



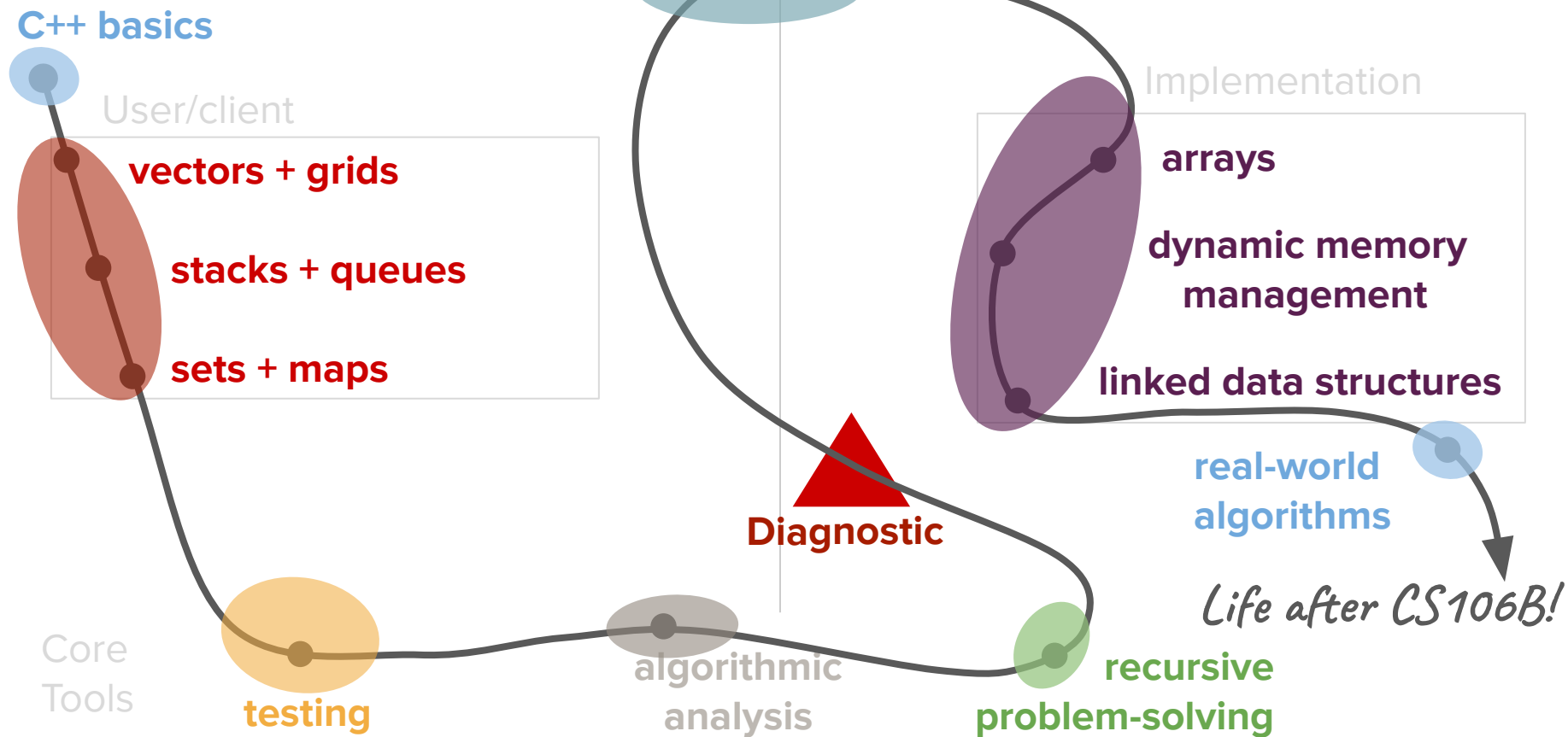
# Programming Fundamentals in C++

**What programming language are you most  
comfortable with?**

(put your answers the chat)



# Roadmap



# Roadmap

C++ basics

User/client

**vectors + grids**

**stacks + queues**

**sets + maps**

Object-Oriented  
Programming

Implementation

**arrays**

**dynamic memory  
management**

**linked data structures**

real-world  
algorithms

*Life after CS106B!*

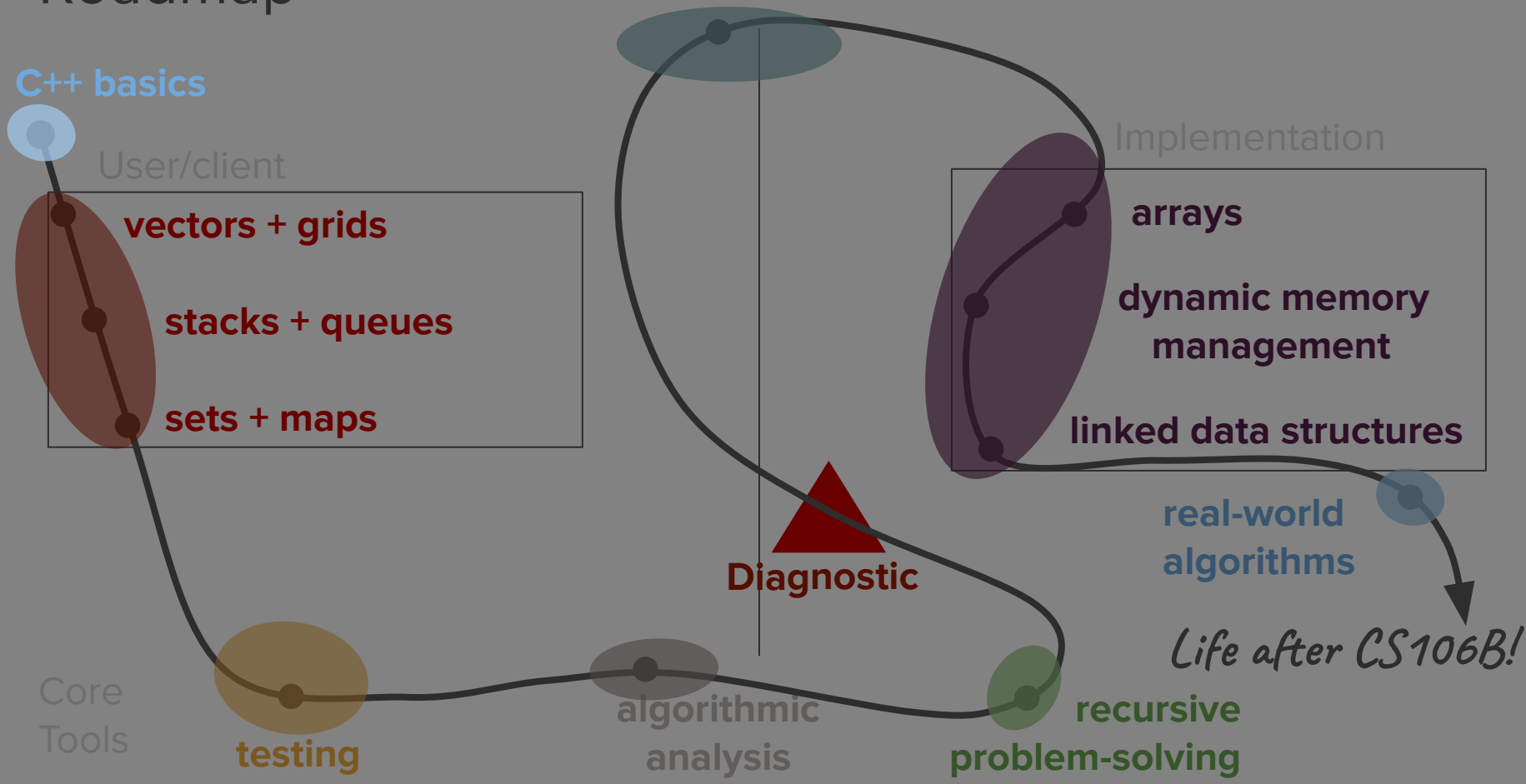
**Diagnostic**

Core  
Tools

**testing**

algorithmic  
analysis

recursive  
problem-solving





# Today's questions

Why C++?

What do core programming  
fundamentals look like in C++?

How do we test code in CS106B?

What's next?



# Why C++?

# Review: How is C++ different from other languages?

- C++ is a **compiled language** (vs. interpreted)
- C++ gives us access to lower-level computing resources (e.g. more direct control over computer memory)

This makes it a great tool for better understanding abstractions!

- If you're coming from a language like Python, the syntax will take some getting used to.

Like learning the grammar and rules of a new language, typos are expected. But don't let this get in the way of working toward literacy!

错别字是意料之中的

不要让这个妨碍到你的识字率

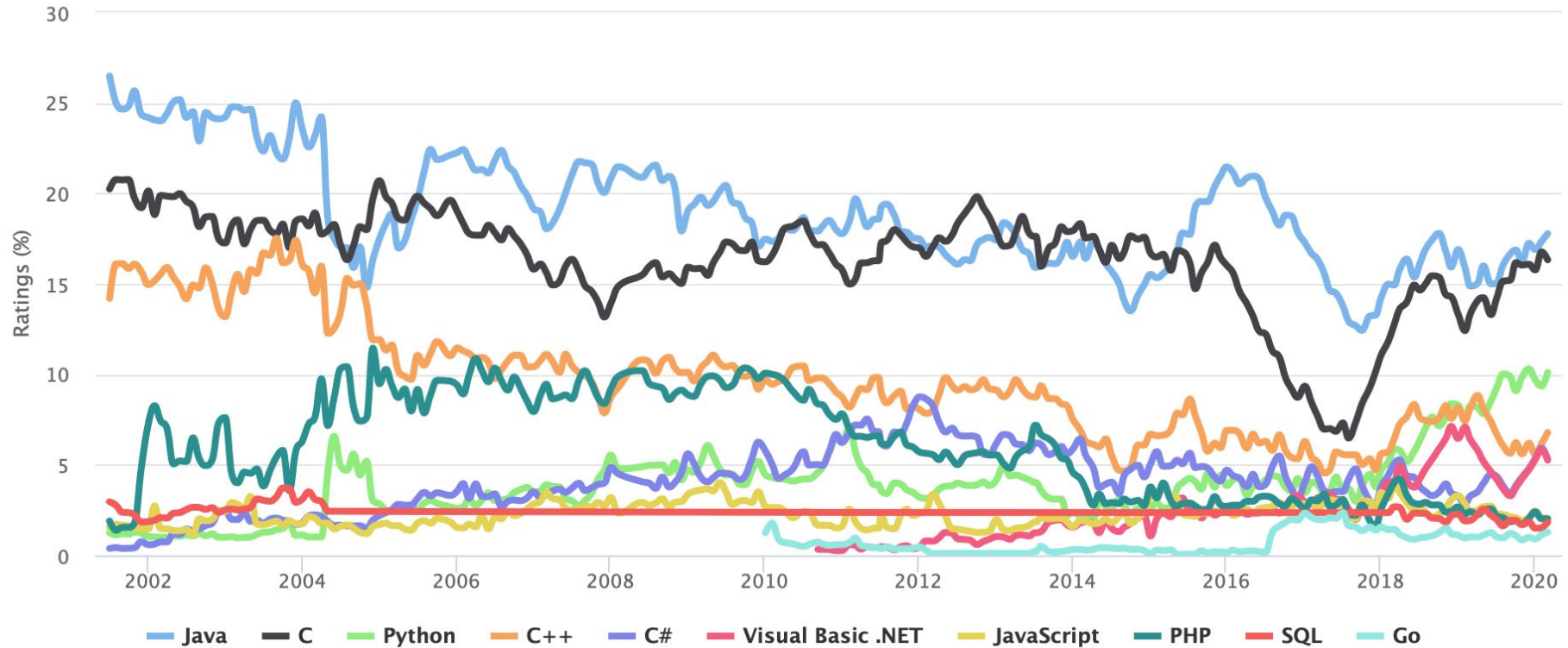


# Zoom Poll!

Where does C++ rank among the popular programming languages of the world?

# TIOBE Programming Community Index

Source: [www.tiobe.com](http://www.tiobe.com)







# C++ Overview

If someone claims to have the perfect programming language, he is either a fool or a salesman or both.

– *Bjarne Stroustrup*, Inventor of C++



# C++ History

- C++ is a **high-performance, robust (and complex)** language built **on top of the C programming language** (originally named *C with Classes*)
  - Bjarne Stroustrup, the inventor of C++, chose to build on top of C because it was fast, powerful, and widely-used
- C++ has been an **object-oriented language** from the beginning
  - We will spend the middle portion of this class talking about the paradigm of object-oriented programming
- C++ is quite mature and has become complex enough that it is challenging to master the language
  - Our goal in this class will be to help you become literate in C++ as a second programming language



# C++ Benefits and Drawbacks

## Benefits

- C++ is **fast**
  - Get ready for the Python vs C++ speed showdown during Assignment 1!
- C++ is **popular**
  - Many companies and research projects use C++ and it is common for coding interviews to be conducted in C++
- C++ is **powerful**
  - C++ brings you closer to the raw computing power that your computer has to offer

## Drawbacks

- C++ is **complex**
  - We will rely on the Stanford C++ libraries to provide a friendlier level of abstraction
  - In the future, you may choose to explore the *standard* libraries
- C++ can be **dangerous**
  - "With great power comes great responsibility"



# What do core programming fundamentals look like in C++?

*Get ready for a whirlwind tour!*



# Comments, Includes, and Console Output



# Comments

- Single-line comments

```
// Two forward slashes comment out the rest of the line
```

```
cout << "Hello, World!" << endl; // everything past the double-slash is a comment
```

- Multi-line comments

```
/* This is a multi-line comment.
```

```
* It begins and ends with an asterisk-slash.
```

```
*/
```

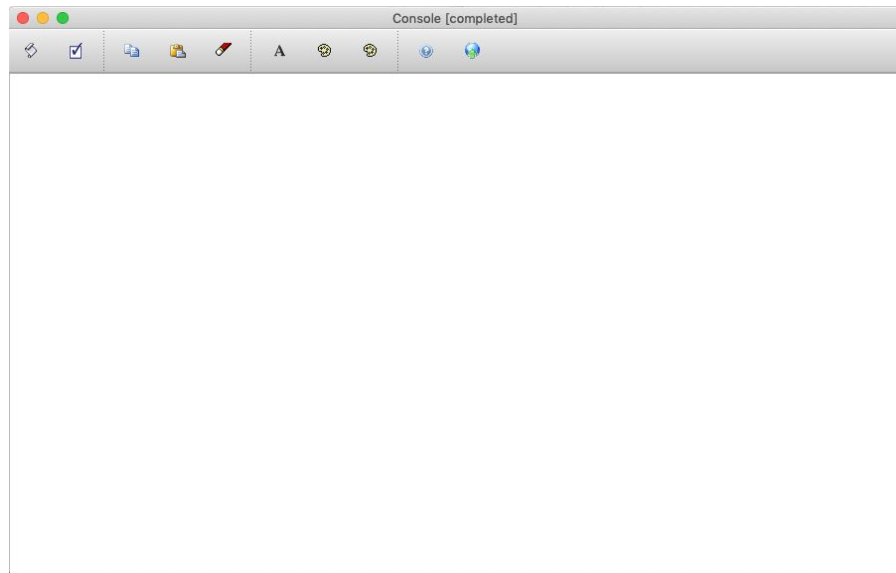


# Includes

- Utilizing code written by other programmers is one of the most powerful things that you can do when writing code.
- In order to make the compiler <sup>知道</sup> aware of other code libraries or other code files that you want to use, you must ***include a header file***. There are two ways that you can do so:
  - **#include <iostream>**
    - Use of the angle bracket operators is usually reserved for code from the C++ Standard library
  - **#include "console.h"**
    - Use of the quotes is usually reserved for code from the Stanford C++ libraries, or code in files that you have written yourself

# Console Output

- The console is the main <sup>场地</sup>venue that we will use in this class to communicate information from a program to the user of the program.

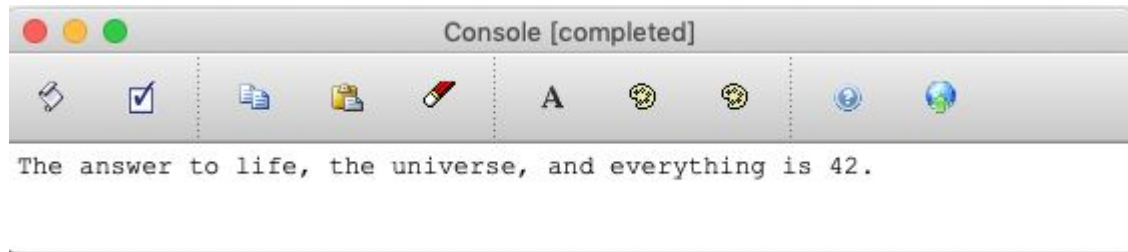




# Console Output

- The console is the main venue that we will use in this class to communicate information from a program to the user of the program.
- In C++, the way that you get information to the console is by using the **cout** keyword and angle bracket operators (<<).

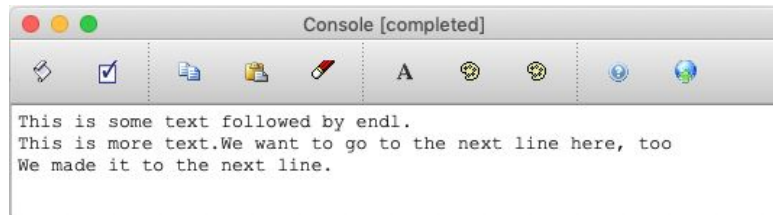
```
cout << "The answer to life, the universe, and everything is " << 42 << "." << endl;
```



# Console Output

- The console is the main venue that we will use in this class to communicate information from a program to the user of the program.
- In C++, the way that you get information to the console is by using the **cout** keyword and angle bracket operators (<<).
- The **endl** is necessary to put the cursor on a different line. Here is an example with and without the **endl** keyword.

```
cout << "This is some text followed by endl." << endl;  
cout << "This is more text.";  
cout << "We want to go to the next line here, too" << endl;  
cout << "We made it to the next line." << endl;
```



**Note: In C++, all programming statements must end in a semicolon.**

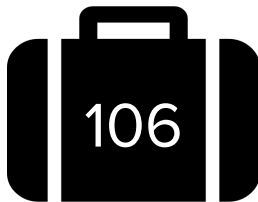


# Variables and Types

# Variables

- A way for code to store information by associating a value with a name

*We will think of  
a variable as a  
named  
container  
storing a value.*



classNum

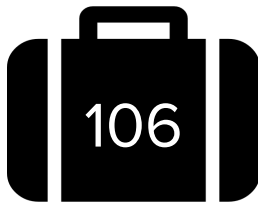


tuesdayTemp

# Variables

- A way for code to store information by associating a value with a name

*Note: C++ uses  
the **camelCase**  
naming  
convention*



Python: class\_num  
classNum



tuesdayTemp



# Variables

- A way for code to store information by associating a value with a name
- **Variables are perhaps one of the most fundamental aspects of programming!** Without variables, the expressive power of our computer programs would be severely degraded.

# Types

- As you should know from prior programming classes, all variables have a type associated with them, where the type describes the representation of the variable.
- Examples of types in C++
  - `int`

42

106

-3



# Types

- As you should know from prior programming classes, all variables have a type associated with them, where the type describes the representation of the variable.
- Examples of types in C++
  - `int` ( or long )
  - `double`

1.06

4.00

-18.3454545





# Types

- As you should know from prior programming classes, all variables have a type associated with them, where the type describes the representation of the variable.

- Examples of types in C++

- `int`
- `double`
- `string`

`"Hello, World!"`

`"CS106B"`

`"I love computer  
science <3"`

# Types

- As you should know from prior programming classes, all variables have a type associated with them, where the type describes the representation of the variable.
- Examples of types in C++
  - `int`
  - `double`
  - `string`
  - `char`

'a'

'&'

'3'



# Types

- As you should know from prior programming classes, all variables have a type associated with them, where the type describes the representation of the variable.
- Examples of types in C++
  - `int`
  - `double`
  - `string`
  - `char`
- In C++, *all types must be explicitly defined when the variable is created, and a variable cannot change its type.*

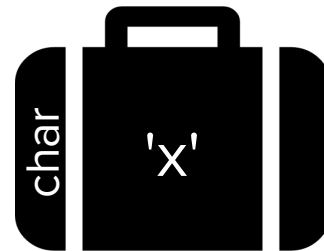
# Typed Variables

```
int a; // declare a new integer variable
a = 5; // initialize the variable value

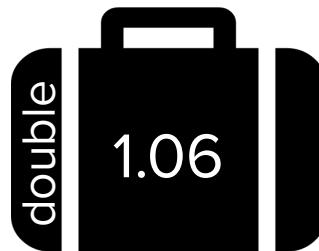
char c = 'x'; // c is a char ("character")
double d = 1.06; // d is a double, a type
used to represent decimal numbers
string s = "this is a C++ string";
double a = 4.2; // ERROR! You cannot
redefine a variable to be another type
int a = 12; // ERROR! You do not need the
type when re-assigning a variable
a = 12; // this is okay, updates variable
value
```



a



c



d

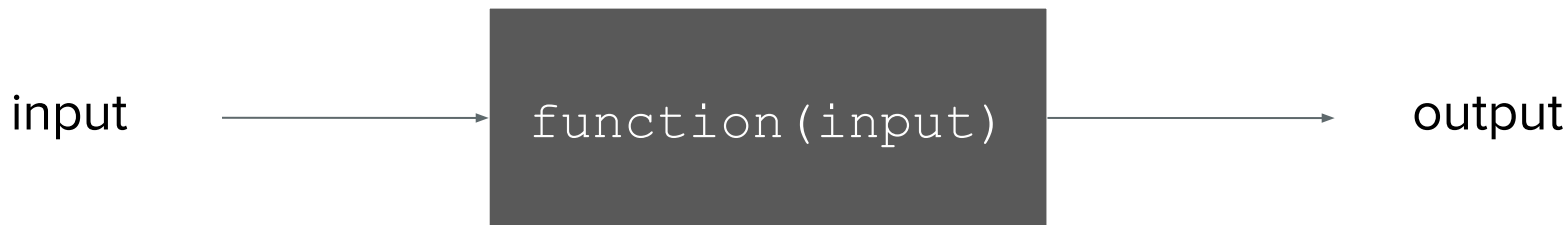


s

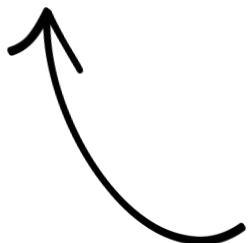


# Functions and Parameters

# Anatomy of a function



*parameters/  
arguments*

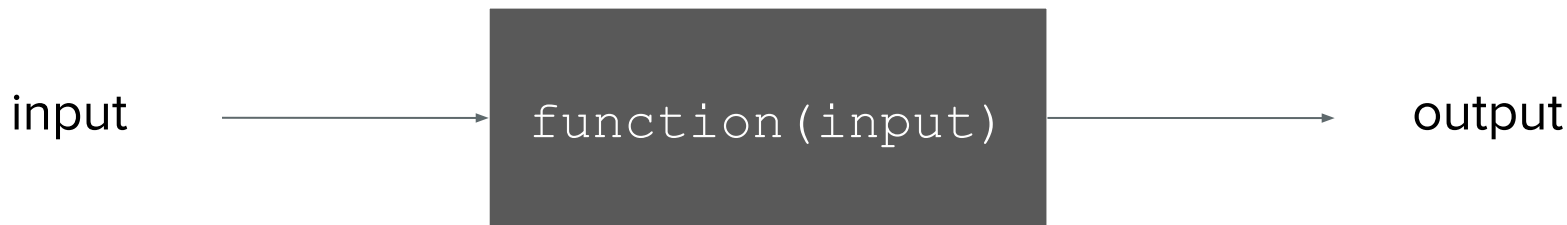


## Definition

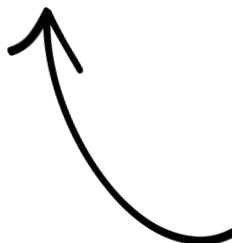
### **parameter(s)**

One or more variables that your function expects as input

# Anatomy of a function



*parameters/  
arguments*

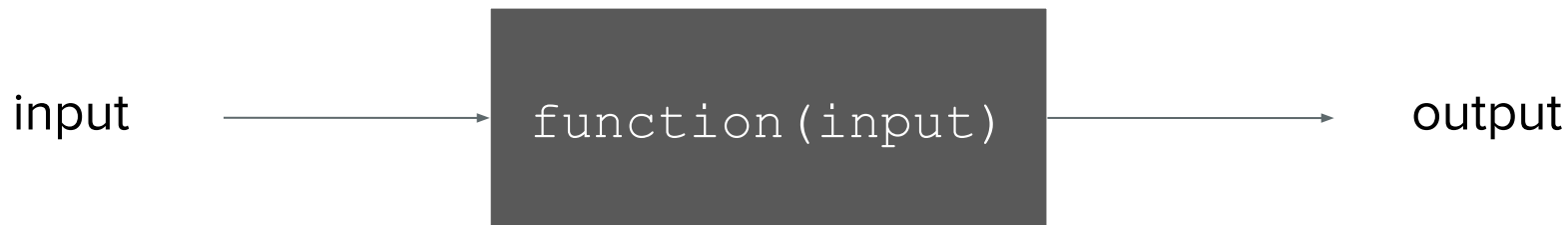


## Definition

### **argument(s)**

The values passed into your function and assigned to its parameter variables

# Anatomy of a function



## Definition

### **return value**

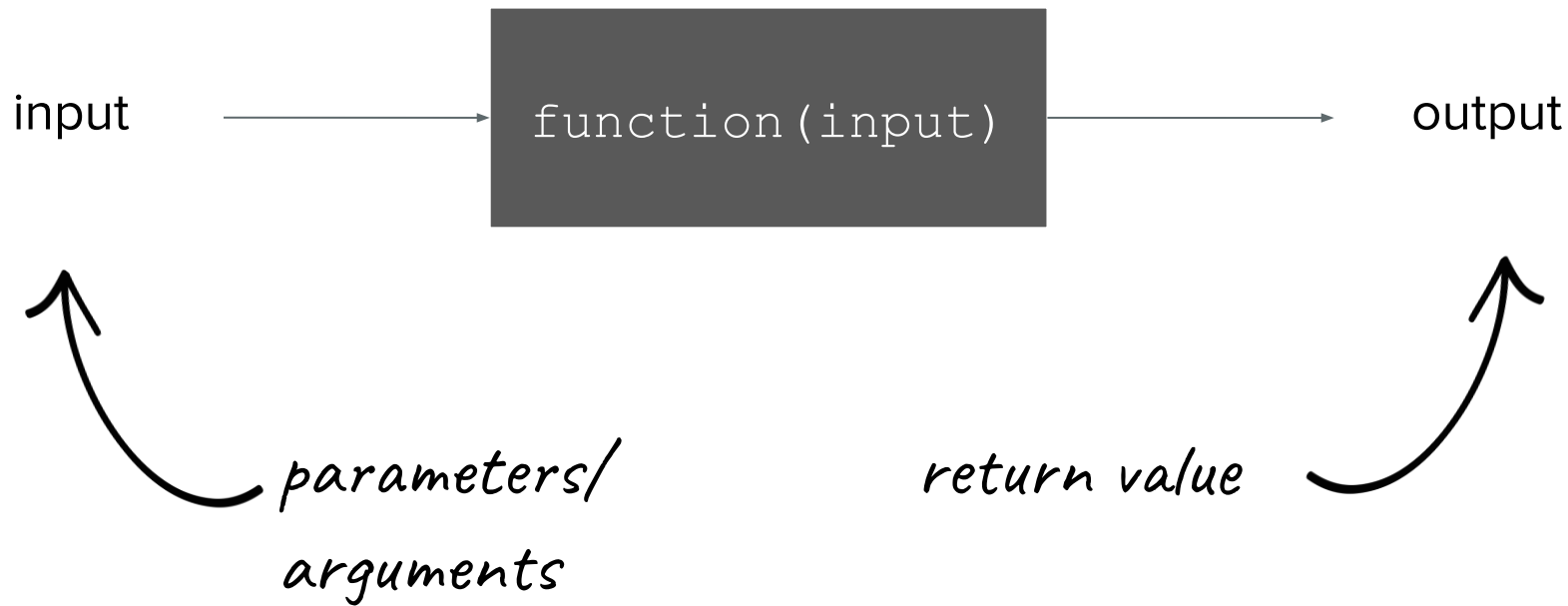
The value that your function hands back to the “calling” function

*return value*



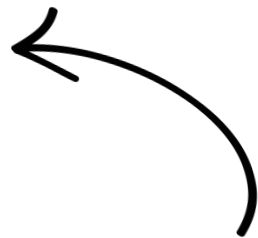


# Anatomy of a function



# Anatomy of a function

```
returnType functionName(varType parameter1, varType parameter2, ...);
```



*function  
prototype*

# Anatomy of a function

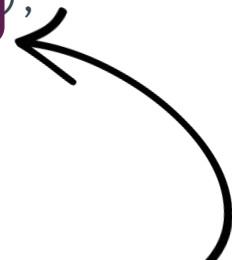
```
returnType functionName(varType parameter1, varType parameter2, ...);
```



*function name*

# Anatomy of a function

```
returnType functionName(varType parameter1, varType parameter2, ...);
```



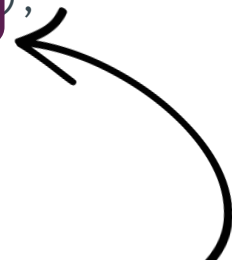
*input expected  
(parameters)*

# Anatomy of a function

```
returnType functionName(varType parameter1, varType parameter2, ...);
```


*Notice that these look very similar to variable declarations! You can think of parameters as a special set of local variables that belong to a function.*

*input expected (parameters)*



# Anatomy of a function

`returnType` `functionName(varType parameter1, varType parameter2, ...);`



*output expected  
(return type)*

*How do you designate a function that doesn't return a value? You can use the special **void** keyword. Note that this type is only applicable for return types, not parameters/variables.*

# Anatomy of a function

```
returnType functionName(varType parameter1, varType parameter2, ...);
```

```
returnType functionName(varType parameter1, varType parameter2, ...) {  
    returnType variable = /* Some fancy code. */  
    /* Some more code to actually do things. */  
    return variable;  
}
```



*function  
definition*

# Anatomy of a function

```
returnType functionName(varType parameter1, varType parameter2, ...);
```

```
returnType functionName(varType parameter1, varType parameter2, ...) {
```

```
    returnType variable = /* Some fancy code. */
```

```
    /* Some more code to actually do things. */
```

```
    return variable;
```

```
}
```



*returned value*



# Function Example

```
double average(double a, double b){  
    double sum = a + b;  
    return sum / 2;  
}
```


```
int main(){  
    double mid = average(10.6, 7.2);  
    cout << mid << endl;  
    return 0;  
}
```

*Order matters! A  
function must always  
be defined before it is  
called.*

Python you can have anything any order that  
you want.

# Function Example

```
double average(double a, double b){  
    double sum = a + b;  
    return sum / 2;  
}
```

  
*callee*  
*(called function)*

```
int main(){  
    double mid = average(10.6, 7.2);  
    cout << mid << endl;  
    return 0;  
}
```

  
*caller*  
*(calling function)*

# Function Example

```
double average(double a, double b) {  
    double sum = a + b;  
    return sum / 2;  
}
```

*parameters*

*return value*

```
int main() {  
    double mid = average(10.6, 7.2);  
    cout << mid << endl;  
    return 0;  
}
```

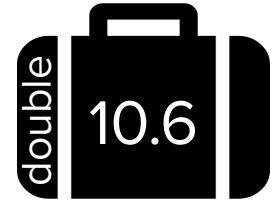
*arguments*

# Function Example

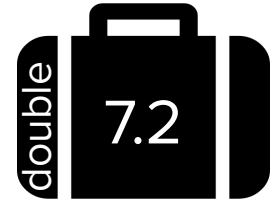
*These variables only  
exist inside average()!*

```
double average(double a, double b){  
    double sum = a + b;  
    return sum / 2;  
}
```

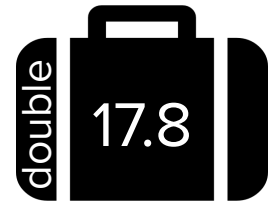
```
int main(){  
    double mid = average(10.6, 7.2);  
    cout << mid << endl;  
    return 0;  
}
```



a



b



sum

# Function Example

```
double average(double a, double b){  
    double sum = a + b;  
    return sum / 2;  
}
```

```
int main(){  
    double mid = average(10.6, 7.2);  
    cout << mid << endl;  
    return 0;  
}
```

*This variable only  
exists inside main()!*





# Pass by Value

```
// C++:
#include<iostream>
using namespace std;

int doubleValue(int x) {
    x *= 2;
    return x;
}

int main() {
    int myValue = 5;
    int result = doubleValue(myValue);

    cout << "myValue: " << myValue << " ";
    cout << "result: " << result << endl;
}
```

*Zoom Poll!*

*What is the console  
output of this block of  
code?*



# Pass by Value

```
// C++:
#include<iostream>
using namespace std;

int doubleValue(int x) {
    x *= 2;
    return x;
}

int main() {
    int myValue = 5;
    int result = doubleValue(myValue);

    cout << "myValue: " << myValue << " ";
    cout << "result: " << result << endl;
}
```

**myValue: 5 result: 10**

*Why is this the case?*

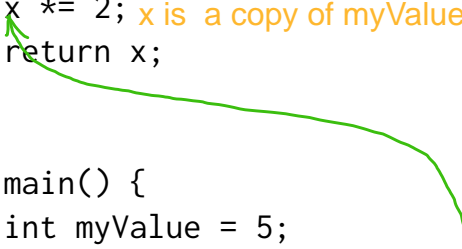
# Pass by Value

```
// C++:
#include<iostream>
using namespace std;

int doubleValue(int x) {
    x *= 2; x is a copy of myValue.
    return x;
}

int main() {
    int myValue = 5;
    int result = doubleValue(myValue);

    cout << "myValue: " << myValue << " ";
    cout << "result: " << result << endl;
}
```



- The reason for the output is that the parameter **x** was passed to the **doubleValue** function *by value*, meaning that the variable **x** is a copy of the variable passed in. Changing it inside the function does *not* change the value in the calling function.
- Pass-by-value is the default mode of operation when it comes to parameters in C++
- C++ also supports a different, more nuanced way of passing parameters – we will see this tomorrow!

细致入微





# Announcements

- Complete the [C++ survey](#) and help us plan tomorrow's lecture!
- Fill out your section time preferences by today at 5pm PDT.
  - Make sure to check what time you've been assigned tomorrow morning.
- Finish [Assignment 0](#) by Wednesday at 11:59 pm local time.
  - If you're running into issues with Qt Creator, come to the Qt Installation Help Session tonight at 7pm PDT. Join the [QueueStatus here](#) to get help.
- Assignment 1 will be released later today, and after this lecture is over, you will have the skills you need to get started on the first part!
  - There be a YEAH (Your Early Assignment Help) session held from 6-7pm PDT on Wednesday evening to help folks get started on the assignment.



# Control Flow



# Boolean Expressions

Expression	Meaning
------------	---------

<code>a &lt; b</code>	<code>a</code> is less than <code>b</code>
-----------------------	--------------------------------------------

<code>a &lt;= b</code>	<code>a</code> is less than or equal to <code>b</code>
------------------------	--------------------------------------------------------

<code>a &gt; b</code>	<code>a</code> is greater than <code>b</code>
-----------------------	-----------------------------------------------

<code>a &gt;= b</code>	<code>a</code> is greater than or equal to <code>b</code>
------------------------	-----------------------------------------------------------

<code>a == b</code>	<code>a</code> is equal to <code>b</code>
---------------------	-------------------------------------------

<code>a != b</code>	<code>a</code> is not equal to <code>b</code>
---------------------	-----------------------------------------------

Operator	Meaning
----------	---------

<code>a &amp;&amp; b</code>	Both <code>a</code> AND <code>b</code> are <b>true</b>
-----------------------------	--------------------------------------------------------

<code>a    b</code>	Either <code>a</code> OR <code>b</code> are <b>true</b>
---------------------	---------------------------------------------------------

<code>!a</code>	If <code>a</code> is <b>true</b> , returns <b>false</b> , and vice-versa
-----------------	--------------------------------------------------------------------------

# Conditional Statements

- The C++ **if** statement tests a boolean expression and runs a block of code if the expression is **true**, and, optionally, runs a different block of code if the expression is **false**. The **if** statement has the following format:

```
○ if (expression) {  
    statements if expression is true  
} else {  
    statements if expression is false  
}
```

*Note: The parentheses around expression are **required**.*



- In Python, a block is defined as an indentation level, where *whitespace* is important. C++ does not have any whitespace restrictions, so blocks are denoted with curly braces, { to begin a block, and } to end a block.
- Blocks are used primarily for conditional statements, functions, and loops.

# Conditional Statements

- The C++ **if** statement tests a boolean expression and runs a block of code if the expression is **true**, and, optionally, runs a different block of code if the expression is **false**. The **if** statement has the following format:

- ```
if (expression) {  
    statements if expression is true  
} else {  
    statements if expression is false  
}
```

- Additional else if statements can be used to check for additional conditions as well

- ```
if (expression1) {  
    statements if expression1 is true  
} else if (expression2) {  
    statements if expression2 is true  
} else {  
    statements if neither expression1 nor expression2 is true  
}
```

# while loops

- Loops allow you to repeat the execution of a certain block of code multiple times
- **while** loops are great when you want to continue executing something until a certain condition is met and you don't know exactly how many times you want to iterate for

```
while (expression) {  
    statement;  
    statement;  
    ...  
}
```



*Execution continues until  
expression evaluates to **false***


# while loops

- Loops allow you to repeat the execution of a certain block of code multiple times
- **while** loops are great when you want to continue executing something until a certain condition is met and you don't know exactly how many times you want to iterate for

```
while (expression) {  
    statement;  
    statement;  
    ...  
}
```



```
int i = 0;  
while (i < 5) {  
    cout << i << endl;  
    i++;  
}
```



Output:

0  
1  
2  
3  
4

Note: The `i++` increments the variable `i` by 1, and is the reason C++ got its name! (and there is a corresponding decrement operator, `--`, as in `i--`).



# for loops

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code
- **for** loop syntax in C++ can look a little strange, let's investigate!



# for loops

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code

```
for (initializationStatement; testExpression; updateStatement) {  
    statement;  
    statement;  
    ...  
}
```




The **initializationStatement** happens at the beginning of the loop, and initializes a variable.

E.g., `int i = 0.`

# for loops

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code

```
for (initializationStatement; testExpression; updateStatement) {  
    statement;  
    statement;  
    ...  
}
```



The **testExpression** is evaluated initially, and after each run through the loop, and if it is **true**, the loop continues for another iteration.

E.g.,  $i < 3$ .

# for loops

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code

```
for (initializationStatement; testExpression; updateStatement) {  
    statement;  
    statement;  
    ...  
}
```

The **updateStatement** happens after each loop, but *before* **testExpression** is evaluated.

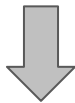
E.g., `i++`.



# for loops

- **for** loops are great when you have a known, fixed number of times that you want to execute a block of code

```
for (initializationStatement; testExpression; updateStatement) {  
    statement;  
    statement;  
    ...  
}
```



```
for (int i = 0; i < 3; i++) {  
    cout << i << endl;  
}
```

Output:

0

1

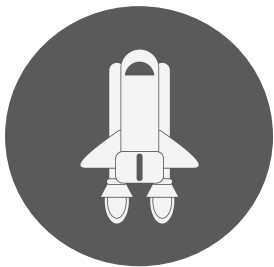
2



# Interactive Example

# Try it for yourself!

Write a program that prints out the calls for a spaceship that is about to launch. Countdown the numbers from 10 to 1 and then write “Liftoff.”



```
10
9
8
7
6
5
4
3
2
1
Liftoff
```

```
def main():
    for i in range(10, 0, -1):
        print(i)
    print ("Liftoff")

if __name__ == "__main__":
    main()
```

Python

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

int main() {
    /* TODO: Your code goes here! */
    for (int i = 10; i > 0; i--){
        cout << i << endl;
    }
    cout << "Liftoff" << endl;
    return 0;
}
```

C++



# How do we test code in CS106B?



# Testing

大教堂

Software and cathedrals are much the same – first we build them,  
then we pray.

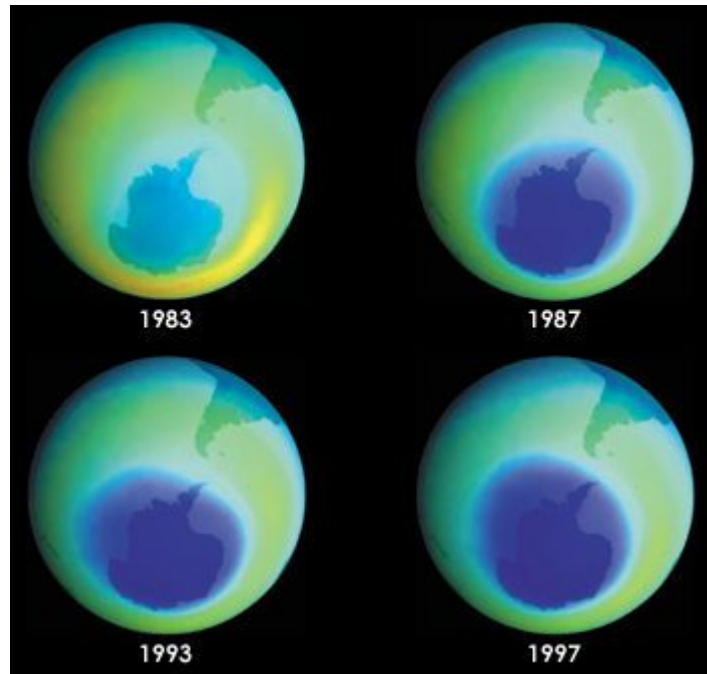
– Sam Redwine



# Why is testing important?

The hole in the ozone layer over Antarctica remained undetected for a long period of time because the data analysis software used by NASA in its project to map the ozone layer had been **designed to ignore values that deviated greatly from expected measurements.**

偏离



# Why is testing important?



In 1996, a European Ariane 5 rocket was set to deliver a payload of satellites into Earth orbit, but problems with the software caused the launch rocket to veer off its path a mere 37 seconds after launch. The problem was the result of **code reuse from the launch system's predecessor, Ariane 4, which had very different flight conditions from Ariane 5.**

# Why is testing important?

A 2002 study commissioned by the National Institute of Standards and Technology (referred to here) **found that software bugs cost the U.S. economy \$59.5 billion every year** (imagine the global costs...). The study estimated that more than a third of that amount, \$22.2 billion, **could be eliminated by improved testing.**





# Why is testing important?

- Testing can save money
- Testing can save lives
- Testing can prevent disasters
- **Testing is a programmer's responsibility.**
  - You must think about ethical considerations when you develop code that impacts people.

# What are good testing strategies?

- Write tests that cover a wide variety of use cases for your function!
  - Use your critical thinking and analysis skills to identify a diverse range of possible ways in which your code might be used.



# Testing strategies

- Write tests that cover a wide variety of use cases for your function!  
*For example, if your function takes in an integer parameter, test what happens if the value that is passed in negative, zero, a large positive number, etc!*
- Consider:
  - Basic use cases
  - Edge cases

## Definition

### edge case

Uses of your function/program that represent extreme situations



# SimpleTest



# What is SimpleTest?

- SimpleTest is a C++ library developed by some of the lecturers here at Stanford that allows standalone, C++ unit testing
- For those of you coming from CS106A in Python, this is similar in functionality to the **doctest** infrastructure that you learned
- We will see SimpleTest a lot this quarter! You will learn how to write good, comprehensive suites of tests using this library, starting from the very first assignment.





How does SimpleTest work?

CS106B Testing Guide

**– make sure to read it!**

# How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "all-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```



NO\_TESTS  
SELECTED\_TESTS  
ALL\_TESTS



# How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "all-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```

all-examples.cpp

```
#include "testing/SimpleTest.h"

int factorial (int num);

int factorial (int num){
    /* Implementation here */
}

PROVIDED_TEST("Some provided tests."){
    EXPECT_EQUAL(factor(12), 16);
    EXPECT(isPerfect(6));
    EXPECT(!isPerfect(12));
}

STUDENT_TEST("student wrote this test"){
    // student tests go here!
}
```

?

.



# How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "all-examples.h"

int main()
{
    if (runSimpleTests(SELECTED_TESTS)) {
        return 0;
    }

    return 0;
}
```

all-examples.cpp

```
#include "testing/SimpleTest.h"

int factorial (int num);

int factorial (int num){
    /* Implementation here */
}

PROVIDED_TEST("Some provided tests."){
    EXPECT_EQUAL(factorial(1), 1);
    EXPECT_EQUAL(factorial(2), 2);
    EXPECT_EQUAL(factorial(3), 6);
    EXPECT_EQUAL(factorial(4), 24);
}

STUDENT_TEST("student wrote this test"){
    // student tests go here!
}
```



# How does SimpleTest work?

main.cpp

```
#include "testing/SimpleTest.h"
#include "all-examples.cpp"
```

```
int main()
{
    if (runSimpleTest())
        return 0;

    return 0;
}
```

all-examples.cpp

```
#include "testing/SimpleTest.h"
```

```
(int num);
```

```
(int num){
    // presentation here */
```

```
"Some provided tests."){
```

```
    EQUAL(factorial(1), 1);
```

```
    EQUAL(factorial(2), 2);
```

```
    EQUAL(factorial(3), 6);
```

```
    EQUAL(factorial(4), 24);
```

```
STUDENT_TEST("student wrote this test"){
    // student tests go here!
}
```



# What's next?

# Strings, Vectors, C++ Review

