Lecture 13 — Functions

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 6, 2016

Acknowledgments: D.W. Harder, W.D. Bishop

Mathematical Functions

You are no doubt familiar with the concept of a mathematical function: a relation between a set of inputs and output.

Mathematical functions take arguments in specific domains and return a value in another domain.

Consider $sin : R \rightarrow R$.
Given a right-angled triangle with an angle θ, return the ratio of the length of the opposite side to the length of the hypotenuse.

It takes a real number input (in R) and returns a real number output.
Each input is related to exactly one output.

Then there is evaluation of a function, such as $f(x) = x^2$.

x is the input to this function, and the function returns some output, like $f(-5) = 25$.

We often graph (on an x-y plane) the output of a function: $y = f(x)$.

Mathematical Functions

You've used mathematical functions a lot over the years:

Trigonometric functions: $cos(x), sin(x), tan(x)$
Absolute value function: $|x|$
Square root function: $\sqrt{x}$

These examples all take a single input, but there's no reason this has to be the case. Imagine a function $g(x, y) = x^y$, for example.

Functions in Programs

In a program, a function is much like a mathematical function.

A function takes 0 or more inputs and returns 0 or 1 values.

A function is sometimes also called a procedure or subroutine.

Assume for now that there is a defined function in the system for the absolute value function, $|x|$.

To make use of this function, we need to know its signature.

The function signature tells us the name, what type(s) the input to the function is, and the type of the output.

The format is: *returnType name* (*formalParameters...*)

We may have zero or more formal parameters: a placeholder for whatever value will be given as input.

The *formalParameters* is a list of variables in the form *type name1, type name2, ... type nameN.*

In $f(x)$, we state that x is a formal parameter: it's a variable that we use in defining how $f(x)$ works.

The abs Function

Suppose this is the function signature: `double abs ( double x )`

Let's look at it from left to right:

The return type is `double`: the output will be a `double`.

The name of the function is `abs` (for **abs**olute value).

The formal parameters (inputs) are in the ( ) brackets. There is one input, a `double` named `x`.

Calling a Function

Now that we know the function signature, we know how to use it.

Function `abs` expects the input of a `double` & will output a `double`.

We are now ready to call (or invoke) this function.

To do so, we use the name `abs` followed by `( )` brackets.
Inside those brackets, we put the actual parameters.

```
double absX = abs( -9.5 );
```

The output is the return value, which is then stored in `absX`.

Actual Parameters

We saw already the formal parameters for `abs` is one variable: `double x`.

The actual parameters are the values actually given as input to the function at the time we want the function to execute.

We must provide one actual parameter value for each of the formal parameter values in the signature.

The types must match (therefore order matters).

In the case of `abs`, we provide one actual parameter of type `double`; in this case, the literal `-9.5`.

Call Me Maybe?

We can also use a variable as an actual parameter:

```
double absY = abs( y );
```

Variable `absY` will hold the absolute value of whatever `y` contained at the time the function was called.

Same thing with expressions:

```
double absZ = abs( y + 7 )
```

Type promotion rules apply, so if `y` is an `int`, the actual parameter will be promoted to `double` before it is used in the function call.

Using vs. Defining a Function

Thus far it was assumed that there exists the abs function.
  i.e., someone else wrote it.

We now understand how to use an already-existing function.

Most of the time, however, we must define a function if we want to use it, since it won't already exist for us.

But: don't re-invent the wheel. If there already is an implementation, it is usually better to use that than to write your own.

A function definition has two parts: the signature and the body.

```
double abs ( double x )
{
    // TODO: Implement Body
}
```

We've already examined the signature; the first line of the definition.

The body is some block of statements that we have to write to make the function do something useful.

The `return` Keyword

There's a keyword needed to implement a function: `return`.
The `return` keyword is followed by the return value.

The return value is the output of the function.

The `return` statement could appear anywhere within the function (but commonly it's at the end).

When a `return` statement is encountered, the return value becomes the function output and the execution of the function ends.

In this respect, `return` is a bit like the `break` statement.

The abs Function Implemented

```
double abs ( double x )
{
    if ( x >= 0 )
    {
        return x;
    } else {
        return -x;
    }
}
```

In the definition, we use the formal parameter names.

(In $f(x)$: the formal parameter x is used to define the function (x^2).)

When we call `double y = abs( -5.0 );`, what happens?

Much like when we evaluate an expression:

```
a = b + 7;
```

before we can do the assignment we have to evaluate the right side.

When we evaluate a function call, we go to the function definition and execute the lines of code there.

```
double abs ( double x )
{
    if ( x >= 0 )
    {
        return x;
    }  else  {
        return -x;
    }
}
```

If abs was called with the actual parameter of `-5.0`, the value of `x` is `-5.0` and that is the value used in expressions within the function.

Here, the statement `return -x;` executes, which evaluates to `return - (-5.0)` $\rightarrow$ `return 5.0;`

Because we encountered a `return` statement, execution of the function is finished and we take the output of 5.0.

Our original assignment statement was `double y = abs( -5.0 );`

After evaluating `abs( -5.0 )`, the assignment is `double y = 5.0;`

And of course, we execute this assignment and store 5.0 in variable `y`.

Fulfilling the Contract

The function signature holds us to an important condition: we have to send back output in the type of the return type.

So for `abs`, because the return type is `double`, a call to that function must always return a `double`.

Our implementation is valid; it lives up to the condition that whatever the control flow in the function, a value of type `double` is returned.

If we forgot the `return` statement in the `else` block, that would be a compile-time error.

Of course, the compiler doesn't check the semantics: if you put `return 0.0;` as the body of the function, it compiles (but is wrong).

Return Types

The `abs` example has a return type of `double`, but any type can be the return type of a function.

This includes enumerated and programmer-defined (`struct`) types.

Every function must have a return type, but not all functions have a return value.

A function that does not return a value has a return type of `void`.

Into the `void`

If a function has return type `void`, it can still take input of zero or more input parameters. It simply returns no output value.

The `return` statement can appear in a `void` function, but when it does so, there's no return value following it.

The statement just appears as: `return;`

When a `return` statement is encountered in a `void` function, it just means to stop function execution at that point.

A function with return type `void` doesn't need a `return` statement.

If the end of the statement block is reached without encountering `return`, it's as if there is a `return` statement there.

```
public void PrintData( )
{
    // Additional statements not shown for space reasons
    cout << "-- End of List --" << endl;
}
```

After the `cout` statement, the function will return.
   The compiler will put `return;` there if it's not explicitly written.

`Main`

We've been using a function in our code all along, although we never really discussed it: `int main()`.

The return type is `int`. Thus, it returns an integer value.

The name is `main`. No surprise there.

The list of formal parameters is enclosed in `( )` brackets, and for `main()`, it is empty (so there are none).

And the block of statements enclosed in `{ }` braces is the body.

Function Example: Temperature Converter

This function converts temperatures in fahrenheit to celsius.

```
// Precondition: provided temperature is in Fahrenheit
// Postcondition: returns equivalent temp in Celsius
double convert_to_celsius ( double temp_f )
{
    return ( temp_f - 32 ) * 5.0d / 9.0d;
}
```

We can then call this function:

```
  double bodyTemp = convert_to_celsius( 98.6 );
```

We already saw in the `struct` that we could re-use some already existing constructs like the `Date` structure.

Functions allow us to re-use groups of statements.

In this lecture we have examined the “what” of functions, but have not yet examined the “why”.

In the next lecture, we will discuss the motivation for using functions.