Array Lesson 2 Outline

1. Array Lesson 2 Outline
2. Reading Array Values Using for Loop #1
3. Reading Array Values Using for Loop #2
4. for Loop: Like Many Statements #1
5. for Loop: Like Many Statements #2
6. for Loop: Like Many Statements #3
7. Reading Array on One Line of Input #1
8. Reading Array on One Line of Input #2
9. Reading Array on One Line of Input #3
10.Aside: Why Named Constants Are Good
11. Named Constants as Loop Bounds #1
12. Named Constants as Loop Bounds #2
13. Computing with Arrays #1
14. Computing with Arrays #2
15. Computing with Arrays #3
16. Computing with Arrays #4
17. Computing with Arrays #5
18. Static Memory Allocation
19. Static Memory Allocation Example #1
20. Static Memory Allocation Example #2
21. Static Sometimes Not Good Enough #1
22. Static Sometimes Not Good Enough #2
23. Static Sometimes Not Good Enough #3
24. Static Sometimes Not Good Enough #4
25. Static Memory Allocation Can Be Wasteful
26. Dynamic Memory Allocation Example #1
27. Dynamic Memory Allocation Example #2
28. Dynamic Memory Allocation Example #3
29. Dynamic Memory Allocation #1
30. Dynamic Memory Allocation #2
31. Dynamic Memory Allocation #3
32. Dynamic Memory Allocation #4
33. Dynamic Memory Deallocation
34. Dynamic Memory Allocation Example #1
35. Dynamic Memory Allocation Example #2
36. Dynamic Memory Allocation Example #3
37. Dynamic Memory Allocation #1
38. Dynamic Memory Allocation #2
39. Dynamic Memory Allocation #3
40. Dynamic Memory Allocation #4
41. Dynamic Memory Allocation #5
42. Dynamic Memory Allocation #6
43. Dynamic Memory Allocation #7
44. Dynamic Memory Allocation #8
45. Dynamic Memory Allocation: Run
Reading Array Values Using for Loop #1

```c
#include <stdio.h>

int main ()
{ /* main */
    const int z_length = 6;
    const int program_success_code = 0;
    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 */
    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 program_success_code;
} /* main */
```

“Use at least 19 spaces total, 7 of which are to the right of the decimal point.”
Reading Array Values Using `for` Loop #2

```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]:
-33.3333
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
```
for Loop: Like Many Statements #1

```c
#include <stdio.h>

int main ()
{
    const int z_length = 6;
    const int program_success_code = 0;
    float z[z_length], z_squared[z_length];

    printf("Input z[%d]: \n", 0);
    scanf("%f", &z[0]);
    printf("Input z[%d]: \n", 1);
    scanf("%f", &z[1]);
    printf("Input z[%d]: \n", 2);
    scanf("%f", &z[2]);
    printf("Input z[%d]: \n", 3);
    scanf("%f", &z[3]);
    printf("Input z[%d]: \n", 4);
    scanf("%f", &z[4]);
    printf("Input z[%d]: \n", 5);
    scanf("%f", &z[5]);
```
for Loop: Like Many Statements #2

```c
for (int i = 0; i < 6; i++) {
    z_squared[i] = z[i] * z[i];
    printf("%.7f^2 = %.7f
", z[i], z_squared[i]);
}
return program_success_code;
} /* main */
```
for Loop: Like Many Statements #3

```bash
% 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
-33.3333328^2 = 1111.1107178
  150000.0000000^2 = 22499999744.0000000
  0.0033333^2 = 0.0000111
  0.0000150^2 = 0.0000000
```
Reading Array on One Line of Input #1

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.
Reading Array on One Line of Input #2

#include <stdio.h>

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

    printf("Input all %d values of z:\n", z_length);
    for (index = 0; index < 6; 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 program_success_code;
} /* main */
Reading Array on One Line of Input #3

% 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 -33.33333 1.5e+05 0.0033333 1.5e-05

\[
\begin{align*}
5.0000000^2 &= 25.0000000 \\
1.1000000^2 &= 1.2100000 \\
-33.3333282^2 &= 1111.1107178 \\
150000.0000000^2 &= 22499999744.0000000 \\
0.0033333^2 &= 0.0000111 \\
0.0000150^2 &= 0.0000000
\end{align*}
\]
Aside: Why Named Constants Are Good

Consider the previous program.
What if we decide that we want to change the array 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 previous program, 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 OU’s 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.
Named Constants as Loop Bounds #1

```c
#include <stdio.h>

int main ()
{ /* main */
    const int z_length = 6;
    const int lower_bound = 0;
    const int program_success_code = 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 program_success_code;
} /* main */
```
Named Constants as Loop Bounds #2

% gcc -o array_for_read_named \ array_for_read_named.c

% array_for_read_named
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
#include <stdio.h>

int main ()
{ /* main */
    const float initial_sum = 0.0;
    const int length = 10;
    const int lower_bound = 0;
    const int upper_bound = length - 1;
    const int program_success_code = 0;
    int a[length];
    int sum;
    int index;

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

% gcc -o array_sum array_sum.c
% array_sum
Input values #0 to #9:
1  4  9  16  25  36  49  64  81  100
The sum of those values is 385.
#include <stdio.h>

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

    printf("Input a values #%d to #%d:\n",
           lower_bound, upper_bound);
    for (index = lower_bound; index < length; index++) {
        scanf("%d", &a[index]);
    } /* for index */
    printf("Input b values #%d to #%d:\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 program_success_code;
} /* main */
Computing with Arrays #5

% gcc -o array_add_pairwise array_add_pairwise.c
% array_add_pairwise

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 are set by the compiler at compile time, and they never change after compilation.
Static Memory Allocation Example #1

```c
#include <stdio.h>

int main ()
{
    const int number_of_elements = 5;
    const int program_success_code = 0;
    int a[number_of_elements];
    int count;

    for (count = 0; count < number_of_elements; count++)
    {
        a[count] = 2 * count;
    }
    for (count = 0; count < number_of_elements; count++)
    {
        printf("a[%2d] = %2d\n", count, a[count]);
    }
    return program_success_code;
}
```
Static Memory Allocation Example #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
```
Static Sometimes Not Good Enough #1

Often, we want to use an array – or perhaps many arrays – whose sizes aren’t specifically known at compile time.
#include <stdio.h>
#include <stdlib.h>

int main ()
{ /* main */
    const int minimum_number_of_elements = 1;
    const int maximum_number_of_elements = 15;
    const int program_failure_code = -1;
    const int program_success_code = 0;
    int a[maximum_number_of_elements];
    int number_of_elements;
    int count;

    printf("How long will the array be (%d to %d)?\n",
        minimum_number_of_elements,
        maximum_number_of_elements);
    scanf("%d", &number_of_elements);
    if ((number_of_elements < minimum_number_of_elements) ||
        (number_of_elements > maximum_number_of_elements))
    {
        printf("That’s not a valid array length!\n");
        exit(program_failure_code);
    } /* if ((number_of_elements < ...) || ...) */
```c
for (count = 0; count < number_of_elements; count++) {
    a[count] = 2 * count;
} /* for count */
for (count = 0; count < number_of_elements; count++) {
    printf("a[%2d] = %2d\n", count, a[count]);
} /* for count */
return program_success_code;
} /* 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)?
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, then we could 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 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 #1

*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:

```c
float* list1_input_value = (float*)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).
Dynamic Memory Allocation #2

We use the `malloc` function ("memory allocate") to allocate the array at runtime, once we know its length:

```c
list1_input_value = (float*)malloc(sizeof(float) * number_of_elements);
```

The `(float*)` is called a `type cast`, which we won’t go into detail about right now.

You **MUST** use it when you use `malloc`.

When the `malloc` function is called, it returns a pointer to a location in memory that is the first byte of an array whose size is the number of elements of the array that is being allocated, times the size of each of the elements – that is, exactly enough space to fit the array being allocated.
Dynamic Memory Allocation #3

Notice the `sizeof` function; it returns the number of bytes in a scalar of the given data type. For example, on an Intel/AMD x86 computer under the `gcc` compiler, `sizeof(float)` returns 4.
Dynamic Memory Allocation #4

After the call to `malloc`:

- If the allocation is **unsuccessful**, then the pointer will still be **null**.
- If the allocation is **successful**, then the pointer will be **something other than null**.

```c
list1_input_value = (float*)malloc(sizeof(float) * number_of_elements);
if (list1_input_value == (float*)NULL) {
    printf("ERROR: the attempt to allocate\n");
    printf(" first input array failed.\n");
    exit(program_failure_code);
} /* if (list1_input_value == (float*)NULL) */
```
Dynamic Memory Deallocation

Dynamic memory deallocation means freeing up 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 named `free`.

For example, to deallocate a `float` array named `list1_input_value`, do this:

```c
free(list1_input_value);
list1_input_value = (float*)NULL;
```

Notice that, after deallocating the array pointed to by `list1_input_value`, we have to set `list1_input_value` to null. We sometimes refer to this as nullifying the pointer.
Dynamic Memory Allocation Example #1

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

int main ()
{ /* main */
    const int minimum_number_of_elements = 1;
    const int program_failure_code = -1;
    const int program_success_code = 0;
    float* array = (float*)NULL;
    int number_of_elements;
    int count;

    printf("How long will the array be (at least %d)?\n", minimum_number_of_elements);
    scanf("%d", &number_of_elements);
    if (number_of_elements < minimum_number_of_elements) {
        printf("That's not a valid array length!\n");
        exit(program_failure_code);
    } /* if (number_of_elements < minimum_number_of_elements) */
```
Dynamic Memory Allocation Example #2

```c
array = (float*)malloc(sizeof(float) * number_of_elements);
if (array == (float*)NULL) {
    printf("ERROR: the attempt to allocate
    array failed.\n");
    exit(program_failure_code);
} /* if (array == (float*)NULL) */
for (count = 0; count < number_of_elements; count++) {
    array[count] = 2.5 * count;
} /* for count */
for (count = 0; count < number_of_elements; count++) {
    printf("array[%2d] = %4.1f\n", count, array[count]);
} /* for count */
free(array);
array = (float*)NULL;
return program_success_code;
} /* main */
```
Dynamic Memory Allocation Example #3

```
gcc -o array_for_mult_read_dynamic array_for_mult_read_dynamic.c

array_for_mult_read_dynamic
How long will the array be (at least 1)?
0
That’s not a valid array length!

array_for_mult_read_dynamic
How long will the array be (at least 1)?
5

array[ 0] =  0.0
array[ 1] =  2.5
array[ 2] =  5.0
array[ 3] =  7.5
array[ 4] = 10.0
```
#include <stdio.h>
#include <stdlib.h>

int main ()
{ /* main */
    const float initial_sum = 0.0;
    const int minimum_number_of_elements = 1;
    const int first_element = 0;
    const int program_success_code = 0;
    const int program_failure_code = -1;
    float* list1_input_value = (float*)NULL;
    float* list2_input_value = (float*)NULL;
    float list1_input_value_sum, arithmetic_mean1;
    float list2_input_value_sum, arithmetic_mean2;
    int number_of_elements;
    int element;
}
printf("I'm going to calculate the arithmetic mean of\n");
printf(" a pair of lists of values that you input.\n");
printf("These lists will have the same length.\n");
printf("How many values would you like to\n");
printf(" calculate the arithmetic mean of in each list?\n");
scanf("%d", &number_of_elements);
if (number_of_elements < minimum_number_of_elements) {
    printf("ERROR: Can't calculate the arithmetic mean of %d values.\n",
            number_of_elements);
    exit(program_failure_code);
} /* if (number_of_elements < minimum_number_of_elements) */
list1_input_value =
    (float*)malloc(sizeof(float) * number_of_elements);
if (list1_input_value == (float*)NULL) {
    printf("ERROR: Can't allocate the 1st float array\n");
    printf(" of length %d.\n", number_of_elements);
    exit(program_failure_code);
} /* if (list1_input_value == (float*)NULL) */

list2_input_value =
    (float*)malloc(sizeof(float) * number_of_elements);
if (list2_input_value == (float*)NULL) {
    printf("ERROR: Can't allocate the 2nd float array\n");
    printf(" of length %d.\n", number_of_elements);
    exit(program_failure_code);
} /* if (list2_input_value == (float*)NULL) */
Arithmetic Mean of Dynamically Allocated Array #5

printf("What are the pair of lists of %d values each\n",
    number_of_elements);
printf(" to calculate the arithmetic mean of?\n");
for (element = first_element;
    element < number_of_elements; element++) {
    scanf("%f %f",
        &list1_input_value[element],
        &list2_input_value[element]);
} /* for element */
list1_input_value_sum = initial_sum;
for (element = first_element;
    element < number_of_elements; element++) {
    list1_input_value_sum =
        list1_input_value_sum + list1_input_value[element];
} /* for element */
arithmetic_mean1 =
    list1_input_value_sum / number_of_elements;

list2_input_value_sum = initial_sum;
for (element = first_element;
    element < number_of_elements; element++) {
    list2_input_value_sum =
        list2_input_value_sum + list2_input_value[element];
} /* for element */
arithmetic_mean2 =
    list2_input_value_sum / number_of_elements;
printf("The %d pairs of input values are:\n", number_of_elements);
for (element = first_element;
    element < number_of_elements; element++) {
    printf("%f %f\n",
        list1_input_value[element],
        list2_input_value[element]);
} /* for element */
printf("The arithmetic mean of the 1st list of %d input values is %f.\n", number_of_elements, arithmetic_mean1);
printf("The arithmetic mean of the 2nd list of %d input values is %f.\n", number_of_elements, arithmetic_mean2);
free(list2_input_value);
list2_input_value = (float*)NULL;
free(list1_input_value);
list1_input_value = (float*)NULL;
return program_success_code;

} /* main */
I'm going to calculate the arithmetic mean of a pair of lists of values that you input. These lists will have the same length.

How many values would you like to calculate the arithmetic mean of in each list?

5

What are the pair of lists of 5 values each
to calculate the arithmetic mean of?

1.1 11.11
2.2 22.22
3.3 33.33
4.4 44.44
9.9 99.99

The 5 pairs of input values are:
1.100000 11.110000
2.200000 22.219999
3.300000 33.330002
4.400000 44.439999
9.900000 99.989998

The arithmetic mean of the 1st list of 5 input values is 4.180000.
The arithmetic mean of the 2nd list of 5 input values is 42.217999.