Array Lesson 1 Outline

1. Array Lesson 1 Outline
2. mean of a List of Numbers
3. mean: Declarations
4. mean: Greeting, Input
5. mean: Calculation
6. mean: Output
7. mean: Compile, Run
8. mean: 5 Input Values
9. mean: 7 Input Values
10. mean: One Line Different
11. mean: Compile, Run for 5
12. mean: Compile, Run for 7
13. Scalars #1
14. Scalars #2
15. Another Scalar Example
16. A Similar Program, with Multiplication
17. A Similar Program, with a Twist
18. Arrays
19. Array Element Properties
20. Array Properties #1
21. Array Properties #2
22. Array Properties #3
23. Array Properties #4
24. Array Properties #5
25. Array Indices #1
26. Array Indices #2
27. Multidimensional Arrays & 1D Arrays
28. Array Declarations #1
29. Array Declarations #2
30. Array Declarations #3
31. Assigning a Value to an Array Element
32. Array Element Assignment Example
33. Getting Array Element Value with scanf
34. Array Element scanf Example #1
35. Array Element scanf Example #2
36. for Loops for Tasks on Arrays #1
37. for Loops for Tasks on Arrays #2
38. Another for/Array Example #1
39. Another for/Array Example #2
40. Another for/Array Example #3
41. Don’t Need to Use Entire Declared Length
mean of a List of Numbers

Consider a list of real numbers of length $n$ elements:

$$x_1, x_2, x_3, \ldots, x_n$$

The **mean** (average) of this list is:

$$\frac{x_1 + x_2 + x_3 + \ldots + x_n}{n}$$
mean: Declarations

#include <stdio.h>

int main ()
{
    const float initial_sum = 0.0;
    const int number_of_elements = 5;
    const int first_element = 0;
    const int program_success_code = 0;
    float input_value[number_of_elements];
    float sum;
    float mean;
    int element;
mean: Greeting, Input

printf("I'm going to calculate the\n");
printf(" mean of a list of length %d values.\n", number_of_elements);
printf("What are the %d values of the list?\n", number_of_elements);
for (element = first_element;
     element < number_of_elements; element++) {
    scanf("%f", &input_value[element]);
} /* for element */
mean: Calculation

```c
sum = initial_sum;
for (element = first_element;
    element < number_of_elements; element++) {
    sum += input_value[element];
} /* for element */
mean = sum / number_of_elements;
```
mean: Output

```c
printf("The %d input values of the list are:\n", number_of_elements);
for (element = first_element;
    element < number_of_elements; element++) {
    printf("%f ", input_value[element]);
} /* for element */
printf("\n");
printf("The mean of the %d values", number_of_elements);
printf(" in the list is %f.\n", mean);
return program_success_code;
} /* main */
```
% gcc -o mean5 mean5.c
% mean5
I'm going to calculate the
  mean of a list of length 5 values.
What are the 5 values of the list?
123.25  234.50  345.75  456.00  567.25
The 5 input values of the list are:
  123.250000, 234.500000, 345.750000, 456.000000 and 567.250000.
The mean of the 5 values in the list is 345.350006.
mean: 5 Input Values

#include <stdio.h>

int main ()
{
    const float initial_sum = 0.0;
    const int number_of_elements = 5;
    const int first_element = 0;
    const int program_success_code = 0;
    float input_value[number_of_elements];
    float sum;
    float mean;
    int element;
mean: 7 Input Values

#include <stdio.h>

int main ()
{
    const float initial_sum = 0.0;
    const int number_of_elements = 7;
    const int first_element = 0;
    const int program_success_code = 0;
    float input_value[number_of_elements];
    float sum;
    float mean;
    int element;

    The rest of the program is EXACTLY THE SAME!
mean: One Line Different

% diff mean5.c mean7.c

6c6
<     const int    number_of_elements = 5;
---
>     const int    number_of_elements = 7;

The `diff` Unix command compares two files of text and shows which lines are different.
The only statement that differs between `mean5.c` and `mean7.c` is the declaration of `number_of_elements`. 
mean: Compile, Run for 5

% gcc -o mean5 mean5.c
% mean5

I'm going to calculate the

mean of a list of length 5 values.

What are the 5 values of the list?

123.25 234.50 345.75 456.00 567.25

The 5 input values of the list are:

123.250000, 234.500000, 345.750000, 456.000000 and 567.250000.

The mean of the 5 values in the list is 345.350006.
mean: Compile, Run for 7

```bash
% gcc -o mean7 mean7.c
% mean7
I'm going to calculate the
  mean of a list of length 7 values.
What are the 7 values of the list?
12.75 23.75 34.75 45.75 56.75 67.75 78.75
The 7 input values of the list are:
12.750000 23.750000 34.750000 45.750000 56.750000 67.750000 78.750000
The mean of the 7 values in the list is 45.750000.
```
 Scalars #1

% cat scalar_names.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 */

% gcc -o scalar_names scalar_names.c
% scalar_names
b = 0
c = 2
d = 4
e = 6
f = 8

Note that, in Unix, a **backslash** at the end of a Unix command line means: “continue this Unix command on the next line.”
% cat scalar_names.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 */

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, char or whatever, are referred to as **scalar** variables.
Another Scalar Example

% 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).
A Similar Program, with Multiplication

```
#include <stdio.h>

int main()
{
    int a0, a1, a2, a3, a4;
    a0 = 0 * 2;
    a1 = 1 * 2;
    a2 = 2 * 2;
    a3 = 3 * 2;
    a4 = 4 * 2;
    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.
A Similar Program, with a Twist

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

int main ()
{ /* main */
  int a[5];
  a[0] = 0 * 2;
  a[1] = 1 * 2;
  a[2] = 2 * 2;
  a[3] = 3 * 2;
  a[4] = 4 * 2;
  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?
int a[5];

An **array** is a special kind of variable. Like a scalar variable, an array 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** in mathematics, 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 just about *exactly like a scalar variable of the same data type*.

An *element* of an array has:

1. a *name*, 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 *element* of an array also has:

5. an *index*, which we’ll learn about shortly.
Array Properties #1

```
int a[5];
```

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; for example, `int`. 
int a[5];

An array **as a whole** has the following properties:

2. It as a **dimension** attribute, sometimes called its **length**, which describes the **number of elements** in the array; for example, [5].
Array Properties #3

```
int a[5];
```

An array **as a whole** has the following properties:

3. It has exactly as many **values** as it has elements, and in fact each of its elements contains exactly one of its values.
int a[5];

An array **as a whole** has the following properties:

4. Its elements are accessed via **indexing** with respect to the variable name; for example,
   
   $$a[2] = 7;$$
Array Properties #5

```c
int a[5];
```

An array **as a whole** has the following properties:

5. Its elements are **contiguous** in memory; for example,

```
```

```
```

```
a[0]  ???????  Address 12340
```
```
a[1]  ???????  Address 12344
```
```
a[2]  ???????  Address 12348
```
```
a[3]  ???????  Address 12352
```
```
a[4]  ???????  Address 12356
```
int a[5];

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

a[2]

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

The number in square brackets – for example, 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:

a₀, a₁, a₂, a₃, a₄
int a[5];

An individual element of an array – for example, \(a[2]\) – 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 (\textit{length} \ - \ 1); in the above example, the elements of \(a\) are

\[a[0], a[1], a[2], a[3], a[4]\]
Multidimensional Arrays & 1D Arrays

An array can have **multiple dimensions**: 
```
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 #1

The general form of an array declaration is:

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

For example:

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

causes the compiler to set up three int arrays in memory.
Array Declarations #2

```c
int a[5], b[4], c[9];
```

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

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
int a[8], b[4], c[9];

In principle, these arrays could be remote from each other in memory (for example, \( a \) could start at address 12340, \( b \) could start at address 67890 and \( c \) could start at address 981439294).

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

However, the compiler isn’t required to make the different arrays contiguous in memory.

The only contiguity constraint is that, \textbf{within each array}, all of the elements are contiguous and sequential.
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.
Array Element Assignment Example

% cat arrayeltassn.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;
} /* main */

% gcc -o arrayeltassn arrayeltassn.c
% arrayeltassn
a[0] = 5
a[1] = 16
a[2] = -77
Getting Array Element Value with scanf

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.
#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 */
Array Element `scanf` Example #2

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

int main ()
{ /* 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 */
for Loops for Tasks on Arrays #2

```bash
% 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
```
```c
#include <stdio.h>
#include <stdlib.h>

int main ()
{ /* main */
    const int minimum_a_length = 1;
    const int maximum_a_length = 15;
    const int program_failure_code = -1;
    const int program_success_code = 0;
    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(program_failure_code);
    } /* if ((a_length < minimum_a_length) || ...) */
```
Another for/Array Example #2

```c
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 program_success_code;
} /* main */
```
Another for/Array Example #3

```bash
% 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
```
#include <stdio.h>

int main ()
{ /* main */
    const int minimum_a_length = 1;
    const int maximum_a_length = 15;
    const int program_failure_code = -1;
    const int program_success_code = 0;
    int a[maximum_a_length];
    ...
} /* main */

...% 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.