Array Lesson 3 Outline

1. Array Lesson 3 Outline
2. Static Memory Allocation
3. Static Memory Allocation Example #1
4. Static Memory Allocation Example #2
5. Static Sometimes Not Good Enough #1
6. Static Sometimes Not Good Enough #2
7. Static Sometimes Not Good Enough #3
8. Static Sometimes Not Good Enough #4
9. Static Memory Allocation Can Be Wasteful
10. Dynamic Memory Allocation #1
11. Dynamic Memory Allocation #2
12. Dynamic Memory Allocation #3
13. Dynamic Memory Allocation #4
14. Dynamic MemoryDeallocation
15. Dynamic Memory Allocation Example #1
16. Dynamic Memory Allocation Example #2
17. Dynamic Memory Allocation Example #3
18. Mean of Dynamically Allocated Array #1
19. Mean of Dynamically Allocated Array #2
20. Mean of Dynamically Allocated Array #3
21. Mean of Dynamically Allocated Array #4
22. Mean of Dynamically Allocated Array #5
23. Mean of Dynamically Allocated Array #6
24. Mean of Dynamically Allocated Array #7
25. Mean of Dynamically Allocated Array #8
26. Mean of Dynamically Allocated Array: Run
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 array declaration static, because the size and location of the array are set by the compiler at compile time, and they don’t change at runtime.
Static Memory Allocation Example #1

```c
#include <stdio.h>

int main ()
{ /* 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 */
    for (count = 0; count < number_of_elements; count++) {
        printf("a[%2d] = %2d\n", count, a[count]);
    } /* for count */
    return program_success_code;
} /* main */
```
Static Memory Allocation Example #2

% 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
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 < ...) || ...) */
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 */
Static Sometimes Not Good Enough #4

```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)?

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 several arrays we want, because we need too many arrays, each 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).

Presently, we’ll talk about why we do that (but not 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 the first element of an array whose size is the number of elements of the array that is being allocated, times the size (in bytes) of each of the elements — that is, exactly enough bytes to fit the array being allocated.
list1_input_value =
    (float*)malloc(sizeof(float) * number_of_elements);

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.
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) */
```

The check of the pointer variable’s value **MUST** occur **IMMEDIATELY AFTER** the call to `malloc` (similar to idiotproofing).
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

#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\n");
    printf(" 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

```c
% 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 ()
{
    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, mean1;
    float list2_input_value_sum, mean2;
    int number_of_elements;
    int element;
Mean of Dynamically Allocated Array #2

printf("I'm going to calculate the mean of\n");
printf(" a pair of lists of values that you input.\n");
printf("These lists will have the same length.\n");
Mean of Dynamically Allocated Array #3

printf("How many values would you like to\n");
printf(" calculate the 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 mean of %d values.\n", number_of_elements);
    exit(program_failure_code);
} /* if (number_of_elements < minimum_number_of_elements) */
Mean of Dynamically Allocated Array #4

    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) */
printf("What are the pair of lists of %d values each\n", number_of_elements);
printf(" to calculate the 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 */
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 */
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 mean of the 1st list of %d input values is %f.\n", number_of_elements, mean1);
printf("The mean of the 2nd list of %d input values is %f.\n", number_of_elements, 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 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 mean of in each list?

5

What are the pair of lists of 5 values each to calculate the 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 mean of the 1st list of 5 input values is 4.180000. The mean of the 2nd list of 5 input values is 42.217999.