



Array Lesson 1 Outline

1. Array Lesson 1 Outline
2. Arithmetic Mean of a List of Numbers
3. Arithmetic Mean: Declarations
4. Arithmetic Mean: Greeting, Input
5. Arithmetic Mean: Calculation
6. Arithmetic Mean: Output
7. Arithmetic Mean: Compile, Run
8. Arithmetic Mean: 5 Input Values
9. Arithmetic Mean: 7 Input Values
10. Arithmetic Mean: One Line Different
11. Arithmetic Mean: Compile, Run for 5
12. Arithmetic 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





Arithmetic Mean of a List of Numbers

Consider a list of real numbers of length n elements:

$$x_1, x_2, x_3, \dots, x_n$$

The arithmetic mean (average) of this list is:

$$(x_1 + x_2 + x_3 + \dots + x_n) / n$$





Arithmetic Mean: Declarations

```
#include <stdio.h>

int main ()
{ /* 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 arithmetic_mean;
    int   element;
```





Arithmetic Mean: Greeting, Input

```
printf("I'm going to calculate the\n");
printf("  arithmetic mean of a list ");
printf("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 */
```





Arithmetic Mean: Calculation

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





Arithmetic Mean: Output

```
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 arithmetic mean of the %d values",
      number_of_elements);
printf(" in the list is %f.\n",
      arithmetic_mean);
return program_success_code;
} /* main */
```





Arithmetic Mean: Compile, Run

```
% gcc -o arithmetic_mean5 arithmetic_mean5.c  
% arithmetic_mean5
```

I'm going to calculate the
arithmetic 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 567.250000

The arithmetic mean of the 5 values in the list is 345.350006.





Arithmetic Mean: 5 Input Values

```
#include <stdio.h>

int main ()
{ /* 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 arithmetic_mean;
    int   element;
```





Arithmetic Mean: 7 Input Values

```
#include <stdio.h>

int main ()
{ /* 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 arithmetic_mean;
    int   element;
```

**The rest of the program is
EXACTLY THE SAME!**





Arithmetic Mean: One Line Different

```
% diff arithmetic_mean5.c arithmetic_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 `arithmetic_mean5.c` and `arithmetic_mean7.c` is the declaration of `number_of_elements`.





Arithmetic Mean: Compile, Run for 5

```
% gcc -o arithmetic_mean5 arithmetic_mean5.c
```

```
% arithmetic_mean5
```

I'm going to calculate the

arithmetic 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 567.250000
```

The arithmetic mean of the 5 values in the list is 345.350006.





Arithmetic Mean: Compile, Run for 7

```
% gcc -o arithmetic_mean7 arithmetic_mean7.c
```

```
% arithmetic_mean7
```

I'm going to calculate the

arithmetic 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 arithmetic 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.”





Scalars #2

```
% 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

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

int main ()
{ /* 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;
} /* main */
```

```
% gcc -o scalar_mult \
    scalar_mult.c

% scalar_mult
a0 = 0
a1 = 2
a2 = 4
a3 = 6
a4 = 8
```

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;
} /* main */
```

```
% 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[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`.





Array Properties #2

```
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.





Array Properties #4

```
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

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

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

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

a[0]	a[1]	a[2]	a[3]	a[4]
?	?	?	?	?

a[0]	????????	Address 12340
a[1]	????????	Address 12344
a[2]	????????	Address 12348
a[3]	????????	Address 12352
a[4]	????????	Address 12356





Array Indices #1

```
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:

$$a_0, a_1, a_2, a_3, a_4$$




Array Indices #2

```
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 (**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];
```

In CS1313, we're going to focus on arrays of 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.

A two-dimensional array is sometimes called a **matrix**.

A three-dimensional array is sometimes called a **field**.





Array Declarations #1

The general form of an array declaration is:

```
type arrayname1 [dimension1], arrayname2 [dimension2], ... ;
```

For example:

```
int a[8], b[4], c[9];
```

causes the compiler to set up three `int` arrays in memory.

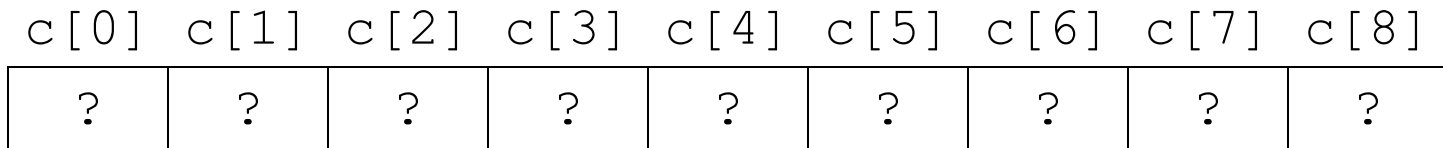
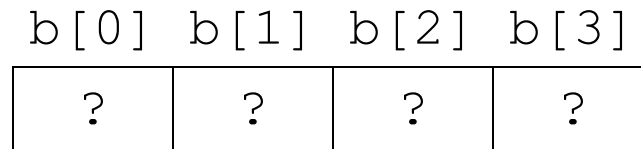
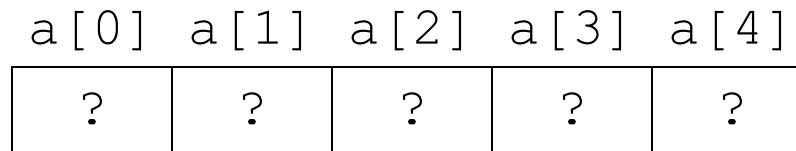




Array Declarations #2

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

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





Array Declarations #3

```
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, **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 ()
```

```
{ /* 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.





Array Element scanf Example #1

```
#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
```

