);
    printf("a[1] = %d\n", a[1]);
    printf("a[2] = %d\n", a[2]);
    return 0;
}
```

```sh
% gcc -o arrayeltassn arrayeltassn.c
% arrayeltassn
a[0] = 5
a[1] = 16
a[2] = -77
```
Using `scanf` to Obtain a Value of an Array Element

Just as we can assign a value to an individual array element, we can use `scanf` to obtain the value of each individual array element:

```
% cat arrayeltread.c
#include <stdio.h>

int main ()
{ /* main */
    float a[3];

    printf("Input a[0], a[1], a[2]:\n");
    scanf("%f %f %f", &a[0], &a[1], &a[2]);
    printf("a[0] = %f\n", a[0]);
    printf("a[1] = %f\n", a[1]);
    printf("a[2] = %f\n", a[2]);
    return 0;
} /* main */
% gcc -o arrayeltread arrayeltread.c
% arrayeltread
Input a[0], a[1], a[2]:
  5.5   16.16  -770.770
a[0] = 5.500000
a[1] = 16.160000
a[2] = -770.770020
```
Using **for** Loops to Accomplish Tasks on Arrays

Many of the tasks that we might want to accomplish using arrays can be helped a lot by using **for** loops. For example:

```c
#include <stdio.h>

int main ()
{
    const int a_length = 5;
    int a[a_length];
    int count;

    for (count = 0; count < a_length; count++)
    {
        a[count] = 2 * count;
    } /* for count */
    for (count = 0; count < a_length; count++)
    {
        printf("a[%2d] = %2d\n", count, a[count]);
    } /* for count */
    return 0;
} /* main */
```

```
go
array_for_mult
a[ 0] = 0
a[ 1] = 2
a[ 2] = 4
a[ 3] = 6
a[ 4] = 8
```
Using for Loops to Accomplish Tasks on Arrays (continued)

% cat array_for_mult_read.c
#include <stdio.h>

int main ()
{ /* main */
    const int minimum_a_length = 1;
    const int maximum_a_length = 15;
    int a[maximum_a_length];
    int a_length;
    int count;

    printf("How long will the array be (%d to %d)?\n", minimum_a_length, maximum_a_length);
    scanf("%d", &a_length);
    if ((a_length < minimum_a_length) ||
        (a_length > maximum_a_length)) {
        printf("That’s not a valid array length!\n");
        exit(-1);
    } /* if ((a_length < minimum_a_length) || ...) */
    for (count = 0; count < a_length; count++) {
        a[count] = 2 * count;
    } /* for count */
    for (count = 0; count < a_length; count++) {
        printf("a[%2d] = %2d\n", count, a[count]);
    } /* for count */
    return 0;
} /* main */
%
% gcc -o array_for_mult_read array_for_mult_read.c
% array_for_mult_read
How long will the array be (1 to 15)?
0
That’s not a valid array length!
%
% array_for_mult_read
How long will the array be (1 to 15)?
16
That’s not a valid array length!
%
% array_for_mult_read
How long will the array be (1 to 15)?
5
 a[ 0] = 0
 a[ 1] = 2
 a[ 2] = 4
 a[ 3] = 6
 a[ 4] = 8

Notice that we can declare an array to be larger than the portion of the array that we actually use, because RAM is cheap.
Reading in an Array’s Values Using a for Loop

Instead of having to explicitly read in every element of an array, we can do individual reads in a for loop:

```c
#include <stdio.h>

int main ()
{
    const int z_length = 6;
    float z[z_length], z_squared[z_length];
    int index;

    for (index = 0; index < z_length; index++) {
        printf("Input z(%d):\n", index);
        scanf("%f", &z[index]);
    }

    for (index = 0; index < z_length; index++) {
        z_squared[index] = z[index] * z[index];
    }

    for (index = 0; index < z_length; index++) {
        printf("%19.7f^2 = %19.7f\n", z[index], z_squared[index]);
    }

    return 0;
}
```

```bash
% gcc -o array_for_read_square array_for_read_square.c
% array_for_read_square
Input z(0):
5
Input z(1):
1.1
Input z(2):
-33.33333
Input z(3):
1.5e+05
Input z(4):
0.0033333
Input z(5):
1.5e-05
5.0000000^2 = 25.0000000
1.1000000^2 = 1.2100000
-33.3333282^2 = 1111.1107178
150000.0000000^2 = 22499999744.0000000
0.0033333^2 = 0.0000111
0.0000150^2 = 0.0000000
```

14
for Loop Behaves
the Same as Several Statements

Our for loop example on the previous page behaves exactly the same as a loopless version:

```c
#include <stdio.h>

int main ()
{
    const int z_length = 6;
    float z[z_length], z_squared[z_length];

    printf("Input z(%d):
", 0);
    scanf("%f", &z[0]);
    printf("Input z(%d):
", 1);
    scanf("%f", &z[1]);
    printf("Input z(%d):
", 2);
    scanf("%f", &z[2]);
    printf("Input z(%d):
", 3);
    scanf("%f", &z[3]);
    printf("Input z(%d):
", 4);
    scanf("%f", &z[4]);
    printf("Input z(%d):
", 5);
    scanf("%f", &z[5]);
    z_squared[0] = z[0] * z[0];
    z_squared[1] = z[1] * z[1];
    printf("%19.7f^2 = %19.7f
", z[0], z_squared[0]);
    printf("%19.7f^2 = %19.7f
", z[1], z_squared[1]);
    printf("%19.7f^2 = %19.7f
", z[2], z_squared[2]);
    printf("%19.7f^2 = %19.7f
", z[3], z_squared[3]);
    printf("%19.7f^2 = %19.7f
", z[4], z_squared[4]);
    printf("%19.7f^2 = %19.7f
", z[5], z_squared[5]);
    return 0;
}
```

% gcc -o array_no_for_read_square array_no_for_read_square.c
% array_no_for_read_square
Input z(0):
  5
Input z(1):
  1.1
Input z(2):
  -33.33333
Input z(3):
  1.5e+05
Input z(4):
  0.0033333
Input z(5):
  1.5e-05

  5.0000000^2 = 25.0000000
  1.1000000^2 = 1.2100000
-33.3333282^2 = 1111.1107178
150000.0000000^2 = 22499999744.0000000
  0.0033333^2 = 0.0000111
  0.0000150^2 = 0.0000000
Reading in an Array’s Values on a Single Line of Input Text

Instead of having to explicitly prompt for each array element, you can have a single prompt, and then the user can input all of the array elements’ values in a single line of input text:

```
% cat array_for_read_1line_square.c
#include <stdio.h>

int main ()
{
    /* main */
    const int z_length = 6;
    float z[z_length], z_squared[z_length];
    int index;

    printf("Input all %d values of z:\n", z_length);
    for (index = 0; index < z_length; index++) {
        scanf("%f", &z[index]);
    } /* for index */
    for (index = 0; index < z_length; index++) {
        z_squared[index] = z[index] * z[index];
    } /* for index */
    for (index = 0; index < z_length; index++) {
        printf("%19.7f^2 = %19.7f\n",
                z[index], z_squared[index]);
    } /* for index */
    return 0;
} /* main */
```

```
% gcc -o array_for_read_1line_square array_for_read_1line_square.c
% array_for_read_1line_square
Input all 6 values of z:
5 1.1 -33.3333 1.5e+05 0.0033333 1.5e-05
   5.0000000^2 = 25.0000000
   1.1000000^2 = 1.2100000
 -33.3333282^2 = 1111.1107178
  150000.0000000^2 = 22499999744.0000000
  0.0033333^2 = 0.0000111
  0.0000150^2 = 0.0000000
```
Aside: Why We Like Declaring Named Constants

Consider this program:

```c
#include <stdio.h>

int main ()
{
    float z[6], z_squared[6];
    int index;

    for (index = 0; index < 6; index++) {
        printf("Input z[%d]:\n", index);
        scanf("%f", &z[index]);
    } /* for index */

    for (index = 0; index < 6; index++) {
        z_squared[index] = z[index] * z[index];
    } /* for index */

    for (index = 0; index < 6; index++) {
        printf("%19.7f^2 = %19.7f\n", z[index], z_squared[index]);
    } /* for index */

    return 0;
} /* main */
```

What if we decide that we want to change the loop length?

Then we’d have to go in and change every for statement in the program.

That may not seem like much work in the above case, but it can be a lot of work with large programs.

For example, the Advanced Regional Prediction System (ARPS), the numerical weather prediction program created by the Center for Analysis & Prediction of Storms, is a Fortran 90 program that is almost 150,000 lines long, with over 5,800 loops. Changing the loop bounds on such a program would take a huge amount of work.
Declaring Named Constants for Loop Bounds

Rather than having to change the loop bounds for every single loop, we can instead declare named constants representing the loop bounds:

```c
#include <stdio.h>

int main ()
{
    /* main */
    const int z_length = 6;
    const int lower_bound = 0;
    float z[z_length], z_squared[z_length];
    int index;

    for (index = lower_bound; index < z_length; index++) {
        printf("Input z[%d]:\n", index);
        scanf("%f", &z[index]);
    } /* for index */

    for (index = lower_bound; index < z_length; index++) {
        z_squared[index] = z[index] * z[index];
    } /* for index */

    for (index = lower_bound; index < z_length; index++) {
        printf("%19.7f^2 = %19.7f\n", z[index], z_squared[index]);
    } /* for index */

    return 0;
} /* main */
```

```
Input z[0]:
5
Input z[1]:
1.1
Input z[2]:
-33.33333
Input z[3]:
1.5e+05
Input z[4]:
0.0033333
Input z[5]:
1.5e-05

5.0000000000^2 = 25.0000000000
1.1000000000^2 = 1.2100000000
-33.33333282^2 = 1111.11071780
150000.0000000000^2 = 22499999744.00000000
0.0033333^2 = 0.0000111
0.0000150^2 = 0.0000000
```
Computing with Arrays

We can compute with arrays much the same way we compute with scalar variables:

```
% cat arraysum.c
#include <stdio.h>

int main () {
    /* main */
    const int length = 10;
    const int lower_bound = 0;
    const int upper_bound = length - 1;
    int a[length];
    int sum;
    int index;

    printf("Input values #1d to #2d: \n", lower_bound, upper_bound);
    for (index = lower_bound; index < length; index++) {
        scanf("%d", &a[index]);
    } /* for index */
    sum = 0;
    for (index = lower_bound; index < length; index++) {
        sum = sum + a[index];
    } /* for index */
    printf("The sum of those values is %d. \n", sum);
    return 0;
} /* main */

% gcc -o arraysum arraysum.c
% arraysum
Input values #0 to # 9:
   1   4   9  16  25  36  49  64  81  100
The sum of those values is 385.
```
Computing with Arrays (continued)

Here’s another example of computing with arrays:

```c
#include <stdio.h>

int main ()
{
    /* main */
    const int length = 10;
    const int lower_bound = 0;
    const int upper_bound = length - 1;
    int a[length], b[length], c[length];
    int index;

    printf("Input a values #\%1d to #\%2d:\n", lower_bound, upper_bound);
    for (index = lower_bound; index < length; index++) {
        scanf("%d", &a[index]);
    } /* for index */
    printf("Input b values #\%1d to #\%2d:\n", lower_bound, upper_bound);
    for (index = lower_bound; index < length; index++) {
        scanf("%d", &b[index]);
    } /* for index */
    for (index = lower_bound; index < length; index++) {
        c[index] = a[index] + b[index];
    } /* for index */
    printf("The pairwise sums of the ");
    printf("%d array elements are:\n", length);
    for (index = lower_bound; index < length; index++) {
        printf("%d ", c[index]);
    } /* for index */
    printf("\n");
    return 0;
} /* main */
```

% gcc -o arrayadd arrayadd.c
% arrayadd

Input a values #0 to # 9:
1 8 27 64 125 216 343 512 729 1000
Input b values #0 to # 9:
1 4 9 16 25 36 49 64 81 100
The pairwise sums of the 10 array elements are:
2 12 36 80 150 252 392 576 810 1100
Static Memory Allocation

Up to now, all of the examples of array declarations that we’ve seen have involved array sizes that are explicitly stated as constants (named or literal), and that therefore are known at compile time.

We call this kind of declaration static, because the size and location of the array is set by the compiler at compile time and never changes after compilation.

```c
#include <stdio.h>

int main ()
{
    const int a_length = 5;
    int a[a_length];
    int count;

    for (count = 0; count < a_length; count++) {
        a[count] = 2 * count;
    }

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

    return 0;
}
```

% `cat array_for_mult.c`
```c```
#include <stdio.h>

int main ()
{
    const int a_length = 5;
    int a[a_length];
    int count;

    for (count = 0; count < a_length; count++) {
        a[count] = 2 * count;
    }

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

    return 0;
}
```c```

% `gcc -o array_for_mult array_for_mult.c`

% `array_for_mult`

a[ 0] = 0
a[ 1] = 2
a[ 2] = 4
a[ 3] = 6
a[ 4] = 8
Sometimes Static Memory Allocation Just Isn’t Good Enough

Often, we want to use an array whose size isn’t specifically known at compile time:

```c
#include <stdio.h>

int main ()
{
    const int minimum_a_length = 1;
    const int maximum_a_length = 15;
    int a[maximum_a_length];
    int a_length;
    int count;

    printf("How long will the array be (%d to %d)?\n", minimum_a_length, maximum_a_length);
    scanf("%d", &a_length);
    if ((a_length < minimum_a_length) || (a_length > maximum_a_length)) {
        printf("That’s not a valid array length!\n");
        exit(-1);
    }

    for (count = 0; count < a_length; count++) {
        a[count] = 2 * count;
    }
    for (count = 0; count < a_length; count++) {
        printf("a[\%2d] = \%2d\n", count, a[count]);
    }
    return 0;
}

% gcc -o array_for_mult_read array_for_mult_read.c
% array_for_mult_read
How long will the array be (1 to 15)?
   0
That’s not a valid array length!
% array_for_mult_read
How long will the array be (1 to 15)?
   16
That’s not a valid array length!
% array_for_mult_read
How long will the array be (1 to 15)?
   5
   a[ 0] = 0
   a[ 1] = 2
   a[ 2] = 4
   a[ 3] = 6
   a[ 4] = 8
```
Static Memory Allocation Can Be Wasteful

If the size of an array — or at least the number of elements that we want to use — isn’t known at compile time, we can always simply allocate an array that’s at least as big as the biggest array that we could imagine needing.

Of course, we might imagine that number to be pretty darn big.

On the one hand, memory is very cheap these days. On the other hand, we might reach the point where we can’t have the arrays we want, because we need too many arrays, any one of which might need to be big.

But what if we could allocate space for our arrays at runtime?
Dynamic Memory Allocation

 Dynamic memory allocation means allocating space for an array at runtime.

To use dynamic memory allocation, we have to declare our array variable, not as a static array, but rather as a pointer to an array of the same data type:

\[
\text{int* } a = (\text{int}*)\text{NULL};
\]

Notice that, when we declare the pointer, we initialize it to the null memory location, which means that the pointer doesn’t point to anything (yet).

We use the \text{malloc} function ("memory allocate") to allocate the array at runtime:

\[
a = (\text{int}*)\text{malloc}((\text{sizeof}(\text{int}) \times a\_length));
\]

The (\text{int}*) is called a \textit{type cast}, which we won’t go into detail about right now. You \textbf{MUST} use it when you use \text{malloc}.

When the \text{malloc} function is called, it returns a pointer to a location in memory that is an array whose size is the number of elements of the array being allocated times the size of each of the elements — that is, exactly enough space to fit the array being allocated.

Notice the \text{sizeof} function; it returns the number of bytes in a scalar of the given data type. For example, on a Pentium4 computer under the \texttt{gcc} compiler, \text{sizeof}(\text{int}) returns 4.

After the call to \text{malloc}, if the allocation is unsuccessful, then the pointer will still be null; if the allocation is successful, then the pointer will be anything except null.
Dynamic MemoryDeallocation

*Dynamic memory deallocation* means getting rid of the space for an array that has been dynamically allocated at runtime.

Often, this is done at the end of the program, though not always.

In C, the deallocate command is `free`.

For example, to deallocate an `int` array named `a`, do this:

```c
free(a);
a = (int*)NULL;
```

Notice that, after deallocating the array pointed to by `a`, we have to set `a` to null. We sometimes call this *nullifying* the pointer.
**Dynamic Memory Allocation Example**

```c
#include <stdio.h>

int main ()
{
    const int minimum_array_length = 1;
    int* a = (int*)NULL;
    int a_length;
    int count;

    printf("How long will the array be (at least %d)?\n", minimum_array_length);
    scanf("%d", &a_length);
    if (a_length < minimum_array_length) {
        printf("That’s not a valid array length!\n");
        exit(-1);
    } /* if (a_length < minimum_array_length) */
    a = (int*)malloc(sizeof(int) * a_length);
    if (a == (int*)NULL) {
        printf("ERROR: the attempt to allocate\n");
        printf(" array a failed.\n");
        exit(-1);
    } /* if (a == (int*)NULL) */
    for (count = 0; count < a_length; count++) {
        a[count] = 2 * count;
    } /* for count */
    for (count = 0; count < a_length; count++) {
        printf("a[%2d] = %2d\n", count, a[count]);
    } /* for count */
    free(a);
    a = (int*)NULL;
    return 0;
} /* main */
```

```bash
% gcc -o array_for_mult_read_dyn array_for_mult_read_dyn.c
% array_for_mult_read_dyn
How long will the array be (at least 1)?
 0
That’s not a valid array length!
% array_for_mult_read_dyn
How long will the array be (at least 1)?
 5
a[ 0] = 0
a[ 1] = 2
a[ 2] = 4
a[ 3] = 6
a[ 4] = 8
```
Exercise: Mean

Given a list of \( n \) real numbers

\[
(x_1, x_2, \ldots, x_n)
\]

the \textit{mean} of the values in the list is an \textit{average}, which is a value that is typical of the values in the list. The mean, here denoted \( \bar{x} \) (pronounced “x-bar”), is calculated as the sum of all the values in the list, divided by the number of values in the list:

\[
\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n}
\]

(Note: \( \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n \))

Write a program that:

1. greets the user;
2. prompts for and inputs the number of elements to be used;
3. dynamically allocates an array of appropriate length and type;
4. prompts for and inputs all of the elements of the array;
5. calculates the mean;
6. outputs the mean;
7. deallocates the array.

The program should work for any positive number of \texttt{float} elements.

The body of the program must not have any numeric literal constants; all constants must be declared using appropriate user-defined identifiers.

Don’t worry about comments, except for labeling block closes.