User Defined Functions 1 Outline

Standard Library Not Enough #1

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

For instance, in the example program for PP#5, we have to calculate the arithmetic mean of the elements of each array.

```
sum = initial_sum;
for (element = first_element;
     element < number_of_elements; element++) {
    sum += list1_input_value[element];
} /* for element */
list1_arithmetic_mean = sum / number_of_elements;
```

Standard Library Not Enough #2

We know that the algorithm for calculating the arithmetic mean of the elements of an array 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?

Calling a Function Instead

So, it'd be nice to replace this code:

```
sum = initial_sum;
for (element = first_element;
     element < number_of_elements; element++) {
    sum += list1_input_value[element];
} /* for element */
list1_arithmetic_mean =
    sum / number_of_elements;
```

with calls to a function that would calculate the arithmetic mean for **<u>any</u>** array:

```
list1_arithmetic_mean =
    arithmetic_mean(list1_input_value, number_of_elements);
```

Why User-Defined Functions?

```
list1_arithmetic_mean =
    arithmetic_mean(list1_input_value, number_of_elements);
...
list2_arithmetic_mean =
    arithmetic_mean(list2_input_value, number_of_elements);
```

Obviously, the designers of C weren't able to anticipate the zillion things that we might need functions to do – such as calculate the arithmetic mean of the elements of an array.

So there are no standard library functions to calculate something that is ***application-specific***.

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

User-Defined `arithmetic_mean`

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_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, arithmetic_mean_value;
    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(program_failure_code);
    } /* if (number_of_elements < ...) */
    if (array == (float*)NULL) {
        printf("ERROR: can't calculate the arithmetic mean of ");
        printf("a nonexistent array.\n");
        exit(program_failure_code);
    } /* if (array == (float*)NULL) */
    sum = initial_sum;
    for (element = first_element;
         element < number_of_elements; element++) {
        sum += array[element];
    } /* for element */
    arithmetic_mean_value = sum / number_of_elements;
    return arithmetic_mean_value;
} /* arithmetic_mean */
```

User-Defined Function Properties

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

1. The function header begins with a ***return type*** that is appropriate for that function (for example, `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.

Declarations Valid in Own Function #1

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_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, arithmetic_mean_value;
    int   element;
    ...
} /* arithmetic_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.

Declarations Valid in Own Function #2

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_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, arithmetic_mean_value;
    int   element;
    ...
} /* arithmetic_mean */
```

For example:

- The declaration of `initial_sum` in the function `arithmetic_mean` is visible only to the function `arithmetic_mean` but not to the `main` function, nor to any other function.

If another function wants to have the same named constant, it has to have its own declaration of that named constant.

Return Type

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

In the function header, immediately **before** the function name 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 (for example, `int`, `float`, `char`).

Notice that the return type of the function **is declared**, but **in a weird way** (in the function header, before the function name).

List of Arguments

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_mean */
    ...
} /* arithmetic_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**.

Names of Arguments

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

The names of the arguments in the function definition **DON'T** 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 function(s) that call(s) that function.

Array Arguments

```
float arithmetic_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 length 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 arithmetic_mean (float array[], int number_of_elements)
```

In CS1313, we prefer `*` notation to `[]` notation.

Local Variables & Named Constants #1

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_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, arithmetic_mean_value;
    int   element;
    ...
} /* arithmetic_mean */
```

The function’s declaration section may contain 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).

Local Variables & Named Constants #2

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_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, arithmetic_mean_value;
    int   element;
    ...
} /* arithmetic_mean */
```

Good programming style requires declaring:

1. **local named constants**, followed by
2. **local variables**

inside the function definition.

Note that these declarations should occur in the usual order.

Returning the Return Value #1

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_mean */
    ...
    sum = initial_sum;
    for (element = first_element;
         element < number_of_elements; element++) {
        sum += array[element];
    } /* for element */
    arithmetic_mean_value = sum / number_of_elements;
    return arithmetic_mean_value;
} /* arithmetic_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.

Returning the Return Value #2

```
float arithmetic_mean (float* array, int number_of_elements)
{ /* arithmetic_mean */
    ...
    sum = initial_sum;
    for (element = first_element;
         element < number_of_elements; element++) {
        sum += array[element];
    } /* for element */
    arithmetic_mean_value = sum / number_of_elements;
    return arithmetic_mean_value;
} /* arithmetic_mean */
```

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.

```
list1_arithmetic_mean =
    arithmetic_mean(list1_input_value, number_of_elements);
...
list2_arithmetic_mean =
    arithmetic_mean(list2_input_value, number_of_elements);
```

Declarations Inside Functions #1

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 some other function – are those that are **explicitly declared in the function**'s declaration section, or in the function's argument list.

(The above statement isn't literally true, but is true enough for our purposes.)

Declarations Inside Functions #2

Thus, a 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).

Declarations Inside Functions #3

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 Function Definitions

```
returntype funcname ( datatype1 arg1, datatype2 arg2, ... )
{ /* funcname */
    const localconst1type localconst1 = localvalue1;
    const localconst2type localconst2 = localvalue2;
    ...
    localvar1type localvar1;
    localvar2type localvar2;
    ...
    [ function body: does stuff ]
    return returnvalue;
} /* funcname */
```

User-Defined Function Example #1

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

int main ()
{ /* main */
    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_arithmetic_mean;
    float  list2_arithmetic_mean;
    int    number_of_elements;
    int    element;
    float  arithmetic_mean(float* array, int number_of_elements);
```

Function prototype

User-Defined Function Example #2

```
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");
```

User-Defined Function Example #3

```
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",
        number_of_elements);
    printf(" values.\n");
    exit(program_failure_code);
} /* if (number_of_elements < minimum_ňumber_of_elements) */
```

User-Defined Function Example #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) */
```

User-Defined Function Example #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 */
```

User-Defined Function Example #6

```
list1_arithmetic_mean =
    arithmetic_mean(list1_input_value, number_of_elements);
list2_arithmetic_mean =
    arithmetic_mean(list2_input_value, number_of_elements);
```

Function calls

User-Defined Function Example #7

```
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, list1_arithmetic_mean);
printf("The arithmetic mean of the 2nd list of %d input values is %f.\n",
    number_of_elements, list2_arithmetic_mean);
```

User-Defined Function Example #8

```
    free(list2_input_value);
    list2_input_value = (float*)NULL;
    free(list1_input_value);
    list1_input_value = (float*)NULL;
    return program_success_code;
} /* main */
```

[The function definition for `arithmetic_mean`, as shown on slide #6, goes here, **AFTER** the block close for the `main` function.]

User-Defined Function Example: Run

```
% gcc -o arithmetic_mean_function arithmetic_mean_function.c -lm
% arithmetic_mean_function
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 42.2180000.
```

Another Used-Defined Function #1

```
float cube_root (float base)
{ /* cube_root */
    const float cube_root_power = 1.0 / 3.0;

    return pow(base, 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, `base`, 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.

Another Used-Defined Function #2

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

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 named constant for the exponent.

We say that `cube_root` is a ***wrapper*** around `pow`, or more formally that `cube_root` ***encapsulates*** `pow`.

Another Used-Defined Function #3

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

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

From the compiler's perspective, absolutely not; you could easily have a function named `square_root` that always returns 12.

But from the perspective of programmers, that'd be a **REALLY REALLY BAD IDEA**, and you'd get a **VERY BAD GRADE**.

Another Function Example #1

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

int main ()
{ /* main */
    const int number_of_elements     = 3;
    const int program_success_code = 0;
    float input_value1, cube_root_value1;
    float input_value2, cube_root_value2;
    float input_value3, cube_root_value3;
    float cube_root(float base);

    printf("What %d real numbers would you\n",
        number_of_elements);
    printf("  like the cube roots of?\n");
    scanf("%f %f %f",
        &input_value1, &input_value2,
        &input_value3);
```

Function prototype

Another Function Example #2

```
    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 program_success_code;
} /* main */
```

Function calls

Another Function Example #3

```
% gcc -o cube_root_scalar \
         cube_root_scalar.c cube_root.c -lm
% cube_root_scalar
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.
```

Function Prototype Declarations #1

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

int main ()
{ /* main */
    const int number_of_elements     = 3;
    const int program_success_code = 0;
    float input_value1, cube_root_value1;
    float input_value2, cube_root_value2;
    float input_value3, cube_root_value3;
    float cube_root(float base);
    ...
} /* main */
```

Notice this declaration:

```
float cube_root(float base);
```

This declaration is a ***function prototype*** declaration.

Function Prototype Declarations #2

```
float cube_root(float base);
```

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 (that is, most of the time), then the compiler will become **ANGRY**.

Actual Arguments & Formal Arguments

```
#include <stdio.h>
#include <stdlib.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 base)
{ /* cube_root */
    ...
} /* cube_root */
```

When we talk about the arguments of a function, we're actually talking about **two very different kinds** of arguments: ***actual arguments*** and ***formal arguments***.

Actual 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 */
```

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 known as ***actual arguments***, because they're the values that **actually** get passed to the function.

Mnemonic: The aCtual arguments are in the function Call.

Formal Arguments

```
float cube_root (float base)
{ /* cube_root */
    ...
} /* cube_root */
```

The arguments that appear in the **definition** of the function – for example, `base`, in the function fragment above – are known as ***formal arguments***, because they're the names that are used in the **formal definition** of the function.

Jargon: **Formal arguments** are also known as ***dummy arguments***.

Mnemonic: The **F**ormal arguments are in the function de**F**inition.

Yet Another Function Example #1

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

int main ()
{ /* main */
    const int first_element        = 0;
    const int number_of_elements   = 5;
    const int program_success_code = 0;
    float input_value[number_of_elements];
    float cube_root_value[number_of_elements];
    int   element;
    float cube_root(float base);

    printf("What %d real numbers would you\n",
        number_of_elements);
    printf("  like the cube roots of?\n");
    for (element = first_element;
         element < number_of_elements; element++) {
        scanf("%f", &input_value[element]);
    } /* for element */
```

Yet Another Function Example #2

```
    for (element = first_element;
         element < number_of_elements; element++) {
        cube_root_value[element] =
            cube_root(input_value[element]);
    } /* for element */
    for (element = first_element;
         element < number_of_elements; element++) {
        printf("The cube root of %f is %f.\n",
            input_value[element],
            cube_root_value[element]);
    } /* for element */
    return program_success_code;
} /* main */
```

Yet Another Function Example #3

```
% gcc -o cube_root_array \
        cube_root_array.c cube_root.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.
```