



User Defined Functions 2 Outline

1. User Defined Functions 2 Outline
2. Argument Order When Passing Arrays #1
3. Argument Order When Passing Arrays #1
4. Code Reuse Is GOOD GOOD GOOD #1
5. Code Reuse Is GOOD GOOD GOOD #2
6. Actual vs. Formal Arguments #1
7. Actual vs. Formal Arguments #2
8. Argument Order
9. Argument Order in Function: Arbitrary #1
10. Argument Order in Function: Arbitrary #2
11. Actual EXACTLY MATCH Formal #1
12. Actual EXACTLY MATCH Formal #2
13. Argument Order Convention #1
14. Argument Order Convention #2
15. Side Effects #1
16. Side Effects #2
17. Side Effects Example #1
18. Side Effects Example #2
19. Side Effects Example #3
20. A Function That Doesn't Return a Value #1
21. A Function That Doesn't Return a Value #2
22. void Functions #1
23. void Functions #2
24. void Function Call Example #1
25. void Function Call Example #2
26. void Function Call Example #3
27. void Function Call Example #4
28. Why Do We Like Code Reuse?
29. Why Do We Like User-Defined Functions?





Argument Order When Passing Arrays #1

```
float mean (float* array, int number_of_elements)
```

When we pass an array to a function as an argument, we also need to pass its length, because the declared length of the array in the `main` function, or the length that was dynamically allocated at runtime, is not automatically known by the function.

When passing an array as a function argument – and therefore passing the length of the array as well – it does not matter what order the formal arguments appear in the function's formal argument list, as long as they match the actual argument list.





Argument Order When Passing Arrays #1

```
float mean (float* array, int number_of_elements)
```

When passing an array as a function argument – and therefore passing the length of the array as well – it does not matter what order the formal arguments appear in the function’s formal argument list.

HOWEVER, it matters very much that the order of the formal arguments in the function’s formal argument list **EXACTLY MATCH** the order of the actual arguments in the function call.

IMPORTANT NOTE:

The length argument **MUST** be an `int`.





Code Reuse Is GOOD GOOD GOOD #1

We like to make our programming experiences reasonably efficient.

Often, we find ourselves doing a particular task the same way in many different contexts.

It doesn't make sense, from a software development point of view, to have to type in the same piece of source code over and over and over.

So, in solving a new problem – that is, in writing a new program – we want to be able to **reuse** as much existing source code as we possibly can.

Not surprisingly, this is called **code reuse**.





Code Reuse Is **GOOD GOOD GOOD** #2

Code reuse is **GOOD GOOD GOOD**.

It makes us happy as programmers, because:

1. We can get to the solution of a new problem much more quickly.
2. We can thoroughly test and debug a piece of source code that does a common, well-defined task, and then be confident that it will work well in a new context.





Actual vs. Formal Arguments #1

In our cube root examples, we've seen function calls that look like this:

```
cube_root_value1 = cube_root(input_value1);
```

We say that:

- this assignment statement
- calls the user-defined function `cube_root`
- using as its *actual argument* the variable `input_value1`
- which corresponds to the function definition's *formal argument* `base`
- and returns the cube root of
- the value stored in the variable `input_value1`.





Actual vs. Formal Arguments #2

The actual argument is the argument that appears in the call to the function (for example, in the main function).

The formal argument is the argument that appears in the definition of the function.

Not surprisingly, the mathematical case is the same. In a mathematical function definition like

$$f(x) = x + 1$$

if we want the value of

$$f(1)$$

then x is the formal argument of the function f , and 1 is the actual argument.





Argument Order

Suppose that a function has multiple arguments. Does their order matter?

No, **yes** and **yes**.

No, in the sense that the order of arguments in the function definition is arbitrary.

Yes, in the sense that the order of the formal arguments in the function definition must **EXACTLY MATCH** the order of the actual arguments in the function call.

Yes, in the sense that it's a good idea to set a convention for how you're going to order your arguments, and then to stick to that convention.



Argument Order in Function: Arbitrary #1

```
float mean (float* array, int number_of_elements)
{ /* mean */
    const float initial_sum          = 0.0;
    const int   minimum_number_of_elements = 1;
    const int   first_element        = 0;
    const int   program_failure_code  = -1;
    float sum;
    int  element;

    if (number_of_elements < minimum_number_of_elements) {
        printf("ERROR: can't have an array ");
        printf("of length %d:\n", number_of_elements);
        printf(" it must have at least %d element.\n",
            minimum_number_of_elements);
        exit(program_failure_code);
    } /* if (number_of_elements < ...) */
    sum = initial_sum;
    for (element = first_element;
        element < number_of_elements; element++) {
        sum = sum + array[element];
    } /* for element */
    return sum / number_of_elements;
} /* mean */
```



Argument Order in Function: Arbitrary #2

```
float mean (int number_of_elements, float* array)
{ /* mean */
    const float initial_sum          = 0.0;
    const int   minimum_number_of_elements = 1;
    const int   first_element        = 0;
    const int   program_failure_code  = -1;
    float sum;
    int element;

    if (number_of_elements < minimum_number_of_elements) {
        printf("ERROR: can't have an array ");
        printf("of length %d:\n", number_of_elements);
        printf(" it must have at least %d element.\n",
            minimum_number_of_elements);
        exit(program_failure_code);
    } /* if (number_of_elements < ...) */
    sum = initial_sum;
    for (element = first_element;
        element < number_of_elements; element++) {
        sum = sum + array[element];
    } /* for element */
    return sum / number_of_elements;
} /* mean */
```





Actual EXACTLY MATCH Formal #1

```
#include <stdio.h>

int main ()
{ /* main */
    ...
    input_value1_mean =
        mean(input_value, number_of_elements);
    ...
} /* main */

float mean (float* array, int number_of_elements)
{ /* mean */
    ...
} /* main */
```





Actual EXACTLY MATCH Formal #2

```
#include <stdio.h>

int main ()
{ /* main */
    ...
    input_value1_mean =
        mean(number_of_elements, input_value);
    ...
} /* main */

float mean (int number_of_elements, float* array)
{ /* mean */
    ...
} /* main */
```





Argument Order Convention #1

In general, it's good practice to pick a convention for how you will order your argument lists, and to stick with that convention.

The reason for this is that, as you develop your program, you'll jump around a lot from place to place in the program, and you'll forget what you did in the other parts of the program.

Pick an argument order convention and stick to it.





Argument Order Convention #2

Here's an example argument order convention:

1. all arrays in alphabetical order, and then
2. all lengths of arrays in the same order as those arrays, and then
3. all non-length scalars, in alphabetical order.

Given this convention:

- when you define a new function, you know what order to use in the function definition;
- when you call a function that you've defined, you know what order to use in the function call.





Side Effects #1

A *side effect* of a function is something that the function does other than calculate and return its return value, and that affects something other than the values of local variables.





Side Effects #2

```
int input_number_of_elements ()
{ /* input_number_of_elements */
    const int minimum_number_of_elements = 1;
    const int program_failure_code      = -1;
    int number_of_elements;

    printf("How many elements would you like ");
    printf("the array to have (at least %d)?\n",
        minimum_number_of_elements);
    scanf("%d", &number_of_elements);
    if (number_of_elements < minimum_number_of_elements) {
        printf("Too few, idiot!\n");
        exit(program_failure_code);
    } /* if (number_of_elements < ... ) */
    return number_of_elements;
} /* input_number_of_elements */
```

This function has the **side effect** of outputting a prompt message to the user, as well as of idiotproofing (i.e., outputting an error message and terminating if needed).



Side Effects Example #1

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

int main ()
{ /* main */
    const int first_element      = 1;
    const int program_success_code = 0;
    const int program_failure_code = -1;
    float* element_value = (float*)NULL;
    int    number_of_elements;
    int    index;
    int    input_number_of_elements();
```

Function prototype





Side Effects Example #2

```
number_of_elements =
    input_number_of_elements();
printf("The number of elements that you\n");
printf("  plan to input is %d.\n",
    number_of_elements);
element_value =
    (float*)malloc(sizeof(float) *
        number_of_elements);
if (element_value == (float*)NULL) {
    printf("ERROR: couldn't allocate the array\n");
    printf("  named element_value of %d elements.\n",
        number_of_elements);
    exit(program_failure_code);
} /* if (element_value == (float*)NULL) */
free(element_value);
element_value = (float*)NULL;
return program_success_code;
} /* main */
```



Side Effects Example #3

```
% cat inputnumelts.c
int input_number_of_elements ()
{ /* input_number_of_elements */
    const int minimum_number_of_elements = 1;
    const int program_failure_code      = -1;
    int number_of_elements;

    printf("How many elements would you like\n");
    printf("  the array to have (at least %d)?\n",
           minimum_number_of_elements);
    scanf("%d", &number_of_elements);
    if (number_of_elements < minimum_number_of_elements) {
        printf("Too few, idiot!\n");
        exit(program_failure_code);
    } /* if (number_of_elements < ... ) */
    return number_of_elements;
} /* input_number_of_elements */
```

```
% gcc -o userarray userarray.c inputnumelts.c
% userarray
```

```
How many elements would you like
  the array to have (at least 1)?
```

```
5
```

```
The number of elements that you plan to input is 5.
```



A Function That Doesn't Return a Value #1

```
int input_elements (float* element_value,
                   int number_of_elements)
{ /* input_elements */
    const int first_element = 0;
    int index;

    printf("What are the %d elements ",
           number_of_elements);
    printf("of the array?\n");
    for (index = first_element;
         index < number_of_elements; index++) {
        scanf("%f", &element_value[index]);
    } /* for index */
    return ???;
} /* input_elements */
```

What on earth are we going to return?





A Function That Doesn't Return a Value #2

What on earth are we going to return?

The best answer is, we're not going to return anything.

But if we're not returning anything, then what return type should the function have?

In C, we have a special data type to use as the return type of a function that doesn't return anything: void.

Thus, a void function is a function whose return type is a `void`, and which therefore returns nothing at all.





void Functions #1

A *void function* is exactly like a typical function, except that its return type is `void`, which means that it returns nothing at all.

```
void input_elements (float* element_value,
                    int number_of_elements)
{ /* input_elements */
    const int first_element = 0;
    int index;

    printf("What are the %d elements ",
           number_of_elements);
    printf("of the array?\n");
    for (index = first_element;
         index < number_of_elements; index++) {
        scanf("%f", &element_value[index]);
    } /* for index */
} /* input_elements */
```





void Functions #2

A **void function** is invoked simply by the name of the function and its arguments (e.g., in the main function):

```
input_elements(independent_variable,  
              number_of_elements);
```

Notice that a void function **must** have side effects to be useful.





void Function Call Example #1

```
#include <stdio.h>

int main ()
{ /* main */
    const int first_element          = 0;
    const int program_failure_code = -1;
    const int program_success_code = 0;
    float* element_value = (float*)NULL;
    int    number_of_elements;
    int    index;
    int    input_number_of_elements();
    void input_elements(float* element_value,
                       int number_of_elements);
```





void Function Call Example #2

```
number_of_elements =
    input_number_of_elements();
element_value =
    (float*)malloc(sizeof(float) *
                    number_of_elements);
if (element_value == (float*)NULL) {
    printf("ERROR: couldn't allocate the array\n");
    printf("  named element_value of ");
    printf("%d elements.\n", number_of_elements);
    exit(program_failure_code);
} /* if (element_value == (float*)NULL) */
```





void Function Call Example #3

```
input_elements(element_value, number_of_elements);
printf("The %d elements are:\n",
       number_of_elements);
for (index = first_element;
     index < number_of_elements; index++) {
    printf("%f ", element_value[index]);
} /* for index */
printf("\n");
free(element_value);
element_value = (float*)NULL;
return program_success_code;
} /* main */
```





void Function Call Example #4

```
% gcc -o userarray2 userarray2.c inputnumelts.c \  
    inputarrayvoidfunc.c
```

```
% userarray2
```

```
How many elements would you like  
the array to have (at least 1)?
```

```
5
```

```
What are the 5 elements of the array?
```

```
1 8 25 27 32
```

```
The 5 elements are:
```

```
1.000000 8.000000 25.000000 27.000000 32.000000
```





Why Do We Like Code Reuse?

1. **Bug avoidance**: Since we don't have to retype the function from scratch every time we use it, we aren't constantly making new and exciting typos.
2. **Implementation efficiency**: We aren't wasting valuable programming time (\$8 - \$100s per programmer per hour) on writing commonly used functions from scratch.
3. **Verification**: We can test a function under every conceivable case, so that we're confident that it works, and then we don't have to worry about whether the function has bugs when we use it in a new program.





Why Do We Like User-Defined Functions?

1. **Code Reuse**
2. **Encapsulation**: We can write a function that **packages** an important concept (e.g., the cube root). That way, we don't have to litter our program with cube root calculations. So, someone reading our program will be able to tell immediately that, for example, a particular statement has a cube root in it, rather than constantly having to figure out what `pow(x, 1.0/3.0)` means.
3. **Modular Programming**: If we make a bunch of encapsulations, then we can have our main function simply call a bunch of functions. That way, it's easy for someone reading our code to tell what's going on in the main function, and then to look at individual functions to see how they work.

