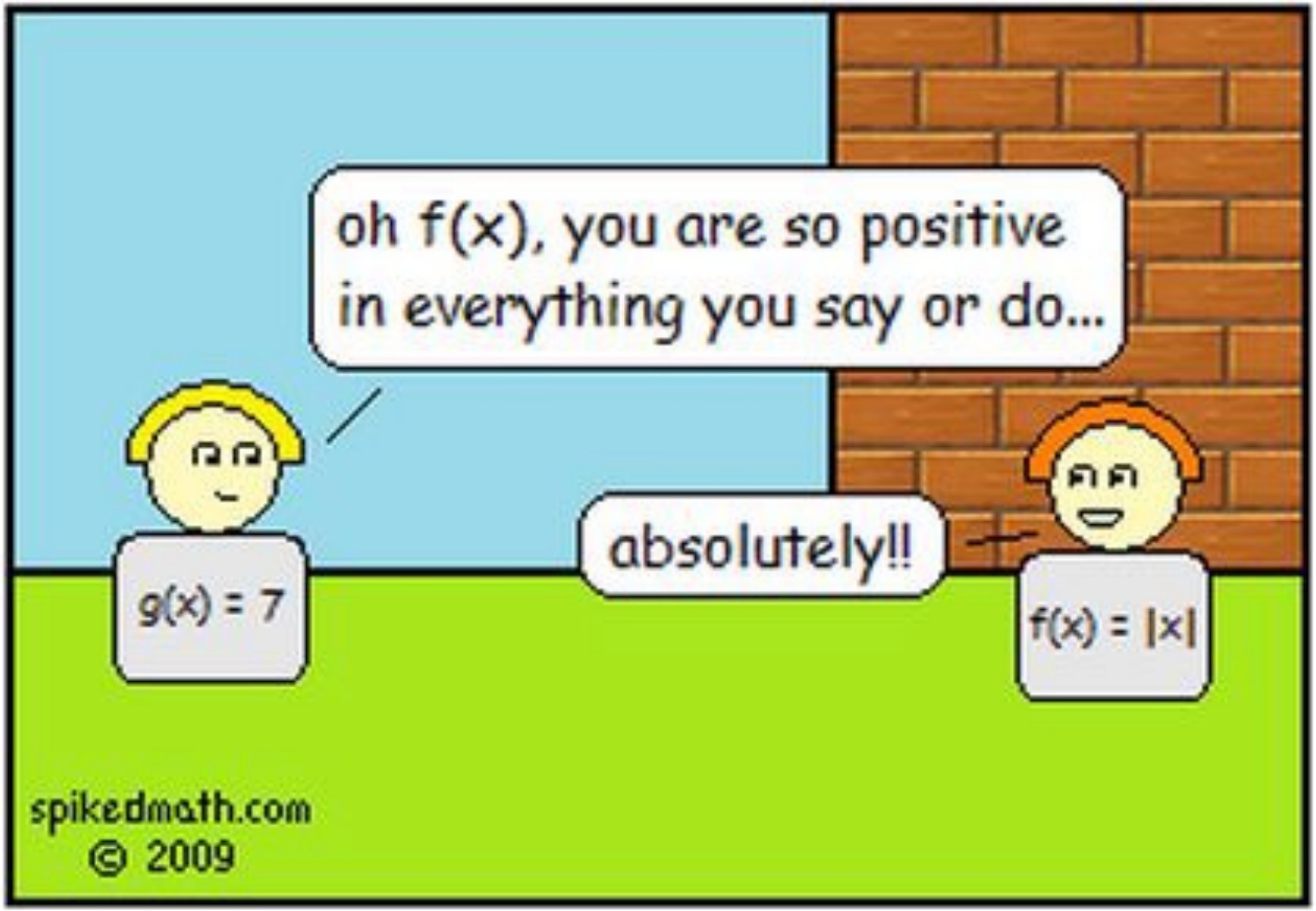

oh f(x), you are so positive in everything you say or do...
absolutely!!
g(x) = 7
f(x) = |x|
spikedmath.com
© 2009

Functions

Practicum Retake

- During final exam slot (TBD)
- Retake practicums 1/2/3..
- Most recent score will be recorded
- 67% average is required for the practicum to receive a C-
- Details available on Syllabus

Due this week

- **Homework 4**
  - Write solutions in VSCode and paste in Autograder, **Homework 4 CodeRunner**.
  - Zip your .cpp files and submit on canvas **Homework 4**.
  - Start early and extra-credit question
- No Quiz this week
- Check the due date! **No late submissions!!**

Today

- What are functions?
- Implementing functions
- Function parameters and arguments
- Return values

Functions

What is a function?

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

  - example:

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

Functions as Black Box

- 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

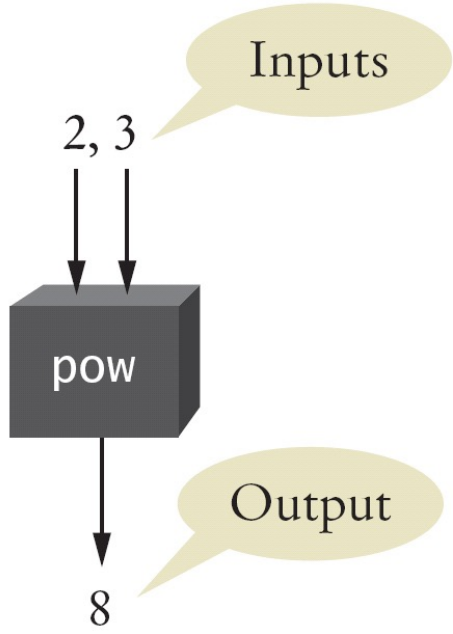
Functions as Black Box

- 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)



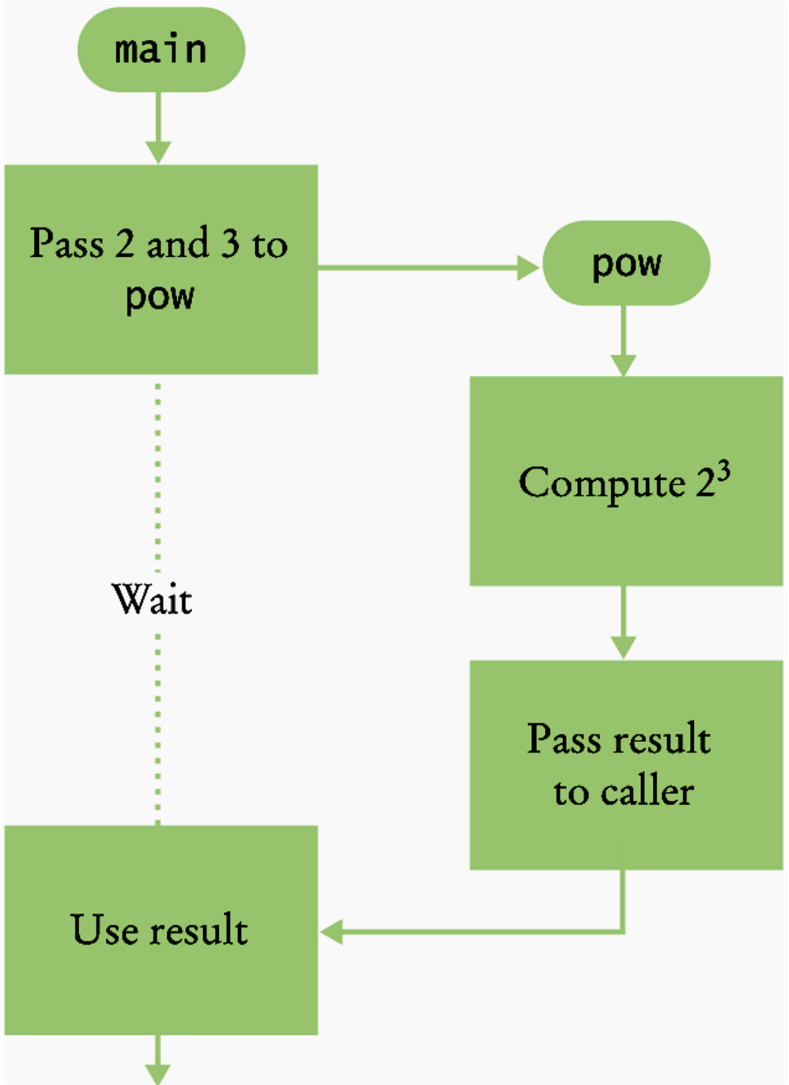
Calling a function

- 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

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

Flowchart: Calling a function

Execution flow during a function call



Actual parameters/arguments

- 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 **function arguments**
- The output that the pow functions computes is called the return value
  - (as opposed to output using cout <<)

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

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.

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 cubeVolume(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

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

Implementing functions

- 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 = cubeVolume(2);
    double result2 = cubeVolume(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;
}
```

Parameter Passing

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);
```

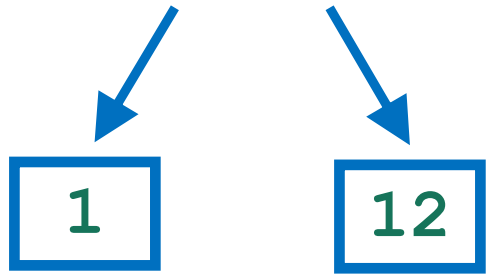
Parameter Passing

- 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 cubeVolume function:

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

- Here is the function definition:

```
double cubeVolume(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 = ____`

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 = cubeVolume(2);`

→ `result1 = ____  side_length = 2`

Parameter Passing – the play-by-play

- **Third,** execute `cubeVolume` **function:**

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

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

Parameter Passing – the play-by-play

- **Third,** execute `cubeVolume` **function:**

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

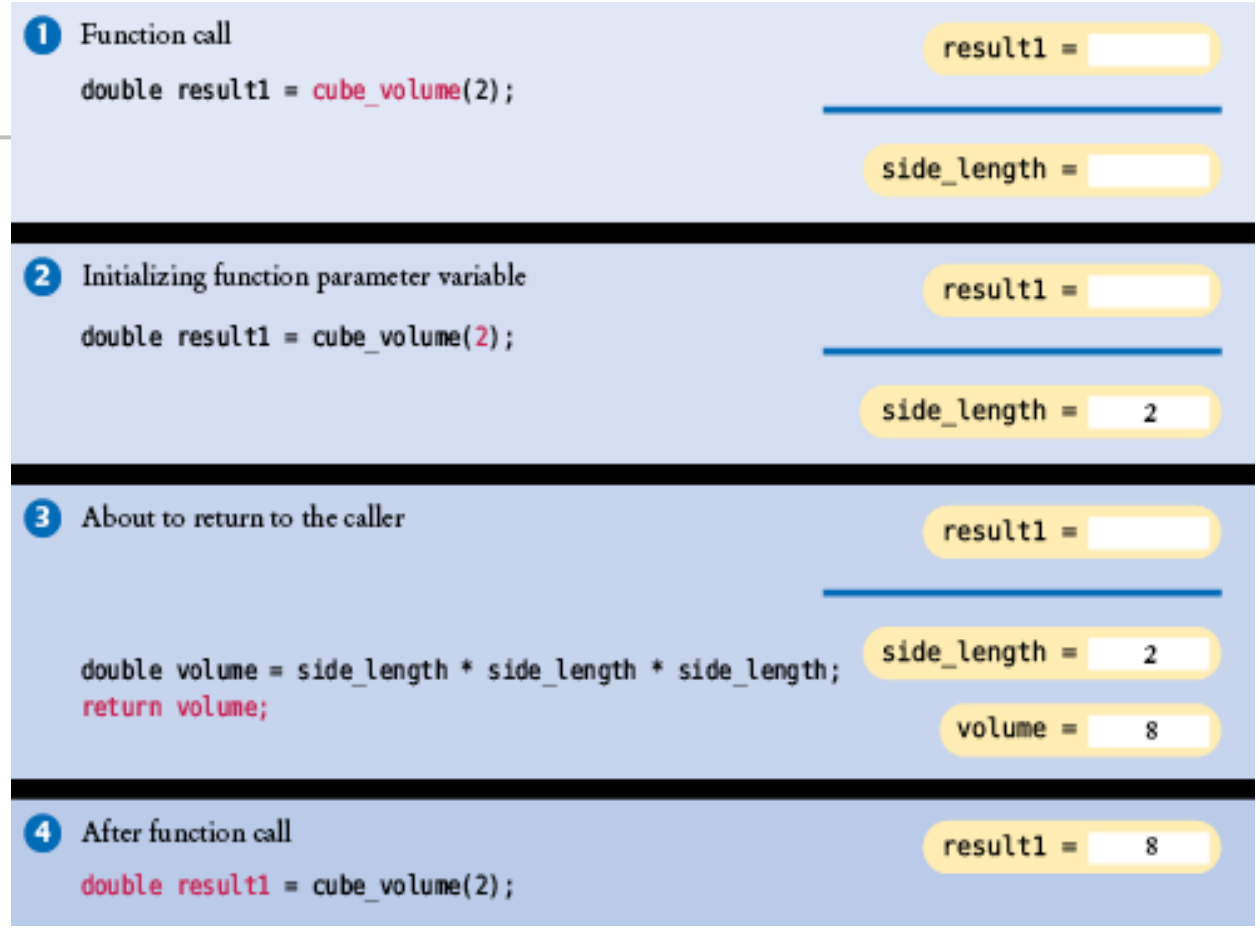
→ `result1 = ____  side_length = 2  volume = 8`

- **Finally,** after the function call: `double result1 = cubeVolume(2);`

→ `result1 = 8`

Parameter Passing

- In the calling function (`main`), the variable **`result1`** is declared.
- When the **`cube_volume`** function is called, the parameter variable **`side_length`** is created & initialized with the value that was passed in the call (2).
- After the return statement, the local variables `side_length` and `volume` disappear from memory.
- The calculated volume is stored in the variable, `result1`



Return Values

Return Values

The `return` statement ends the function execution. This behavior can be used to handle unusual cases.

What should we do if the side length is negative?

We choose to return a zero and not do any calculation:

```
double cube_volume(double side_length)
{
  if (side_length < 0)
                  return 0;
  double volume = side_length * side_length * side_length;
  return volume;
}
```

- Nothing is executed after a return statement !!!
- Execution returns to main()

Return Values: Shortcut

The **`return`** statement can return the value of any expression.

Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

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

Common Error – Missing Return Value

Your function always needs to return something.

The code below: what is returned if the call passes in a negative value?

You need to ensure all paths of execution include a `return` statement.

```
double cube_volume(double side_length)
{
  if (side_length >= 0)
  {
     return side_length * side_length * side_length;
  }
}
```

Functions without return values

- Consider the task of writing/printing a string with the following format around it
- Any string could be used
- For example, the string “Hello” would produce:

```
-------
!Hello!
-------
```

Functions without return values – the *void* type

Definition: This kind of function is called a void function

- `void` is a type, just like `int` or `double`
- Use a return type of void to indicate that a function does not return a value
- `void` functions are used to simply perform a sequence of instructions, but not return any particular values to the caller
- Example: `void box_string(string str)`

Functions without return values – the *void* type

```
void box_string(string str)
{
	int n = str.length();
	for (int i = 0; i < n + 2; i++)
	{
		cout << "-";
	}
	cout << endl;
	cout << "!" << str << "!" << endl;
	for (int i = 0; i < n + 2; i++)
	{
		cout << "-";
	}
	cout << endl;
}
```

- Note that this function doesn't compute any value.
- It performs some actions and then returns to the caller without returning a value
- There is no return statement

Calling void functions

- A void function has no return value, so we cannot call it with assignment like this:

```
result = box_string(“Hello”);
```
// Error: box_string does not return a result

- Instead, we call it like this, without assignment:

```
box_string(“Hello”);
```