Standard Library Functions Outline

1. Standard Library Functions Outline
2. Functions in Mathematics
3. Functions in C
4. A Quick Look at `abs`
5. Standard Library Functions in C
6. Math: Domain & Range
7. Programming: Argument Type
8. Programming: Return Type
9. More on Function Arguments
10. Function Argument Example
11. Using Functions
12. Function Use Example
13. Evaluation of Functions in Expressions
14. Functions with Multiple Arguments
Functions in Mathematics

“In mathematics a function is a rule that assigns to each element of a set an element of the same or of another set.” (Bers & Karal, *Calculus*, 2nd ed, Holt, Reinhart & Winston, 1976, p. 50.)

So, for example, if we have a function

\[ f(x) = x + 1 \]

then we know that

\[ f(\text{-}2.5) = -2.5 + 1 = -1.5 \]
\[ f(\text{-}2) = -2 + 1 = -1 \]
\[ f(\text{-}1) = -1 + 1 = 0 \]
\[ f(0) = 0 + 1 = +1 \]
\[ f(1) = +1 + 1 = +2 \]
\[ f(2) = +2 + 1 = +3 \]
\[ f(2.5) = +2.5 + 1 = +3.5 \]

Likewise, if we have a function

\[ a(y) = |y| \]

then we know that

\[ a(\text{-}2.5) = |\text{-}2.5| = 2.5 \]
\[ a(\text{-}2) = |-2| = 2 \]
\[ a(\text{-}1) = |-1| = 1 \]
\[ a(0) = |0| = 0 \]
\[ a(1) = |1| = 1 \]
\[ a(2) = |2| = 2 \]
\[ a(2.5) = |2.5| = 2.5 \]

We refer to the thing inside the parentheses – in the first example it’d be \( x \), and in the second example \( y \) – as the argument (or sometimes the parameter) of the function.
Functions in C

In my_number.c, we saw this:

```c
...  
else if (abs(users_number - computers_number) <= 
    close_distance) {  
    printf("Close, but no cigar.\n");  
} /* if (abs(...)) <= close_distance) */  
...
```

So, what does `abs` do?

The `abs` function calculates the *absolute value* of its argument. It is the C analogue of the mathematical function

\[
a(y) = |y|
\]

(the absolute value function) that we just looked at. So:

```c
...  
fabs(-2.5) → 2.5  
abs(-2) → 2  
abs(-1) → 1  
abs(0) → 0  
abs(1) → 1  
abs(2) → 2  
fabs(2.5) → 2.5  
...
```

Note: in this example, → denotes “evaluates to,” or, in computing jargon, “returns.” We say “`abs` of -2 evaluates to 2” or “`abs` of -2 returns 2.”

Note that the function named `abs` calculates the absolute value of an `int` argument, and `fabs` calculates the absolute value of a `float` argument.
A Quick Look at \texttt{abs}

\% \texttt{cat abstest.c}
\#include <stdio.h>

\begin{verbatim}
int main ()
{ /* main */
    printf("fabs(-2.5) = %f\n", fabs(-2.5));
    printf(" abs(-2) = %d\n", abs(-2));
    printf(" abs(-1) = %d\n", abs(-1));
    printf(" abs( 0) = %d\n", abs( 0));
    printf(" abs( 1) = %d\n", abs( 1));
    printf(" abs( 2) = %d\n", abs( 2));
    printf("fabs( 2.5) = %f\n", fabs( 2.5));
    return 0;
}
\} /* main */
\% gcc -o abstest abstest.c
\% abstest
fabs(-2.5) = 2.500000
 abs(-2) = 2
 abs(-1) = 1
 abs( 0) = 0
 abs( 1) = 1
 abs( 2) = 2
fabs( 2.5) = 2.500000
\end{verbatim}

\textbf{Jargon}: in programming, the use of a function in an expression is referred to as an \textit{invocation}, or more colloquially as a \textit{call}. We say that the statement

\begin{verbatim}
    printf("%d\n", abs(-2));
\end{verbatim}

invokes or calls the function \texttt{abs}; the statement passes an argument of -2 to the function; the function \texttt{abs} returns a value of 2.

An important distinction between a mathematical function and a C function: a mathematical function is simply a \textbf{definition}, while a C function \textbf{does stuff}. More on this presently.
Standard Library Functions in C

Every implementation of C comes with a library of predefined functions. Most of these functions are common to all versions of C, and we therefore refer to them as the C Standard Library. A few examples:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Math Name</th>
<th>Value</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs (x)</td>
<td>absolute value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt (x)</td>
<td>square root</td>
<td>$x^{1/2}$</td>
<td>sqrt (2.0) → 1.414...</td>
</tr>
<tr>
<td>exp (x)</td>
<td>exponential</td>
<td>$e^x$</td>
<td>exp (1.0) → 2.718...</td>
</tr>
<tr>
<td>log (x)</td>
<td>natural logarithm</td>
<td>ln $x$</td>
<td>exp (2.718...) → 1.0</td>
</tr>
<tr>
<td>log10 (x)</td>
<td>common logarithm</td>
<td>log $x$</td>
<td>log10 (100.0) → 2.0</td>
</tr>
<tr>
<td>sin (x)</td>
<td>sine</td>
<td>sin $x$</td>
<td>sin (3.14...) → 0.0</td>
</tr>
<tr>
<td>cos (x)</td>
<td>cosine</td>
<td>cos $x$</td>
<td>cos (3.14...) → -1.0</td>
</tr>
<tr>
<td>tan (x)</td>
<td>tangent</td>
<td>tan $x$</td>
<td>tan (3.14...) → 0.0</td>
</tr>
<tr>
<td>ceil (x)</td>
<td>least integer ≥ $x$</td>
<td>$\lceil x \rceil$</td>
<td>ceil (2.5) → 3</td>
</tr>
<tr>
<td>floor (x)</td>
<td>greatest integer ≤ $x$</td>
<td>$\lfloor x \rfloor$</td>
<td>floor (2.5) → 2</td>
</tr>
</tbody>
</table>

As it turns out, the set of standard library functions is grossly insufficient for most real world tasks, so in C, and in most programming languages, there are ways for programmers to develop their own user-defined functions, which we’ll learn more about in a future lesson.
Math: Domain & Range

In mathematics, we refer to the set of numbers that can be the argument of a given function as the domain of that function.

Similarly, we refer to the set of numbers that can be the result of a given function as the range of that function.

For example, in the case of the function

\[ f(x) = x + 1 \]

we define the domain to be the set of real numbers (sometimes denoted \( \mathbb{R} \)), which means that the \( x \) in \( f(x) \) can be any real number. Likewise, we also define the range to be the set of real numbers, because for every real number \( y \) there is some real number \( x \) such that \( f(x) = y \).

On the other hand, for a function

\[ q(x) = \frac{1}{x - 1} \]

the domain cannot include 1, because

\[ q(1) = \frac{1}{1 - 1} = \frac{1}{0} \]

which is undefined. So the domain might be \( \mathbb{R} - \{1\} \) (the set of all real numbers except 1).

In that case, the range of \( q \) would be the set of all real numbers except 0, because there’s no real number \( y \) such that \( 1/y \) is 0.

(Note: if you’ve taken calculus, you’ve seen that, as \( y \) gets arbitrarily large, \( 1/y \) approaches 0 as a limit – but “gets arbitrarily large” is not a real number, and neither is “approaches 0 as a limit.”)
Programming: Argument Type

C has analogous concepts to the mathematical domain and range: *argument type* and *return type*.

The *argument type*, not surprisingly, is the data type that C expects for an argument of a particular standard library function.

Some implementations of C will not check for you whether the data type of the argument you pass is correct. If you pass the wrong data type, you can get a bogus answer:

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

int main ()
{ /* main */
   printf(" abs(-2) = %d\n", abs(-2));
   printf(" abs(-2.5) = %f\n", abs(-2.5));
   printf("fabs(-2) = %f\n", fabs(-2));
   printf("fabs(-2.5) = %f\n", fabs(-2.5));
   return 0;
} /* main */
%
% gcc -o absfabs absfabs.c
% absfabs
    abs(-2) = 2
  abs(-2.5) = -1.994530
    fabs(-2) = 2.000000
  fabs(-2.5) = 2.500000
```

This problem is more likely to come up when you pass a *float* where the function expects an *int*. In the reverse case, typically C simply promotes the *int* to a *float*. 
Programming: Return Type

Just as the programming concept of argument type is analogous to the mathematical concept of domain, so too is the programming concept of return type analogous to the mathematical concept of range.

The *return type* of a C function is the data type of the value that the function returns. The return value is **guaranteed** to have that data type, and the compiler gets upset if you use the return value inappropriately:

```c
#include <stdio.h>

int main ()
{
    /* main */
    printf(" abs(-2) = %d
", abs(-2));
    printf(" abs(-2.5) = %f\n", abs(-2.5));
    printf("fabs(-2) = %f\n", fabs(-2));
    printf("fabs(-2.5) = %f\n", fabs(-2.5));
    return 0;
}
/* main */
```

```
cat absfabs.c
#include <stdio.h>

int main ()
{
    /* main */
    printf(" abs(-2) = %d\n", abs(-2));
    printf(" abs(-2.5) = %f\n", abs(-2.5));
    printf("fabs(-2) = %f\n", fabs(-2));
    printf("fabs(-2.5) = %f\n", fabs(-2.5));
    return 0;
}
/* main */
```

```
cat absfabs.c
#include <stdio.h>

int main ()
{
    /* main */
    printf(" abs(-2) = %d
", abs(-2));
    printf(" abs(-2.5) = %f\n", abs(-2.5));
    printf("fabs(-2) = %f\n", fabs(-2));
    printf("fabs(-2.5) = %f\n", fabs(-2.5));
    return 0;
}
/* main */
```
More on Function Arguments

In mathematics, a function argument can be:

- a number:
  \[
  f(5) = 5 + 1 = 6
  \]

- a variable:
  \[
  f(z) = z + 1
  \]

- an arithmetic expression:
  \[
  f(5 + 7) = (5 + 7) + 1 = 12 + 1 = 13
  \]

- another function:
  \[
  f(a(w)) = |w| + 1
  \]

- any combination of these; i.e., any general expression whose value is in the domain of the function:
  \[
  f(3a(5w + 7)) = 3(|5w + 7|) + 1
  \]

Likewise, in C the argument of a function can be any non-empty expression that evaluates to an appropriate data type, including an expression containing a function call.
```c
#include <stdio.h>
#include <math.h>

int main ()
{
    const float pi = 3.1415926;
    float angle_in_radians;

    printf("cos(%10.7f) = %10.7f\n",
            1.5707963, cos(1.5707963));
    printf("cos(%10.7f) = %10.7f\n", pi, cos(pi));
    printf("Enter an angle in radians:\n");
    scanf("%f", &angle_in_radians);
    printf("cos(%10.7f) = %10.7f\n",
            angle_in_radians, cos(angle_in_radians));
    printf("fabs(cos(%10.7f)) = %10.7f\n",
            angle_in_radians, fabs(cos(angle_in_radians)));
    printf("cos(fabs(%10.7f)) = %10.7f\n",
            angle_in_radians, cos(fabs(angle_in_radians)));
    printf("fabs(cos(2.0 * %10.7f)) = %10.7f\n",
            angle_in_radians,
            fabs(cos(2.0 * angle_in_radians)));
    printf("fabs(2.0 * cos(%10.7f)) = %10.7f\n",
            angle_in_radians,
            fabs(2.0 * cos(angle_in_radians)));
    printf("fabs(2.0 * cos(1.0 / 5.0 * %10.7f)) = %10.7f\n",
            angle_in_radians,
            fabs(2.0 * cos(1.0 / 5.0 * angle_in_radians)));
    return 0;
}
```

Using Functions

Functions are used in expressions in exactly the same ways that variables and constants are used. For example, a function call can be used on the right hand side of an assignment:

```c
float theta = 3.1415926 / 4.0;
float cos_theta;
cos_theta = cos(theta);
```

A function call can also be used in an expression in list-directed output:

```c
printf("%f
", 2.0);
printf("%f
", pow(cos(theta), 2.0));
```

And, since any expression can be used as some function’s argument, a function call can also be used as an argument to another function:

```c
const float pi = 3.1415926;
printf("%f
",
    1 + cos(asin(sqrt(2.0)/2.0) + pi));
```

Most function calls can be used in initialization, as long as its arguments already have been initialized:

```c
float cos_theta = cos(3.1415926);
```

Similarly, most function calls can be used in named constant declaration:

```c
const float cos_theta = cos(3.1415926);
```
# Function Use Example

```c
% cat funcuse.c
#include <stdio.h>
#include <math.h>

int main () {
    /* main */
    const float pi = 3.1415926;
    const float cos_pi = cos(3.1415926);
    const float sin_pi = sin(pi);
    float phi = 3.1415926 / 4.0;
    float cos_phi = cos(phi);
    float theta, sin_theta;
    theta = 3.0 * pi / 4;
    sin_theta = sin(theta);
    printf("2.0 = %f\n", 2.0);
    printf("pi = %f\n", pi);
    printf("theta = %f\n", theta);
    printf("cos(pi) = %f\n", cos(pi));
    printf("cos_pi = %f\n", cos_pi);
    printf("sin(pi) = %f\n", sin(pi));
    printf("sin_pi = %f\n", sin_pi);
    printf("sin(theta) = %f\n", sin(theta));
    printf("sin_theta = %f\n", sin_theta);
    printf("sin(theta)^(1.0/3.0) = %f\n", 
            pow(sin(theta), (1.0/3.0)));
    printf("1 + sin(acos(1.0)) = %f\n", 
            1 + sin(acos(1.0)));
    printf("sin(acos(1.0)) = %f\n", sin(acos(1.0)));
    printf("sqrt(2.0) = %f\n", sqrt(2.0));
    printf("sqrt(2.0) / 2 = %f\n", sqrt(2.0) / 2);
    printf("acos(sqrt(2.0)/2.0) = %f\n", 
            acos(sqrt(2.0)/2.0));
    printf("sin(acos(sqrt(2.0)/2.0)) = %f\n", 
            sin(acos(sqrt(2.0)/2.0)));
    return 0;
} /* main */
```

% gcc -o funcuse funcuse.c -lm
% funcuse

2.0 = 2.000000
pi = 3.141593
theta = 2.356194
cos(pi) = -1.000000
cos_pi = -1.000000
sin(pi) = 0.000000
sin_pi = 0.000000
sin(theta) = 0.707107
sin_theta = 0.707107
sin(theta)^(1.0/3.0) = 0.890899
1 + sin(acos(1.0)) = 1.000000
sin(acos(1.0)) = 0.000000
sqrt(2.0) = 1.414214
sqrt(2.0) / 2 = 0.707107
acos(sqrt(2.0)/2.0) = 0.785398
sin(acos(sqrt(2.0)/2.0)) = 0.707107
Evaluation of Functions in Expressions

When a function call appears in an expression – for example, on the right hand side of an assignment statement – the function is evaluated just before its value is needed, in accordance with the rules of precedence order.

For example, if $x$ and $y$ are float variables, and $y$ has already been assigned the value -10.0, then the assignment statement

$$x = 1 + 2.0 \times 8.0 + \text{fabs}(y) / 4.0$$

is evaluated like so:

$$x = 1 + 2.0 \times 8.0 + \text{fabs}(y) / 4.0 \Rightarrow$$

$$x = 1 + 16.0 + \text{fabs}(y) / 4.0 \Rightarrow$$

$$x = 1 + 16.0 + \text{fabs}(-10.0) / 4.0 \Rightarrow$$

$$x = 1 + 16.0 + 10.0 / 4.0 \Rightarrow$$

$$x = 1 + 16.0 + 2.5 \Rightarrow$$

$$x = 1.0 + 16.0 + 2.5 \Rightarrow$$

$$x = 17.0 + 2.5 \Rightarrow$$

$$x = 19.5$$

So, the variable $x$ is ultimately assigned the value 19.5.
Functions with Multiple Arguments

In mathematics, we sometimes have functions that have multiple arguments:

\[ h(x, y) = xy + 2x + 3y + 5 \]

In this case, we know that

\[
\begin{align*}
    h(-2.5, -1.5) &= (-2.5)(-1.5) + (2)(-2.5) + (3)(-1.5) + 5 = -0.75 \\
    h(-2, -0.5) &= (-2)(-0.5) + (2)(-2) + (3)(-0.5) + 5 = +0.5 \\
    h(-1, 1.25) &= (-1)(1.25) + (2)(-1) + (3)(1.25) + 5 = 0 \\
    h(0, 0) &= (0)(0) + (2)(0) + (3)(0) + 5 = +5 
\end{align*}
\]

Here, we define the domain of the first argument of the function \( h \) to be the set of all real numbers \( \mathbb{R} \), and likewise we define the domain of the second argument of \( h \) to be the set of all real numbers \( \mathbb{R} \), so the domain of \( h \) as a whole is \( \mathbb{R} \times \mathbb{R} \), pronounced “real times real.”

Since the result of \( h \) is a single real value, we define the range of \( h \) to be \( \mathbb{R} \), and we denote the mapping as

\[ h : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \]

Similarly, in C we have standard library functions with multiple arguments. Examples include:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Math Name</th>
<th>Math Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow(x, y)</td>
<td>Power</td>
<td>( x^y )</td>
</tr>
</tbody>
</table>

Functions with multiple arguments can be used in exactly the same ways as functions with a single argument.

In a function call, the list of arguments must be in exactly the correct order and have exactly the correct data types.