# Functions

Almost as good as ice cream.

Check out the examples posted on Blackboard

Seeing functions in action will probably help a lot.

# Functions

You've been writing functions since day one:

- that `main()` thingy, which is required in all C++ applications, is a function!
- the operating system calls the `main()` function when you run your program

Some predefined functions you hopefully recognize:

- `getline()`
- `acos()`
- `sqrt()`
- `width()`
- `precision()`

# Why Use Functions?

(short answer: they make you a better person)

# Modularity

## Functions encapsulate code with a specific purpose

- the more specific, the better

## Divide and Conquer

- break a large project into smaller, more manageable pieces
- ideally, have many independent functions that each do a single task exceptionally well

## Functions (modules) provide:

- a logical grouping of related code
- ease of testing
- increased readability

# Abstraction

A function provides a service

- we generally don't care HOW it does it, so long as the result is what we expect
- once a function has been thoroughly tested, we can use it without caring about the details

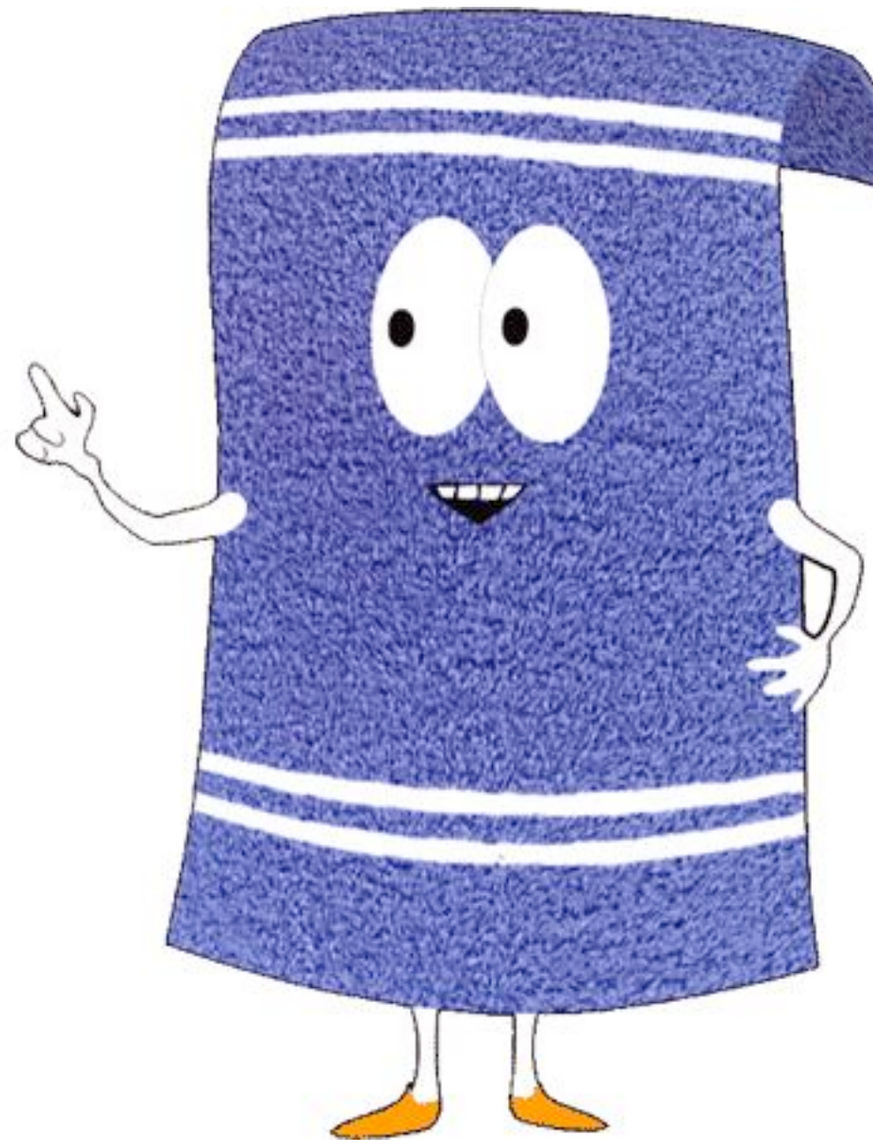
Take the `sqrt()` function as an example:

- it finds the square root of some number
- as long as it returns the right answer, you probably don't care *how* it does it.

# D.R.Y. Programming

Imagine you're coding a simple program…

- you want to calculate x^y
- cmath hasn't been created (alternate reality)
- so, you come up with some code...



# D.R.Y. Programming

```
1  int main() {
2
3      // calculate 5 to the 4th power (5^4)
4      int number = 5, power = 4, result = 1;
5
6      while (--power) {
7          result *= number;
8      }
9
10     // print the result
11     cout << "5^4 = " << result << endl;
12
13     system("PAUSE");
14     return 0;
15
16 }
```

- You figure out a cool way to do the calculation and implement it once...

# D.R.Y. Programming

```
int main() {

    // calculate 5 to the 4th power (5^4)
    int number1 = 5, power1 = 4, result1 = 1;

    while (--power1) {
        result1 *= number1;
    }

    // calculate 6 to the 5th power (6^4)
    int number2 = 6, power2 = 5, result2 = 1;

    while (--power2) {
        result2 *= number2;
    }

    // print the results
    cout << "5^4 = " << result1 << endl;
    cout << "6^5 = " << result2 << endl;

    system("PAUSE");
    return 0;

}
```

- and then again...

# D.R.Y. Programming

```
int main() {

    // calculate 5 to the 4th power (5^4)
    int number1 = 5, power1 = 4, result1 = 1;

    while (--power1) {
        result1 *= number1;
    }

    // calculate 8 to the 5th power (6^4)
    int number2 = 8, power2 = 5, result2 = 1;

    while (--power2) {
        result2 *= number2;
    }

    // calculate 18 to the 3rd power (18^3)
    int number3 = 18, power3 = 3, result3 = 1;

    while (--power3) {
        result3 *= number3;
    }

    // calculate 7 to the 9th power (7^9)
    int number4 = 8, power4 = 5, result4 = 1;

    while (--power4) {
        result4 *= number4;
    }

    // calculate 12 to the 3rd power (12^3)
    int number5 = 12, power5 = 3, result5 = 1;
```

- and then 38 more times (you're having fun)

# D.R.Y. Programming

...and then you realize that your code has an error.

(make that 40 errors)

```cpp
int main() {

    // calculate 5 to the 4th power (5^4)
    int number = 5, power = 4, result = 1;

    while (--power) {
        result *= number;
    }

    // print the result
    cout << "5^4 = " << result << endl;

    system("PAUSE");
    return 0;

}
```

# D.R.Y. Programming

Don't Repeat Yourself

Problems with (needlessly) repetitious code:

- debugging is harder
- new features are more difficult to add
- readability suffers
- more work to create!

Functions help reduce repetition:

- they can apply the same logic or repeat the same process using whatever arguments they're passed
- instead of copy/pasting code, use a function instead!

# Function Basics

(they really do make you a better person)

# Function Syntax

General syntax:

```
/**
 * a header comment describing the function's purpose, the arguments it
 * accepts, and any special conditions that must be met
**/
return_type functionName( argument_list ) {
    // function body
}
```

Components:

- `return_type` is the type of value produced by the function (can be any data type)
- `functionName` is the name used to call the function (must be a valid identifier)
- `argument_list` is a comma-separated list of 0 or more inputs that the function takes

# Function Names

Function names must be valid C++ identifiers:

- can only contain letters, numbers, and the underscore character
- cannot have a number as the first character
- cannot be a C++ keyword
- same rules as those for variable names!

To call a function,

- just add a pair of parentheses `()` to the function name, with any arguments to be passed inside
- for example, to get the square root of 25 using the `sqrt` function: `sqrt(25)`

# The return type

The return type of a function is a promise:

- `int main()` promises to always return an int
- `double sqrt()` promises to always return a double

You've seen this before:

```
// main promises to return an int
int main() {
    return 0; // this statement 'returns' an int (0)
}
```

The compiler will complain if you break your promise!

# The return keyword

The `return` keyword immediately exits the current function

- it returns control to whatever function (or operating system) called it
- it returns whatever value you specify as the result of the function call

Return value in action:

```
// this function must return a string
string getText() {
    return “Hi!”; // calling this fn will yield “Hi!”
}

string greeting = getText(); // greeting’s value = “Hi”
```

# Function Arguments

General syntax:

```
// the initial values of arg1-argN depend on the values passed to the
// function at the time you call it
int myFunction(data_type arg1, data_type arg2, ..., data_type argN) {
    return 0;
}
```

Functions can accept as many inputs (arguments) as you need

- this makes them extremely versatile and useful
- each argument is specified by both a data_type and a name (*arg1*, *arg2*, etc…)
- multiple arguments are separated by commas
- the values of arguments depend on the values passed to the function at the time you call it

# Function Arguments

Assume we have this function:

```
// height and radius must both be doubles
double getCylinderVolume(double height, double radius) {
    const double PI = acos(-1);
    // calculate and return the volume
    return PI * radius * radius * height; // return volume as a double
}
```

To call this function:

- we always have to provide exactly 2 arguments
- both arguments must be doubles (or numeric values that can be converted to doubles)

Example:

```
double volume = getCylinderVolume(10, 5); // height will be 10, radius 5
```

# Using a Function

```
#include <iostream>
using namespace std;

/**
 * Returns @base raised to the power @exponent,
 * which must NOT be negative.
**/
int myPow(int base, int exponent) {
    int result = 1;

    while (exponent--) {
        result *= base;
    }

    return result;
}

/**
 * The ubiquitous main() function
**/
int main() {

    cout << myPow(5, 3) << endl;

    return 0;
}

```

- If the entire function is placed above the `main()` function, calling from inside `main()` works

# Using a Function

```
1  #include <iostream>
2  using namespace std;
3
4  /**
5   * The ubiquitous main() function
6  **/
7  int main() {
8
9      cout << myPow(5, 3) << endl;
10
11     return 0;
12 }
13
14 /**
15  * Returns @base raised to the power @exponent,
16  * which must NOT be negative.
17 **/
18 int myPow(int base, int exponent) {
19     int result = 1;
20
21     while (exponent--) {
22         result *= base;
23     }
24
25     return result;
26 }
27
28
```

- Placing the entire function *below* main() and trying to call it does NOT work

- Why?

# Function Prototypes

A function prototype provides the compiler everything it needs to be able to validate calls to the function.

```
1 // function prototype = function header + semicolon
2 int raiseToPower(int base, int exponent);
3
```

Argument names are optional in the prototype:

```
1 // function prototype = function header + semicolon
2 int raiseToPower(int, int);
3
```

The prototype also provides the programmer all the info he needs:

- the name of the function
- the number and types of arguments it requires
- what type of value to expect in return
- the header comment provides additional important details

# Using a Function

```
1  #include <iostream>
2  using namespace std;
3
4  /**
5   * Returns @base raised to the power @exponent,
6   * which must NOT be negative.
7  **/
8  int myPow(int base, int exponent);
9
10 // the ubiquitous main() function
11 int main() {
12
13     cout << myPow(5, 3) << endl;
14
15     return 0;
16 }
17
18 // implementation
19 int myPow(int base, int exponent) {
20     int result = 1;
21
22     while (exponent--) {
23         result *= base;
24     }
25
26     return result;
27 }
28
```

- Placing a prototype for the function above the main() routine satisfies the compiler.

- If the function is NOT implemented, the *linker* will complain (and rightfully so!)

- Also, the function prototype and implementation must both agree on everything but the names of the arguments.

# Using a Function

```
1  #include <iostream>
2  using namespace std;
3
4  /**
5   * Returns @base raised to the power @exponent,
6   * which must NOT be negative.
7  **/
8  int myPow(int base, int exponent);
9
10 // the ubiquitous main() function
11 int main() {
12
13     cout << myPow(5, 3) << endl;
14
15     return 0;
16 }
17
18 // implementation missing! =(
19
20
21
22
23
24
25
26
27
28
```

- Placing a prototype for the function above the main() routine satisfies the compiler.

- If the function is NOT implemented, the *linker* will complain (and rightfully so!)

- Also, the function prototype and implementation must both agree on everything but the names of the arguments.

# Prototype & Implementation in Different Files

`myMathLib.h` (header file)

```
1  #pragma once // include only once in program
2
3  /**
4   * Returns @base raised to the power @exponent,
5   * which must NOT be negative.
6  **/
7  int myPow(int base, int exponent);
8
```

- add `#pragma once` to the `myMathLib.h` header file

`myMathLib.cpp` (implementation file)

```
1  #include "myMathLib.h"
2
3  // function implementation
4  int myPow(int base, int exponent) {
5      int result = 1;
6
7      while (exponent--) {
8          result *= base;
9      }
10
11     return result;
12 }
13
```

- `#include “myMathLib.h”` in the `myMathLib.cpp` implementation file

# Prototype & Implementation in Different Files

`main.cpp`

```
1  #include <iostream>
2  #include "myMathLib.h" // include function file
3
4  using namespace std;
5
6  int main() {
7
8      int number = 10;
9      int power  = 4;
10
11     cout << myPow(number, power) << endl;
12
13     system("PAUSE");
14     return 0;
15 }
16
```

- To use your new math library, simply `#include “myMathLib.h”` in your `main.cpp` file