User-Defined Functions
Part 2 Outline

1. User-Defined Functions Part 2 Outline
2. Argument Order When Passing Arrays
3. Code Reuse Is GOOD GOOD GOOD
4. Reusing User-Defined Functions via the #include Directive
5. #include Directive Example #1
6. #include Directive Example #2
7. #include Directive Example #2 (continued)
8. Example #2 Without #include Directive
9. Example #2 Without #include Directive (continued)
10. Actual Arguments vs. Formal Arguments
11. Argument Order
12. Argument Order Within the Function Is Arbitrary
13. Argument Order of Function Definition MUST EXACTLY MATCH Argument Order of Function Call
14. Argument Order of Function Definition MUST EXACTLY MATCH Argument Order of Function Call (continued)
15. Argument Order Conventions: Pick One and Stick to It
16. Side Effects
17. Side Effects Example
18. A Function That Doesn’t Return a Value
19. void Functions
20. void Function Call Example
21. void Function Call Example (continued)
22. Why Do We Like Code Reuse? /
 Why Do We Like User-Defined Functions?

Argument Order When Passing Arrays

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.

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

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.

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.

Reusing User-Defined Functions via the #include Directive

In the two example programs at the end of the previous slide packet, we saw two copies of the function definition for cube_root, and these two copies were character-for-character identical.

It’d be a big waste of time to have to type the exact same function definition over and over and over if we wanted to use it multiple times, especially if our function definition was, say, 100 lines long (which is not at all unreasonable).

So, C allows us to put the function in a completely separate C source file, and to include it when we’re compiling, by using the #include directive.

The general form of the #include directive is:

    #include "filename"

For example:

    #include "cube_root.c"

What does this mean?

When the compiler encounters an #include directive, it brings the contents of the included source file into the primary source file that it’s currently compiling.

So, the upshot is that the compiler ends up treating the primary source file exactly as if you had typed the contents of the included file into it, in the place where the #include directive is.

This encourages code reuse.
```c
#include <stdio.h>
#include <math.h>

int main ()
{
    const int number_of_inputs = 3;
    float input_value1, cube_root_value1;
    float input_value2, cube_root_value2;
    float input_value3, cube_root_value3;
    float cube_root(float original_value);

    printf("What %d real numbers would you like the cube roots of?\n", number_of_inputs);
    scanf("%f %f %f", &input_value1, &input_value2, &input_value3);
    cube_root_value1 = cube_root(input_value1);
    cube_root_value2 = cube_root(input_value2);
    cube_root_value3 = cube_root(input_value3);
    printf("The cube root of %f is %f.\n", input_value1, cube_root_value1);
    printf("The cube root of %f is %f.\n", input_value2, cube_root_value2);
    printf("The cube root of %f is %f.\n", input_value3, cube_root_value3);
    return 0;
}
```

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

int main ()
{
    const int number_of_inputs = 5;
    float input_value[number_of_inputs];
    float cube_root_value[number_of_inputs];
    int index;
    float cube_root(float original_value);

    printf("What %d real numbers would you like the cube roots of?\n", number_of_inputs);
    for (index = first_input; index < number_of_inputs; index++)
    {
        scanf("%f", &input_value[index]);
    }

    for (index = first_input; index < number_of_inputs; index++)
    {
        cube_root_value[index] = cube_root(input_value[index]);
    }

    for (index = first_input; index < number_of_inputs; index++)
    {
        printf("The cube root of %f is %f.\n", input_value[index], cube_root_value[index]);
    }
    return 0;
}
```
What 5 real numbers would you like the cube roots of?
1 8 25 27 32
The cube root of 1.000000 is 1.000000.
The cube root of 8.000000 is 2.000000.
The cube root of 25.000000 is 2.924018.
The cube root of 27.000000 is 3.000000.
The cube root of 32.000000 is 3.174802.

Example #2 Without #include Directive

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

int main ()
{ /* main */
  const int first_input = 0;
  const int number_of_inputs = 5;
  float input_value[number_of_inputs];
  float cube_root_value[number_of_inputs];
  int index;

  float cube_root(float original_value);
  printf("What %d real numbers would you\n", number_of_inputs);
  printf("like the cube roots of?\n");
  for (index = first_input; index < number_of_inputs; index++)
  { /* for index */
    scanf("%f", &input_value[index]);
  } /* for index */
  for (index = first_input; index < number_of_inputs; index++)
  { /* for index */
    cube_root_value[index] =
    cube_root(input_value[index]);
  } /* for index */
  for (index = first_input; index < number_of_inputs; index++)
  { /* for index */
    printf("The cube root of %f is %f.\n", input_value[index],
    cube_root_value[index]);
  } /* for index */
  return 0;
} /* main */

float cube_root(float original_value)
{ /* cube_root */
  const float cube_root_power = 1.0 / 3.0;
  return pow(original_value, cube_root_power);
} /* cube_root */
```
% gcc -o cube_root_array cube_root_array.c -lm
% cube_root_array
What 5 real numbers would you like the cube roots of?
1 8 25 27 32
The cube root of 1.000000 is 1.000000.
The cube root of 8.000000 is 2.000000.
The cube root of 25.000000 is 2.924018.
The cube root of 27.000000 is 3.000000.
The cube root of 32.000000 is 3.174802.

Actual Arguments vs. Formal Arguments
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 original_value
- and returns the cube root of
- the value of stored in the variable input_value1.

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 Within the Function Is Arbitrary

```c
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 error_exit_value = -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(error_exit_value);
  } /* if (number_of_elements < ...) */
  sum = initial_sum;
  for (element = first_element;
       element < number_of_elements; element++)
  { /* for element */
    sum = sum + array[element];
  } /* for element */
  return sum / number_of_elements;
} /* mean */
```

```c
float mean (int number_of_elements, float* array) /* <-- NOTICE! */
{ /* mean */
  const float initial_sum = 0.0;
  const int minimum_number_of_elements = 1;
  const int first_element = 0;
  const int error_exit_value = -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(error_exit_value);
  } /* if (number_of_elements < ...) */
  sum = initial_sum;
  for (element = first_element;
       element < number_of_elements; element++)
  { /* for element */
    sum = sum + array[element];
  } /* for element */
  return sum / number_of_elements;
} /* mean */
```
Argument Order of Function Definition
MUST EXACTLY MATCH
Argument Order of Function Call

% cat meanfunctestrevarg.c
#include <stdio.h>
int main ()
{
  /* main */
  const int minimum_number_of_elements = 1;
  const int first_index = 0;
  const int error_exit_value = -1;
  float* input_value = (float*)NULL;
  float input_mean;
  int number_of_elements, index;

  float mean(int number_of_elements, float* array);
  printf("How many elements are in the input_value
");
  printf(" (at least %d)? \n", minimum_number_of_elements);
  scanf("%d", &number_of_elements);

  if (number_of_elements < minimum_number_of_elements)
    printf("Idiot! I said at least %d! \n", minimum_number_of_elements);
  exit(error_exit_value);

  input_value = (float*)malloc(sizeof(float) * number_of_elements);
  if (input_value == (float*)NULL)
    printf("ERROR: can’t allocate a float array of %d elements. \n", number_of_elements);
  exit(error_exit_value);

  printf("What are the %d elements? \n", number_of_elements);
  for (index = first_index;
      index < number_of_elements; index++)
    scanf("%f", &input_value[index]);

  input_mean = mean(input_value, number_of_elements);
  printf("The mean of the %d elements is %f. \n", number_of_elements, input_mean);
  free(input_value);
  input_value = (float*)NULL;
  return 0;
}

/* NOTICE! VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV */
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 error_exit_value = -1;
  float sum;
  int element;

  if (number_of_elements < minimum_number_of_elements)
    printf("ERROR: can’t have an array of length %d: \n", number_of_elements);
  printf(" it must have at least %d element. \n", minimum_number_of_elements);
  exit(error_exit_value);

  sum = initial_sum;
  for (element = first_element;
      element < number_of_elements; element++)
    sum = sum + array[element];

  return sum / number_of_elements;
}

/* main */
% gcc -o meanfunctestrevarg meanfunctestrevarg.c
meanfunctestrevarg.c: In function 'main':
meanfunctestrevarg.c:35: warning:
  passing arg 1 of 'mean' makes integer from pointer without a cast
meanfunctestrevarg.c:35: warning:
  passing arg 2 of 'mean' makes pointer from integer without a cast

/* NOTICE! VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV */
Argument Order Conventions: Pick One and Stick to It

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.

So, pick an argument order convention and stick to it.

Here’s an example:

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

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. For example, consider this function:

```c
int input_number_of_elements ()
{ /* input_number_of_elements */
    const int minimum_number_of_elements = 1;
    const int error_exit_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(error_exit_code);
    } /* if (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

```c
#include <stdio.h>

int main() {
    const int error_exit_value = -1;
    const int first_element = 1;
    const int number_of_elements;
    int allocate_status, index;
    int input_number_of_elements();

    number_of_elements = input_number_of_elements();
    printf("The number of elements that you plan to input is %d.
    
", number_of_elements);
    element_value = (float*)malloc(sizeof(float) * number_of_elements);
    if (element_value == (float*)NULL) {
        printf("ERROR: couldn't allocate the array named element_value of %d elements.
        
", number_of_elements);
        exit(error_exit_value);
    }
    for (index = first_element; index < number_of_elements; index++) {
        printf("What are the %d elements of the array?\n", index + 1);
        scanf("%f", &element_value[index]);
        return 0;
    }
    exit(error_exit_value);
}
```

A Function That Doesn’t Return a Value

Suppose we wanted, for example, to read values into an array of a specified length. We might define a function like so:

```c
int input_elements(  
    float* element_value, int number_of_elements)  
{  
    const int first_element = 0;
    int index;
    printf("What are the %d elements of the array?\n", number_of_elements);
    for (index = first_element; index < number_of_elements; index++) {
        scanf("%f", &element_value[index]);
    }
    return 0;
}
```

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 therefore returns nothing.
void Functions

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

```c
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 of the array?
", number_of_elements);  
  for (index = first_element;  
       index < number_of_elements; index++) {  
    scanf("%f", &element_value[index]);  
  } /* for index */  
} /* input_elements */
```

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

```c
input_elements(student_ID, number_of_students);
```

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

void Function Call Example

```c
#include <stdio.h>

int main ()
{ /* main */
  const int first_element = 0;
  const int error_exit_code = -1;
  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);
  number_of_elements = input_number_of_elements();
  void input_elements(
    float* element_value,
    int number_of_elements);
  element_value = malloc(sizeof(float) * number_of_elements);
  if (element_value == (float*)NULL) {
    printf("ERROR: couldn't allocate the array
named element_value of %d elements.
", number_of_elements);
    exit(error_exit_code);
  } /* if (element_value == (float*)NULL) */
  input_elements(element_value, number_of_elements);
  printf("The %d elements are:
", 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 0;
} /* main */

#include "inputnumelts.c"
#include "inputarrayvoidfunc.c"
```
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 (see above)

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 the meaning of $\text{pow}(x, 1.0 / 3.0)$

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.