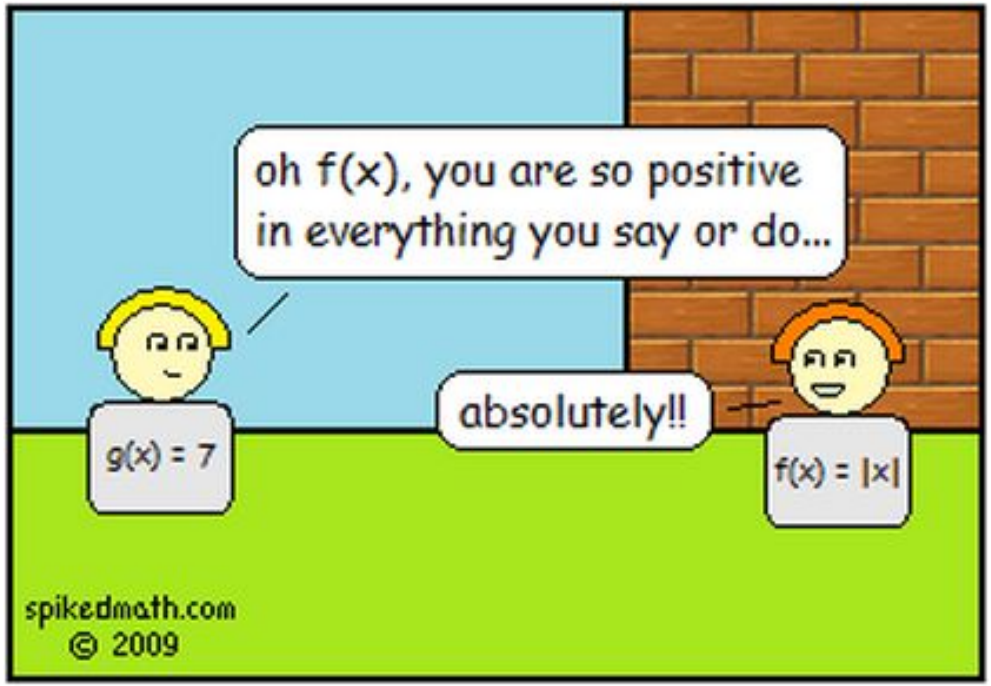
Lecture 6: Functions



Announcements and reminders

Submissions:

- HW 3 (functions) -- due Saturday at 6 PM

Course reading to stay on track:

- 5.1 - 5.4 today
- 5.5 - 5.8 before Wednesday
- 5.9 before Friday

Practicum 1

- 5:30 - 7 PM, Wednesday 20 Feb
- Let us know (Piazza) about conflicts.
  Include some verification (covering all our tails)

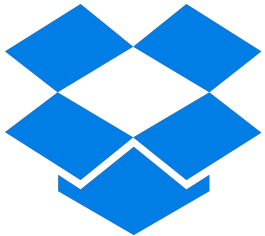
Back-up your work!

- GitHub (make it a **private** repository)
- Google Drive
- Dropbox

Don't care how. **No extensions** if you find that your assignment is lost to the ether the day before it is due.

- You might want to revert to an earlier version
- Your computer might crash
- You might need to access your assignment from another computer

- **Use cloud storage - <u>always save your work in more than one place</u>!**



Last time on *Starting Computing…*

We saw how **variables** work!

- Variable naming conventions
- Assignments

We saw how to **represent** different types of **numbers**!

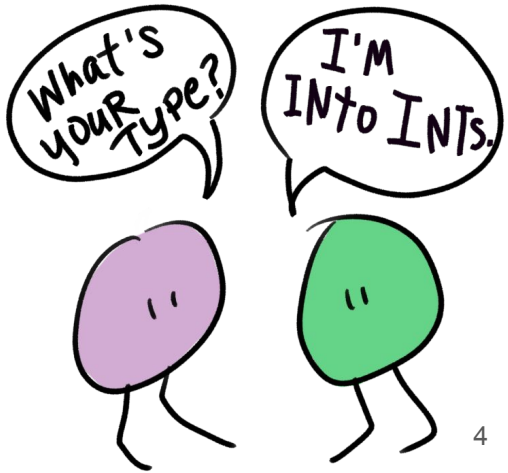
- Floating point (`double`)
- Integer (`int`)
- Constants (`const`)

We saw some **mathematical** functions and **arithmetic**!

- `+ - * / % pow() abs() sqrt()` etc...
- Need to include the right header files -- Google is your friend!

We saw how to manage **input** from keyboard and **output** to screen!

- `cin >> ... ;`
- `cout << ... ;`
- `fixed << setprecision(##)` and `setw(##)`



Chapter 5: Functions

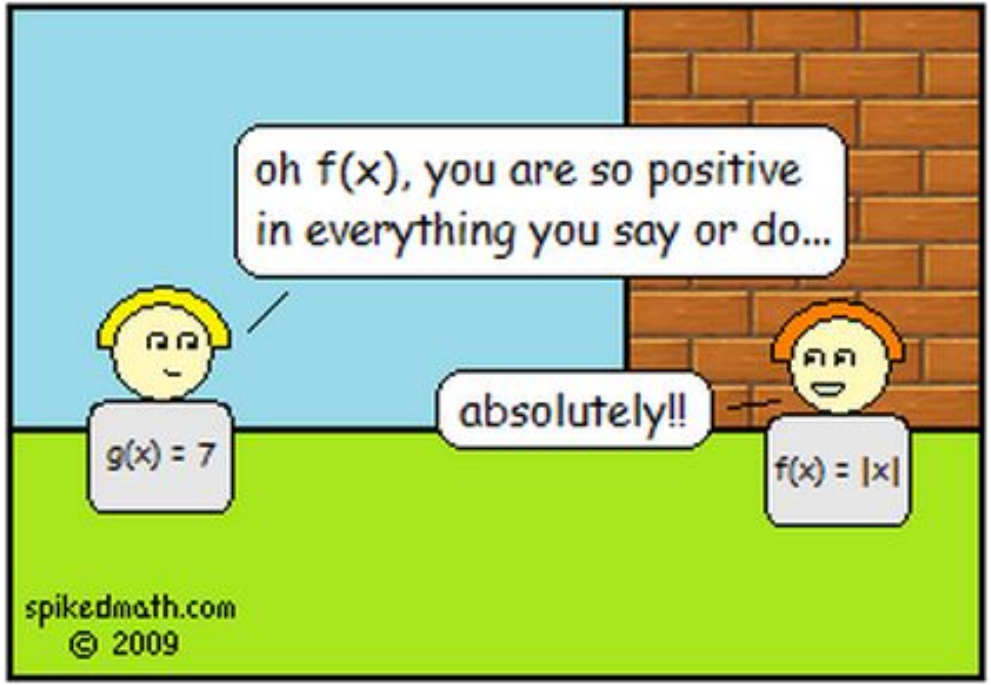
Chapter goals:

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To appreciate the importance of function comments
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To recognize when to use value and reference parameters

Chapter 5: Functions

Chapter topics:

- Functions as black boxes
- Implementing functions
- Parameter passing
- Return values
- Not return values
- Reusable functions
- Stepwise refinement
- Variable scope and globals
- ~~Reference parameters~~
- ~~Recursive functions~~



What is a function? Why is a function? Should I fear it?

Definition: A **function**…

- is a sequence of instructions with a name
- packages a computation into a form that can be easily understood and reused

What is a function? -- Example, in diagram

Definition: A **function**…

- is a sequence of instructions with a name
- packages a computation into a form that can be easily understood and reused

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

What is a function? -- Example, in words

Definition: A **function**…

- is a sequence of instructions with a name
- packages a computation into a form that can be easily understood and reused

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

- main is a function, and so is pow
- main ***calls*** the pow function, asking it to compute 2^3
- The main function is temporarily suspended while pow does its thing
- The instructions of the pow function execute and compute the result
- The pow function ***returns*** its result back to main
- main resumes execution

Parameters

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

Definitions:

- When another function calls the pow function, it provides **inputs**
  (e.g., the 2 and 3 in the call `pow(2, 3)`)
- In order to avoid confusion with user-provided inputs (`cin >>`), these values are called **parameter values**
- The **output** that the pow functions computes is called the **return value**
  (as opposed to output using `cout <<`)

Parameters

Note: An **output** statement (`cout`) ***does not*** return a value and the **return** statement ***does not*** display output

- output ≠ return
- `return` statement ends the called function and resumes execution of the program that called that function
  - Can also pass a value back to the calling program (e.g., `return 0;`)
- A `cout <<` statement communicates **only** with the user running the program
  - Just spits things out to the screen. That's it.

The Black Box Concept

You can think of a function as a “black box”

- Know what the box does, but can’t see what’s inside
- Like a **pressure cooker** -- can’t see inside, know what it does

Example: How did the `pow` function do its job?

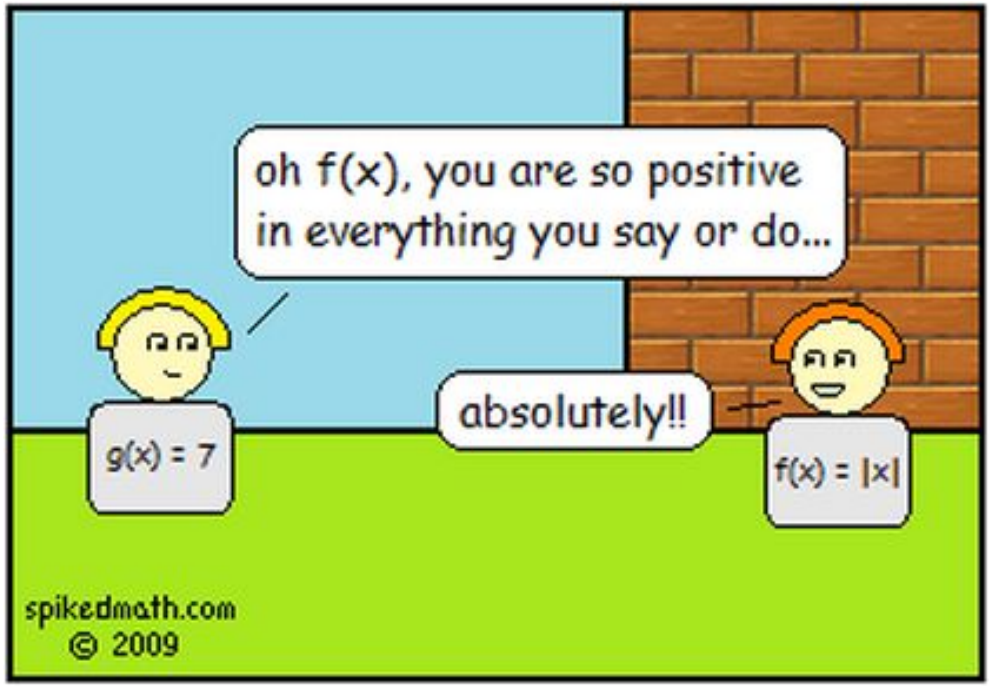
→ You didn’t need to know in order to use it

→ You only need to know its ***specification*** (inputs/outputs, syntax)

Chapter 5: Functions

Chapter topics:

- Functions as black boxes
- Implementing functions
- Parameter passing
- Return values
- Not return values
- Reusable functions
- Stepwise refinement
- Variable scope and globals



Implementing functions

Example: Calculate the volume of a cube

1) Pick a good descriptive name for the function
2) Give a type and name for each parameter

   There will be one parameter for each piece of information the function needs to do its job

3) Specify the type of the return value:

```
double cube_volume(double side_length);
```

4) Then write the body of the function, as statements enclosed in curly braces { … }

Implementing functions

Example: Calculate the volume of a cube

Note: Useful comments at the top: **description**, **parameters**, **return**, **algorithm**

Implementing functions

Example: Calculate the volume of a cube

Note: Useful comments at the top: **description**, **parameters**, **return**, **algorithm**

```
/*
    Computes the volume of a cube
    @param side_length -- the side length of the cube
    @return the volume of the cube
*/
double cube_volume(double side_length)
{
    double volume = side_length * side_length * side_length;
    return volume;
}
```

Implementing functions

Question: How do you know your function works as intended??

Implementing functions

Question: How do you know your function works as intended??

- You should always **test the function**
- Write a **main()** function to do this
- Let's test a couple different `side_lengths` for our `cube_volume` function and see if it outputs the correct volumes

Implementing functions

Question: How do you know your function works as intended??

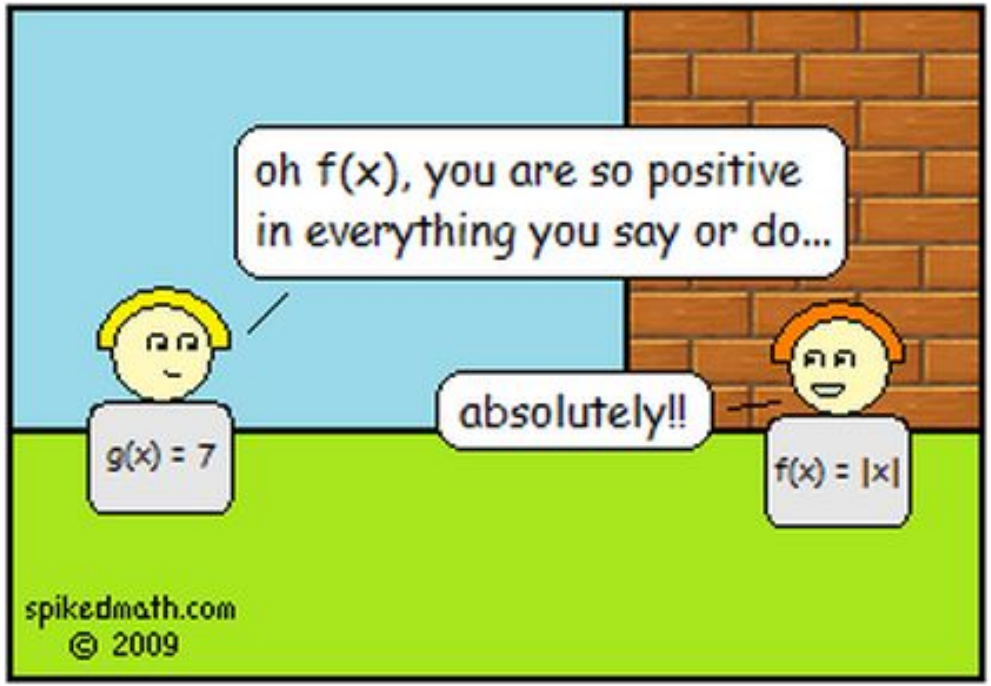
- You should always **test the function**
- Write a **main()** function to do this
- Let's test a couple different `side_lengths` for our `cube_volume` function and see if it outputs the correct volumes

```
int main()
{
    double result1 = cube_volume(2);
    double result2 = cube_volume(10);
    cout << "A cube with side length 2 has volume " << result1 << endl;
    cout << "A cube with side length 10 has volume " << result2 << endl;
    return 0;
}
```

Chapter 5: Functions

Chapter topics:

- Functions as black boxes
- Implementing functions
- Parameter passing
- Return values
- Not return values
- Reusable functions
- Stepwise refinement
- Variable scope and globals



Parameter Passing

When a function is called, a **parameter variable** is created for each value passed in.

Each parameter variable is **initialized** with the corresponding parameter value from the call.

```
int hours = read_value_between(1, 12);
...
int read_value_between(int low, int high);
```

Parameter Passing

Example: A call to our cube_volume function:

```
double result1 = cube_volume(2);
```

Here is the function definition:

```
double cube_volume(double side_length)
{
    double volume = side_length * side_length * side_length;
    return volume;
}
```

Let's keep track of the variables and their parameters:

```
result1, side_length, volume
```

Parameter Passing -- the play-by-play

First, the function call: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = _____`

Second, initializing function parameter variable: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = 2`

Third, execute cube_volume function:

```
double volume = side_length * side_length * side_length;
return volume;
```

→ `result1 = _____` `side_length = 2` `volume = 8`

Finally, after the function call: `double result1 = cube_volume(2);`

→ `result1 = 8`

Parameter Passing -- the play-by-play

First, the function call: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = _____`

Second, initializing function parameter variable: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = 2`

Third, execute cube_volume function:

```
double volume = side_length * side_length * side_length;
return volume;
```

→ `result1 = _____` `side_length = 2` `volume = 8`

Finally, after the function call: `double result1 = cube_volume(2);`

→ `result1 = 8`

Parameter Passing -- the play-by-play

First, the function call: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = _____`

Second, initializing function parameter variable: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = 2`

Third, execute cube_volume function:

```
double volume = side_length * side_length * side_length;
return volume;
```

→ `result1 = _____` `side_length = 2` `volume = 8`

Finally, after the function call: `double result1 = cube_volume(2);`

→ `result1 = 8`

Parameter Passing -- the play-by-play

First, the function call: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = _____`

Second, initializing function parameter variable: `double result1 = cube_volume(2);`

→ `result1 = _____` `side_length = 2`

Third, execute `cube_volume` function:

```
double volume = side_length * side_length * side_length;
return volume;
```

→ `result1 = _____` `side_length = 2` `volume = 8`

Finally, after the function call: `double result1 = cube_volume(2);`

→ `result1 = 8`

What just happened…?

We learned what a function is!

We learned how to implement a function!

We learned how to pass parameters into a function and send return values back out!

We learned a little bit about the **scope** of variables!

- Much more on this later!