Functions in Mathematics #1

“A **rule** that relates two variables, typically $x$ and $y$, is called a **function** if to each value of $x$ the rule assigns one and only one value of $y$.”

http://www.themathpage.com/aPreCalc/functions.htm

So, for example, if we have a function

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

then we know that

- $f(-2.5) = -2.5 + 1 = -1.5$
- $f(-2) = -2 + 1 = -1$
- $f(-1) = -1 + 1 = 0$
- $f(0) = 0 + 1 = +1$
- $f(+1) = +1 + 1 = +2$
- $f(+2) = +2 + 1 = +3$
- $f(+2.5) = +2.5 + 1 = +3.5$

...
Functions in Mathematics #2

For example, if we have a function

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

then we know that

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

...
Likewise, if we have a function

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

then we know that

\begin{align*}
  a(-2.5) &= | -2.5 | = +2.5 \\
  a(-2) &= | -2 | = +2 \\
  a(-1) &= | -1 | = +1 \\
  a(0) &= | 0 | = 0 \\
  a(+1) &= | +1 | = +1 \\
  a(+2) &= | +2 | = +2 \\
  a(+2.5) &= | +2.5 | = +2.5 \\
\end{align*}
Function Argument

\[ f(x) = x + 1 \]
\[ a(y) = |y| \]

We refer to the thing inside the parentheses immediately after the name of the function as the **argument** (also known as the **parameter**) of the function.

In the examples above:

- the argument of the function named \( f \) is \( x \);
- the argument of the function named \( a \) is \( y \).
Absolute Value Function in C #1

In my_number.c, we saw this:
...
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’s the C analogue of the mathematical function
\[ a(y) = |y| \]
(the absolute value function) that we just looked at.
Absolute Value Function in C #2

fabs(-2.5) returns 2.5
abs(-2) returns 2
abs(-1) returns 1
abs(0) returns 0
abs(1) returns 1
abs(2) returns 2
fabs(2.5) returns 2.5
Absolute Value Function in C #3

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 abs

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

int main ()
{ /* main */
    const int program_success_code = 0;
    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 program_success_code;
} /* main */
```

% gcc -o abs_test abs_test.c -lm

% abs_test
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
Function Call in Programming

**Jargon**: In programming, the use of a function in an expression is referred to as an **invocation**, or more colloquially as a **call**.

We say that the statement

```
printf("%d\n", abs(-2));
```

- **invokes** or **calls** the function **abs**;
- the statement **passes** an argument of -2 to the function;
- the function **abs** **returns** a value of 2.
Math Function vs Programming Function

An important distinction between a function in mathematics and a function in programming:

A function in mathematics is simply a definition ("this name means that expression"), while a function in programming is an action ("this name means execute that sequence of statements").

More on this later.
Every implementation of C comes with a standard *library* of predefined functions.

Note that, in programming, a *library* is a **collection of functions**.

The functions that are common to all versions of C are known as the *C Standard Library*.

On the next slide are examples of just a few of the functions in the C standard library.
C Standard Math Library Function Examples

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Math Name</th>
<th>Value</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
<td></td>
<td></td>
<td>abs(-1)</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root</td>
<td>$x^{0.5}$</td>
<td></td>
<td>sqrt(2.0)</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential</td>
<td>$e^x$</td>
<td></td>
<td>exp(1.0)</td>
</tr>
<tr>
<td>log(x)</td>
<td>natural logarithm</td>
<td>ln $x$</td>
<td></td>
<td>log(2.718…)</td>
</tr>
<tr>
<td>log10(x)</td>
<td>common logarithm</td>
<td>log $x$</td>
<td></td>
<td>log10(100.0)</td>
</tr>
<tr>
<td>sin(x)</td>
<td>sine</td>
<td>sin $x$</td>
<td></td>
<td>sin(3.14…)</td>
</tr>
<tr>
<td>cos(x)</td>
<td>cosine</td>
<td>cos $x$</td>
<td></td>
<td>cos(3.14…)</td>
</tr>
<tr>
<td>tan(x)</td>
<td>tangent</td>
<td>tan $x$</td>
<td></td>
<td>tan(3.14…)</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>ceiling</td>
<td>⌈$x$⌉</td>
<td></td>
<td>ceil(2.5)</td>
</tr>
<tr>
<td>floor(x)</td>
<td>floor</td>
<td>⌊$x$⌋</td>
<td></td>
<td>floor(2.5)</td>
</tr>
</tbody>
</table>
Is the Standard Library Enough?

It turns out that the set of C 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**. We’ll learn more about user-defined functions in a future lesson.
Math: Domain & Range #1

In **mathematics**:  
- The **domain** of a function is the set of numbers that can be used for the **argument(s)** of that function.  
- The **range** is the set of numbers that can be the **result** of that function.
Math: Domain & Range #2

For example, in the case of the function

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

we can define the **domain** of the function \( f \) to be the set of real numbers (sometimes denoted \( \mathbb{R} \)), which means that the \( x \) in \( f(x) \) can be any real number.

Similarly, we define the **range** of the function \( f \) to be the set of real numbers, because for every real number \( y \) there is some real number \( x \) such that \( f(x) = y \).
Math: Domain & Range #3

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 infinity (in the limit).

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 \( \mathbb{R} - \{0\} \) (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

Programming has concepts that are analogous to the mathematical **domain** and **range**: **argument type** and **return type**.

For a given function in C, the **argument type** — which corresponds to the **domain** in mathematics — is the data type that C expects for an argument of that function.

For example:

- the argument type of `abs` is `int`;
- the argument type of `fabs` is `float`. 
Argument Type Mismatch

An **argument type mismatch** is when you pass an argument of a particular data type to a function that expects a different data type for that argument.

Some implementations of C **WON’T** check for you whether the data type of the argument you pass is correct. So if you pass the wrong data type, you can get a bogus answer.

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 the programming concept of return type is analogous to the mathematical concept of range.

The return type of a C function – which corresponds to the range in mathematics – 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 – or you get a bogus result – if you use the return value inappropriately.
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 ()
{
    /* main */
    const float pi = 3.1415926;
    const int program_success_code = 0;
    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)));
Function Argument Example Part 2

```c
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 * ");
printf("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 program_success_code;
}
```
Function Argument Example Part 3

```bash
% gcc -o function_arguments function_arguments.c -lm
% function_arguments

\[ \cos(1.5707963) = 0.0000000 \]
\[ \cos(3.1415925) = -1.0000000 \]
Enter an angle in radians:

\[-3.1415925\]
\[ \cos(-3.1415925) = -1.0000000 \]
\[ \text{fabs}(\cos(-3.1415925)) = 1.0000000 \]
\[ \cos(\text{fabs}(-3.1415925)) = -1.0000000 \]
\[ \text{fabs}(\cos(2.0 \times -3.1415925)) = 1.0000000 \]
\[ \text{fabs}(2.0 \times \cos(-3.1415925)) = 2.0000000 \]
\[ \text{fabs}(2.0 \times \cos(1.0 / 5.0 \times -3.1415925)) = 1.6180340 \]
```
Using the C Standard Math Library

If you’re going to use functions like \texttt{cos} that are from the part of the C standard library that has to do with math, then you need to do two things:

1. In your source code, immediately below the \texttt{#include <stdio.h>}
   you must also put
   \texttt{#include <math.h>}

2. When you compile, you must append \texttt{-lm} to the end of your compile command:
   \texttt{gcc -o function_arguments function_arguments.c -lm}

(Note that this is \texttt{hyphen ell em}, NOT \texttt{hyphen one em}.)

\textbf{NOTE:} \texttt{-lm} means “link to the standard math library.”
Function Call in Assignment

Function calls 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 side of an assignment or initialization:

```c
float theta = 3.1415926 / 4.0;
float cos_theta;
...

    cos_theta = cos(theta);

length_of_c_for_any_triangle = sqrt(a * a + b * b -
    2 * a * b * cos(theta));
```
Function Call in `printf`

A function call can also be used in an expression in a `printf` statement:

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

In CS1313, we discourage this usage, because we prefer that all calculations get done in the calculation subsection, not in the output subsection.

But the C language permits this usage.
Function Call as Argument

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;
float complicated_expression;
...
complicated_expression =
    1.0 + cos(asin(sqrt(2.0)/2.0) + pi));
```

```
Function Call in Initialization

Most function calls can be used in **initialization**, as long as its arguments are literal constants:

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

This is true both in **variable initialization** and in **named constant initialization**:

```c
const float cos_pi = cos(3.1415926);
```
Function Use Example Part 1

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

int main ()
{
    const float pi = 3.1415926;
    const float cos_pi = cos(3.1415926);
    const float sin_pi = sin(pi);
    const int program_success_code = 0;
    float phi = 3.1415926 / 4.0;
    float cos_phi = cos(phi);
    float theta, sin_theta;
```
Function Use Example Part 2

```c
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)));
```
Function Use Example Part 3

```c
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 program_success_code;
} /* main */
```
Function Use Example Part 4

```bash
% gcc -o function_use function_use.c -lm
% function_use
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.
Evaluation Example #1

For example, suppose that \( x \) and \( y \) are float variables, and that \( y \) has already been assigned the value \(-10.0\).

Consider this assignment statement:

\[
x = 1 + 2.0 \times 8.0 + \text{fabs}(y) / 4.0;
\]
Evaluation Example #2

\[
x = 1 + 2.0 \times 8.0 + \text{fabs}(y) \div 4.0;
\]

\[
x = 1 + 16.0 + \text{fabs}(y) \div 4.0;
\]

\[
x = 1 + 16.0 + \text{fabs}(-10.0) \div 4.0;
\]

\[
x = 1 + 16.0 + 10.0 \div 4.0;
\]

\[
x = 1 + 16.0 + 2.5;
\]

\[
x = 1.0 + 16.0 + 2.5;
\]

\[
x = 17.0 + 2.5;
\]

\[
x = 19.5;
\]