User-Defined Functions
Part 1 Outline

1. User-Defined Functions Part 1 Outline
2. Standard Library Functions Are Not Enough
3. User-Defined Functions
4. A User-Defined Function for Mean
5. Flowchart for Mean Function
6. User-Defined Function Definitions
7. User-Defined Function Definitions: Declarations Are Valid Only Within the Function Where They Occur
8. User-Defined Function Definitions: Return Type
10. User-Defined Function Definitions: Array Arguments
11. User-Defined Function Definitions: Local Variables & Named Constants
12. User-Defined Function Definitions: Returning the Return Value
13. An Important Point About Declarations Inside Functions
14. General Form of User-Defined Function Definitions
15. Using a User-Defined Function: Example
16. Using a User-Defined Function: Example Run
17. Another User-Defined Function Example
18. Using a User-Defined Function: Example #1
19. Function Prototype Declarations
20. Actual Arguments & Formal Arguments
21. Using a User-Defined Function Example #2: Array
22. Using a User-Defined Function Example #2: Array (continued)

Standard Library Functions Are Not Enough

Often, we have a particular kind of value that we need to calculate over and over again, under a variety of circumstances.

For example, in our recent project, we had to calculate the mean, which is a common function that comes up in a lot of contexts:

```c
sum = 0.0;
for (student = first_student; student < number_of_students; student++) {
    sum += overall_percentage[student];
} /* for student */
overall_percentage_mean = sum / number_of_students;
```

We know that the mean calculation is always the same. So why should we have to write the same piece of code over and over and over and over and over?

Wouldn’t it be better if we could write that piece of code just once and then reuse it in many applications?
User-Defined Functions

So, it’d be nice to replace the code

```c
sum = 0.0;
for (student = first_student;
     student < number_of_students; student++) {
    sum += overall_percentage[student];
} /* for student */
overall_percentage_mean = sum / number_of_students;
```

with calls to a function that would calculate the mean for any array:

```c
overall_percentage_mean = 
mean(overall_percentage, number_of_students);
```

Obviously, the designers of C weren’t able to anticipate the zillion things that we might need functions to do – such as calculate the mean – so there are no standard library functions to calculate things that are somewhat more application-specific.

Instead, we as C programmers are going to have to define our own function to do it.

---

A User-Defined Function for Mean

```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 exit_return_value = -1;
    float sum;
    int element;
    if (number_of_elements <
        minimum_number_of_elements) {
        printf("ERROR: can't have an array 
of length %d:
     it must have at least %d element.
", number_of_elements);
        exit(exit_return_value);
    } /* 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 */
```

Because we as users of C define this kind of function our own personal selves, this kind of function is referred to as a user-defined function.
Flowchart for Mean Function

User-Defined Function Definitions

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 exit_return_value = -1;
    float sum;
    int element;

    if (number_of_elements < minimum_number_of_elements)
        {printf("ERROR: can't have an array ");
         printf("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(exit_return_value);
        } /* 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 */

In general, the definition of a user-defined function looks an awful lot like a program, except for the following things:

1. The function header begins with a return type that is appropriate for that function (e.g., int, float, char).
2. The function has a name that is chosen by the programmer.
3. At the end of the function header is a list of arguments, enclosed in parentheses and separated by commas, each argument preceded by its data type.
4. The function may declare local named constants and local variables.
5. In the body of the function, the return statement tells the function what value to return to the statement that called the function.
User-Defined Function Definitions:
Declarations Are Valid Only Within
the Function Where They Occur

```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 exit_return_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: 
", number_of_elements);
        printf(" it must have at least %d element.\n",
               minimum_number_of_elements);
        exit(exit_return_value);
    } /* 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 */
```

The compiler treats each function completely independently of the others.

Most importantly, the declarations inside a function — including the declarations of its arguments — apply only to that function, not to any others.

For example, the declaration of `initial_sum` in the function `mean` is visible only to the function `mean` and not to the main function or to any other function. If another function wants to have the same named constant, it must have its own declaration.

User-Defined Function Definitions:
Return Type

```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 exit_return_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: 
", number_of_elements);
        printf(" it must have at least %d element.\n",
               minimum_number_of_elements);
        exit(exit_return_value);
    } /* 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 */
```

In the function header, immediately before the name of the function is a data type.

This data type specifies the *return type*, which is the data type of the value that the function will return.

The return type (for now) must be a basic scalar type (e.g., `int`, `float`, `char`).

Notice that the return type of the function is not declared in the traditional way, but is declared nonetheless.
User-Defined Function Definitions:
List of Arguments

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  exit_return_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: it must have at least %d elements.");
        exit(exit_return_value);
    } /* if (number_of_elements < ...) */
    for (element = first_element;
         element < number_of_elements; element++) {
        sum = sum + array[element];
    } /* for element */
    return sum / number_of_elements;
} /* mean */

At the end of the function header, immediately after the function
name, is a list of arguments, enclosed in parentheses and separated
by commas, each argument preceded by its data type.

Thus, the function’s arguments are declared, but not in the function’s
declaration section.

The names of these arguments do not have to match the names of
the arguments that are passed into the function by the main function
(or by whatever other function) that calls the function. They should
be meaningful with respect to the function in which they occur,
not with respect to the other function(s) that call that function.

User-Defined Function Definitions:
Array Arguments

float mean (float* array, int number_of_elements)

When passing an array argument, you must also pass an argument
that represents the length of the array.

Not surprisingly, this argument should be of type int.

Also, when passing an array argument, you have two choices about
how to express the argument’s data type. The first is above; the
second is below.

float mean (float array[], int number_of_elements)
User-Defined Function Definitions: Local Variables & Named Constants

```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 exit_return_value = -1;
    float sum;
    int element;
    ...
} /* mean */
```

The function’s declaration section may contain, in addition to declarations for the functions arguments, declarations of local named constants and local variables.

These names that are valid ONLY within the function that is being defined.

On the other hand, these same names can be used with totally different meanings by other functions (and by the calling function).

Good programming style requires declaring:
1. local named constants, followed by
2. local variables.

User-Defined Function Definitions: Returning the Return Value

```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 exit_return_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:
", number_of_elements);
        printf(" it must have at least %d element.
", minimum_number_of_elements);
        exit(exit_return_value);
    } /* 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 */
```

In the body of the function, the return statement tells the function to return the return value.

If the function does not return a value, then the compiler may get upset.

The return value is returned to the statement that called the function, and in some sense “replaces” the function call in the expression where the function call appears.

```c
overall_percentage_mean =
    mean(overall_percentage, number_of_students);
```
An Important Point About Declarations Inside Functions

The following point is EXTREMELY important:

**For our purposes**, the only user-defined identifiers that a given function is aware of — whether it's the `main` function or otherwise — are those that are explicitly declared in the function's declaration section, or in the function's argument list.

(The above statement is not literally true, but is true enough for our purposes.)

Thus, the function is aware of:

1. its arguments, if any;
2. its local named constants, if any;
3. its local variables, if any;
4. other functions that it has declared *prototypes* for, if any (described later).

The function knows **NOTHING AT ALL** about variables or named constants declared inside any other function. It isn’t aware that they exist and cannot use them.

Therefore, the **ONLY** way to send information from one function to another is by passing arguments from the calling function to the called function.

General Form of User-Defined Function Definitions

```
returntype funcname (datatype1 arg1, datatype2 arg2, ...)
{
    // *funcname*/
    const localconst1type localconst1;
    const localconst2type localconst2;
    ...
    localvar1type localvar1;
    localvar2type localvar2;
    ...
    [function body: does stuff]
    return returnvalue;
} // *funcname*/
```
Using a User-Defined Function: Example

#include <stdio.h>

int main ()
{
    /* main */
    const int minimum_number_of_elements = 1;
    const int first_index = 0;
    const int exit_return_value = -1;
    float* input_value = (float*)NULL;
    float input_mean;
    int number_of_elements, index;

    float mean(float* array, int number_of_elements);

    printf("How many elements are in the input_value
n");
    printf(" (at least %d)?", 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(exit_return_value);
    }

    input_value = (float*)malloc(sizeof(float) * number_of_elements);
    if (input_value == (float*)NULL)
    {
        printf("ERROR: can't allocate a float array
n");
        exit(exit_return_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;
}
/* main */

Using a User-Defined Function: Example Run

#include <stdio.h>

int main ()
{
    /* main */
    const int minimum_number_of_elements = 1;
    const int first_index = 0;
    const int exit_return_value = -1;
    float* input_value = (float*)NULL;
    float input_mean;
    int number_of_elements, index;

    float mean(float* array, int number_of_elements);

    printf("How many elements are in the input_value
n");
    printf(" (at least %d)?", 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(exit_return_value);
    }

    input_value = (float*)malloc(sizeof(float) * number_of_elements);
    if (input_value == (float*)NULL)
    {
        printf("ERROR: can't allocate a float array
n");
        exit(exit_return_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;
}
/* main */
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 exit_return_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: ");
        printf("it must have at least %d element.");
        exit(exit_return_value);
    } /* 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 */

Another User-Defined Function Example

Here’s a definition of a user-defined function.

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 */

What can we say about this user-defined function?

1. Its name is cube_root.
2. Its return type is float.
3. It has one argument, original_value, whose type is float.
4. It has one local named constant, cube_root_power.
5. It has no local variables.
6. It calculates and returns the cube root of the incoming argument.

So, cube_root calculates the cube root of a float argument and returns a float result whose value is the cube root of the argument.

Notice that cube_root simply calls the C standard library function pow, using a specific value for the exponent. We say that cube_root is a wrapper around pow, or more formally that cube_root encapsulates pow.

Does the name of a user-defined function have to be meaningful?

Absolutely not; you could easily have a function named square_root that always returns 12. But that’d be REALLY REALLY DUMB, and you’d get a VERY BAD GRADE.
Using a User-Defined Function: Example #1

```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;
}

float cube_root (float original_value)
{
    const float cube_root_power = 1.0 / 3.0;
    return pow(original_value, cube_root_power);
}
```

Notice this declaration:

```c
float cube_root(float original_value);
```

This declaration is a function prototype declaration.

The function prototype declaration tells the compiler that there’s a function named `cube_root` with a return type of `float`, and that it’s declared external to (outside of) the function that’s calling the `cube_root` function.

You must declare prototypes for the functions that you’re calling. Otherwise, the compiler will assume that, by default, the function returns an `int` and has no arguments. If that turns out not to be the case (i.e., most of the time), then the compiler will become angry.

Function Prototype Declarations

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

int main ()
{
    ... float cube_root(float original_value);
    ... }
}
```

What 3 real numbers would you like the cube roots of?

1 8 25

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.
Actual Arguments & Formal Arguments

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

int main ()

    /* main */
    ...
    cube_root_value1 = cube_root(input_value1);
    cube_root_value2 = cube_root(input_value2);
    cube_root_value3 = cube_root(input_value3);
    ...

    } /* main */

float cube_root (float original_value)

    /* cube_root */
    ...

    } /* cube_root */

When we talk about the arguments of a function, we’re actually talking about two very different kinds of arguments.

The arguments that appear in the call to the function — for example, input_value1, input_value2 and input_value3 in the program fragment above — are referred to as actual arguments, because they’re the values that are actually getting passed to the function.

On the other hand, the arguments that appear in the definition of the function — for example, original_value, in the function fragment above — are referred to as formal arguments, because they’re the names that are used in the formal definition of the function.

Jargon: sometimes formal arguments are referred to as dummy arguments.

User-Defined Function Example #2: Array

% cat cube_root_array.c
#include <stdio.h>
#include <math.h>

int main ()

    /* main */
    {
        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)
        {
            /* cube_root */
            ...
        } /* cube_root */

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