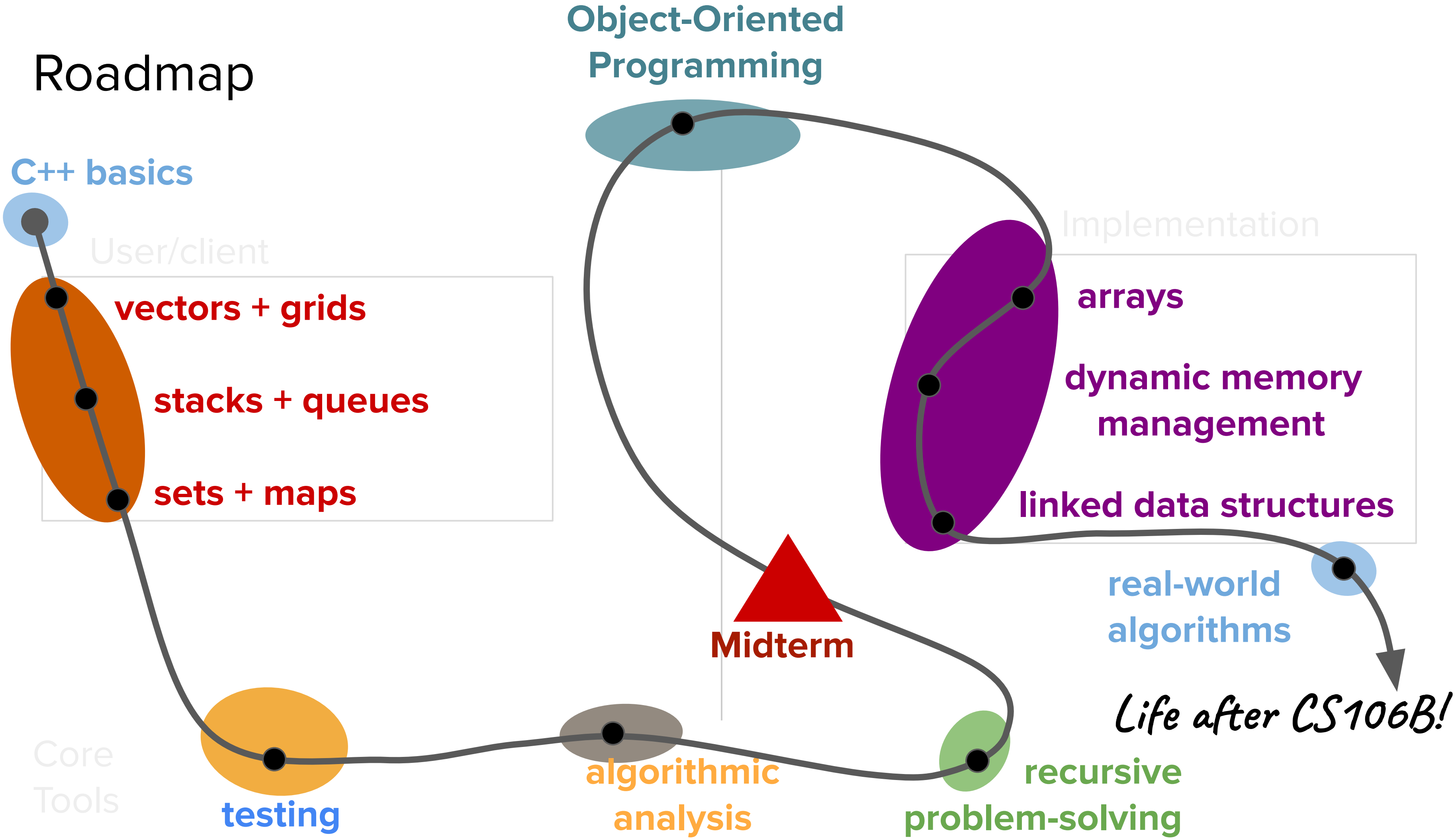
Programming Fundamentals in C++

Hope everyone is taking care today!

Power Outage Updates - everything is tentative

- The **Qt Creator Help Session** will be moved to Jenny's Group OH time this week instead: **Thursday, June 23 at 1:30-3:30pm in Huang 019**.
- **There will be no sections or LaIR today.** When the cs198.stanford.edu website is back up, we will extend the deadline for section sign-ups. Tentatively plan on attending a section on Thursday or Friday (regardless of what you end up being assigned), but know that no one will be penalized for missing section this week.
- While everything is down, we are unable to update the course website or receive emails via our @cs.stanford.edu addresses. **If you need to contact us privately about something, please use a private Ed post instead.**

Roadmap



C++ basics

User/client

vectors + grids

stacks + queues

sets + maps

Core Tools

testing

algorithmic analysis

recursive problem-solving

Midterm

Object-Oriented Programming

Implementation

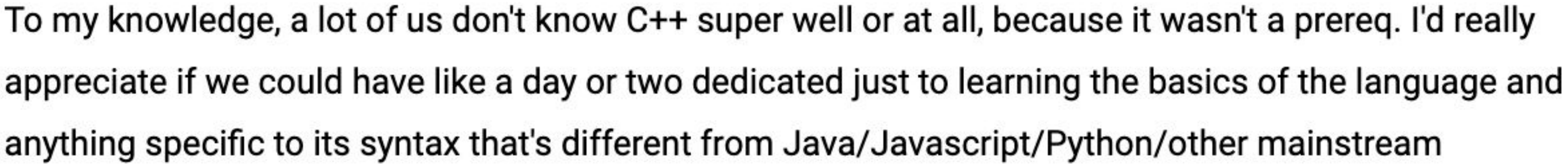
arrays

dynamic memory management

linked data structures

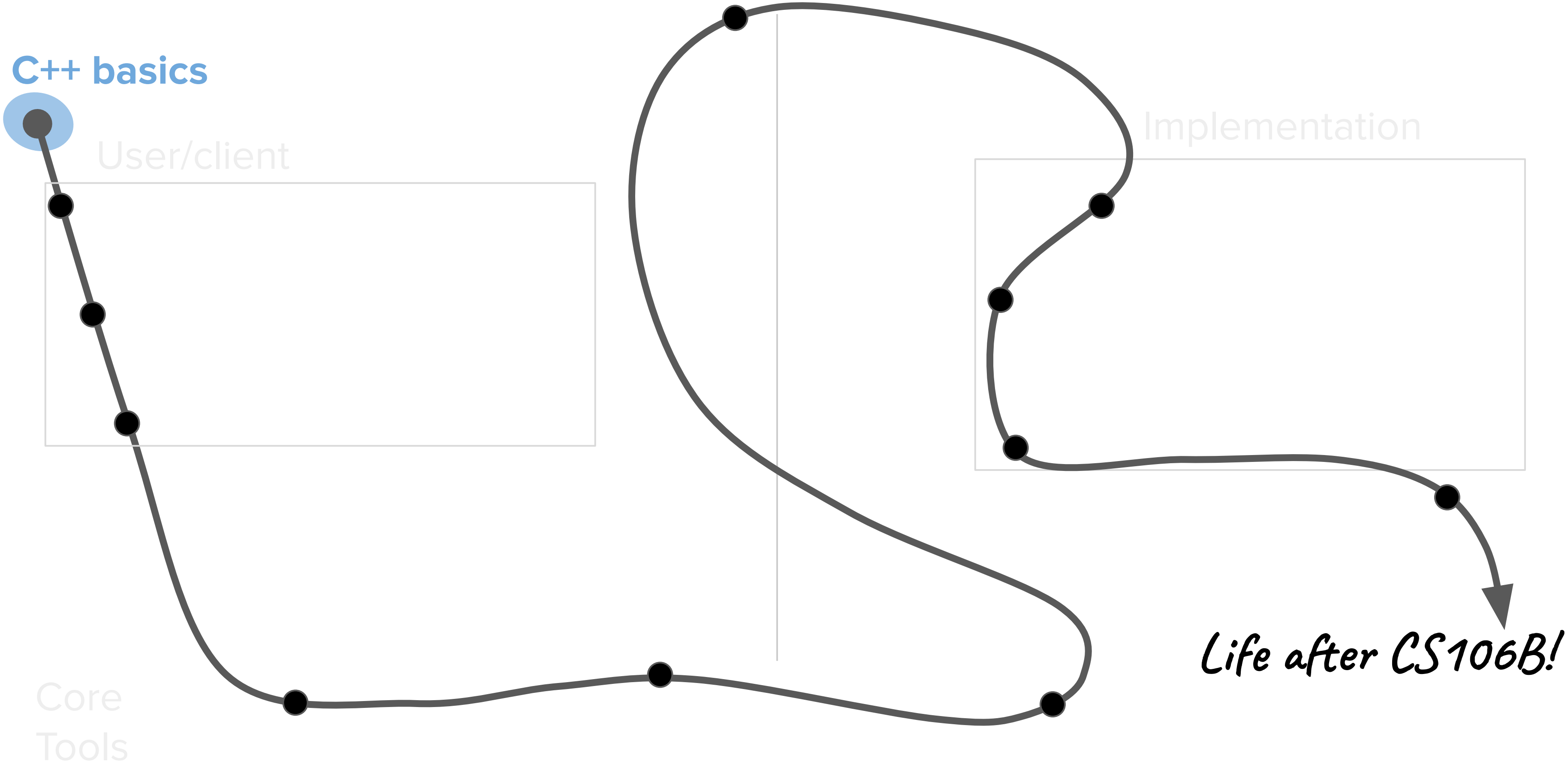
real-world algorithms

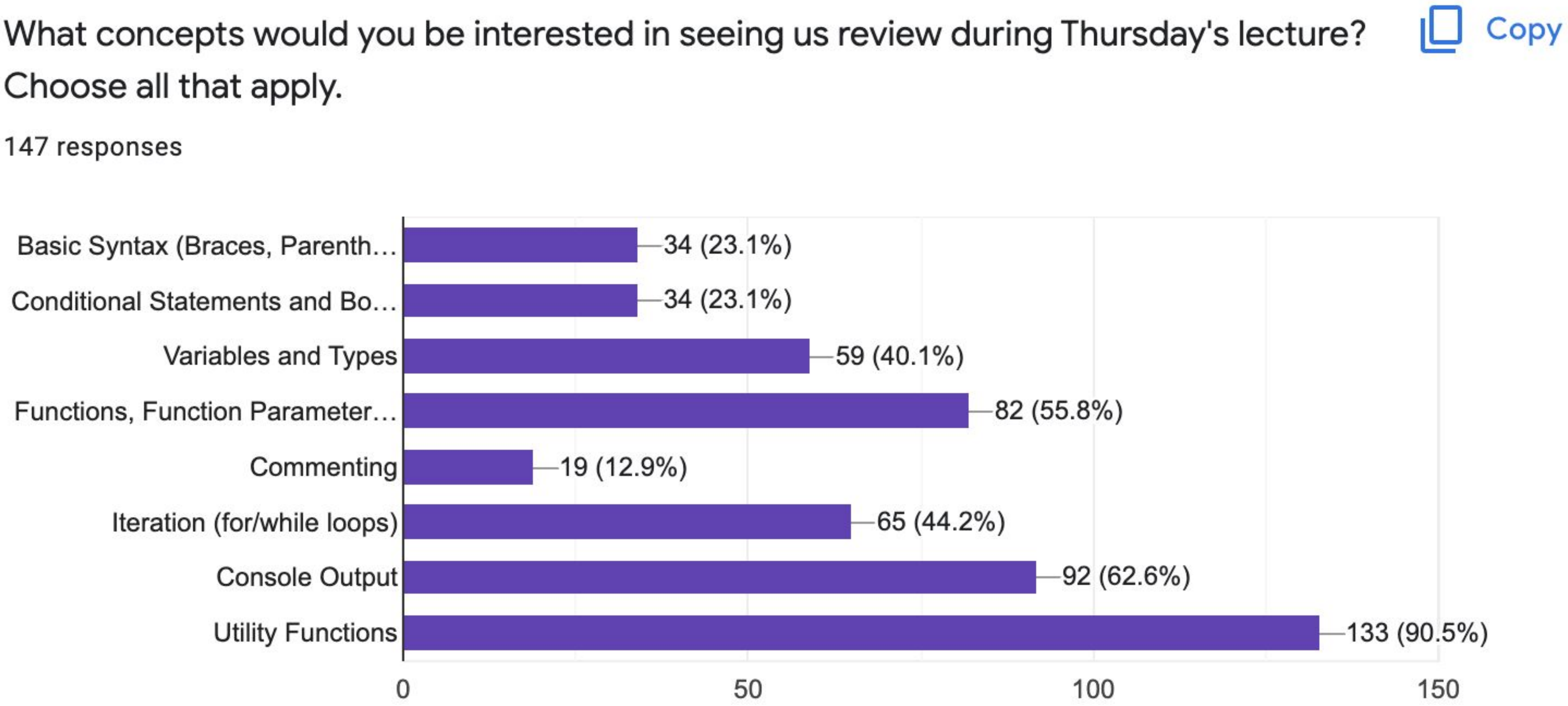
Life after CS106B!



To my knowledge, a lot of us don't know C++ super well or at all, because it wasn't a prereq. I'd really appreciate if we could have like a day or two dedicated just to learning the basics of the language and anything specific to its syntax that's different from Java/Javascript/Python/other mainstream

Today...


C++ basics
User/client
Implementation
Core
Tools
Life after CS106B!



What concepts would you be interested in seeing us review during Thursday's lecture? Choose all that apply.

147 responses

Today's questions

Why C++?

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

What's next?

Why C++?

How is C++ different from other languages?

- C++ is a compiled language (vs. interpreted)

- C++ is gives us access to lower-level computing resources (e.g. more direct control over computer memory)
  - 10 times faster than python!

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

The structure of a program

```
#include <iostream>
#include "console.h"
using namespace std;

// The C++ compiler will look for a function
// called “main”
int main() {
   cout << "Hello, world!" << endl;
   return 0;  // must return an int to indicate
              // successful program completion
}
```

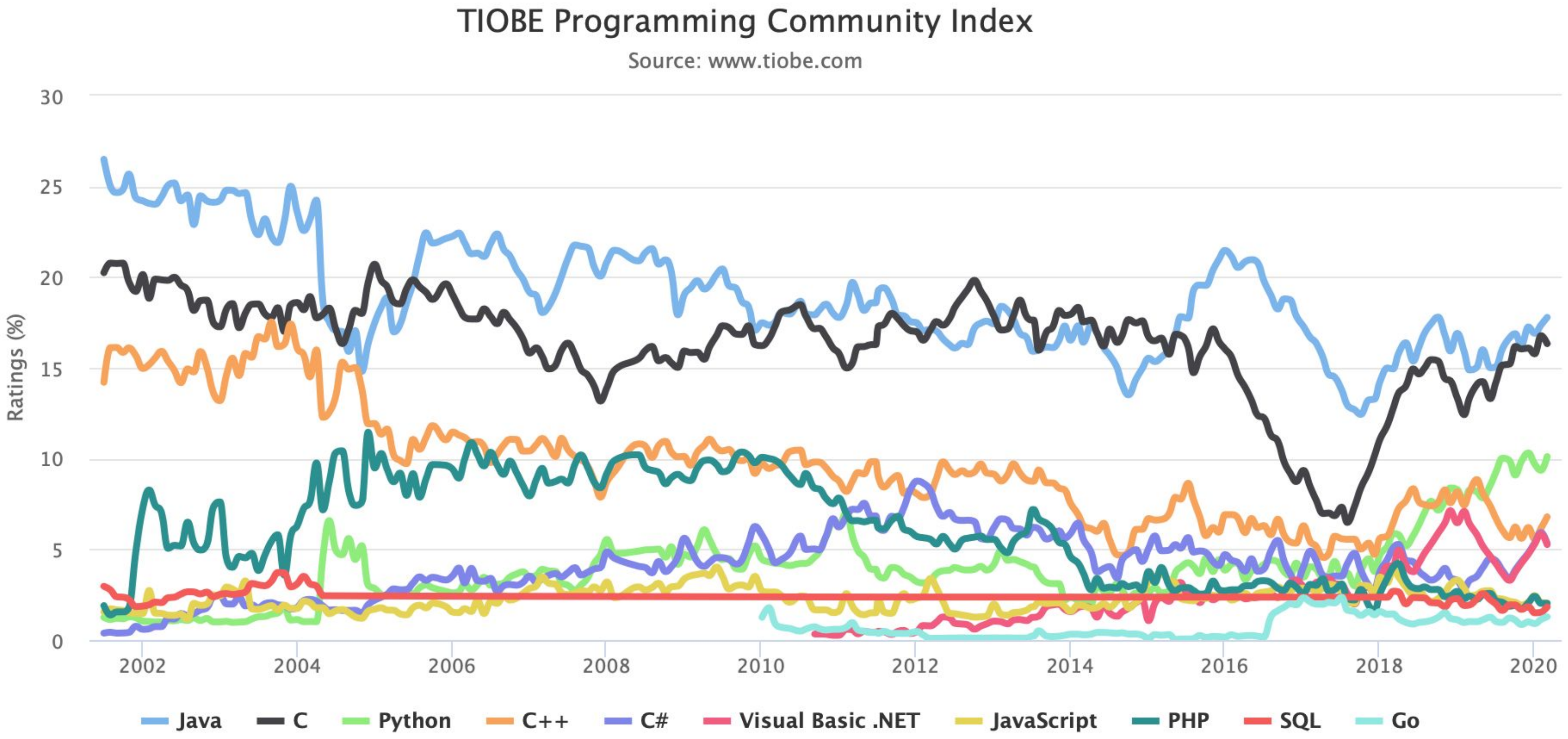
C++

```
import sys


# This function does not need to be called “main”
def main():
   print('Hello, world!')


if __name__ == '__main__':
    # Any function that gets placed here will get
    # called when you run the program with
    # `python3 helloworld.py`
    main()
```

Python

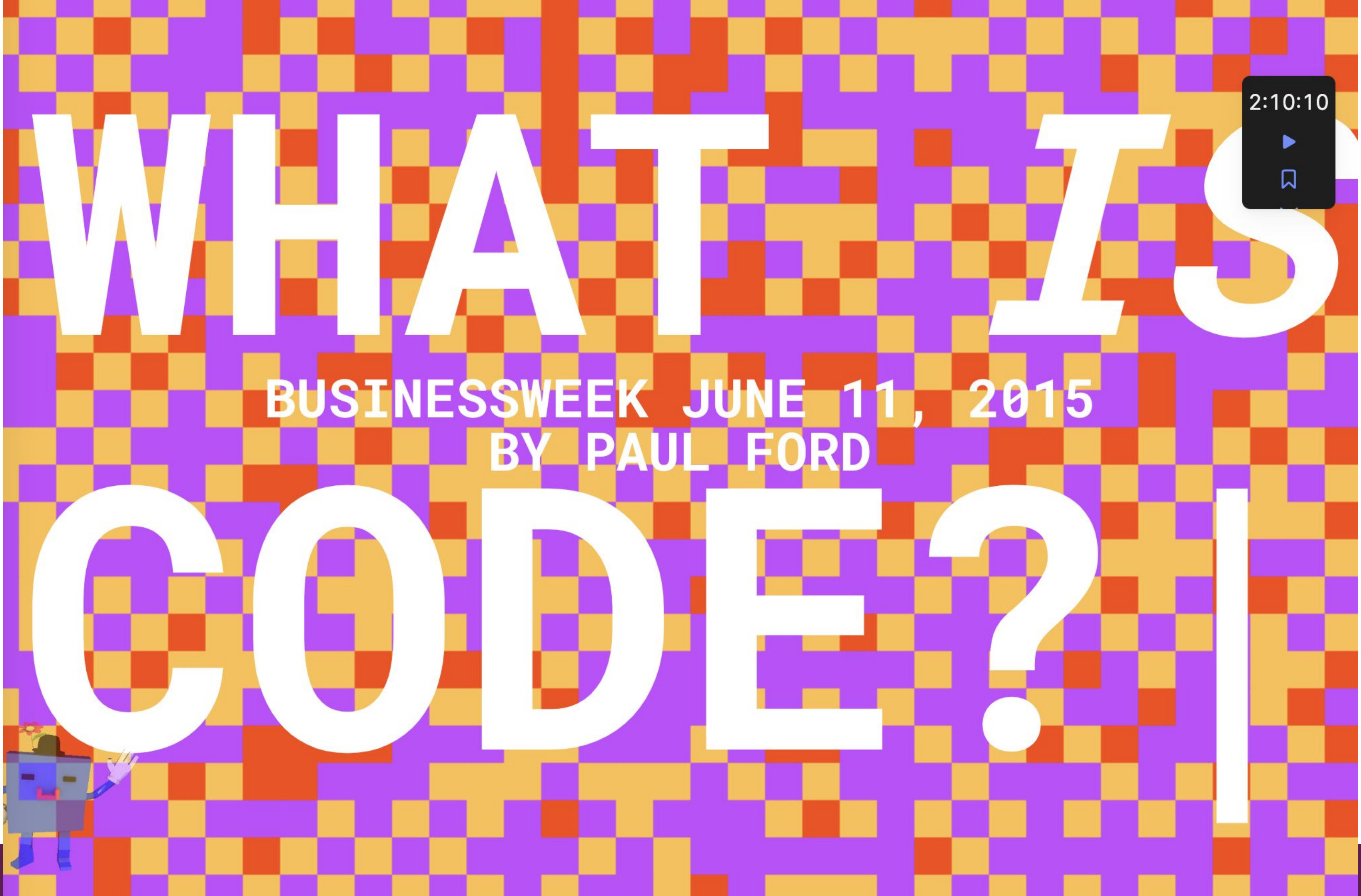
Take a guess

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

TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Java
C
Python
C++
C#
Visual Basic .NET
JavaScript
PHP
SQL
Go

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++



WHAT IS CODE?

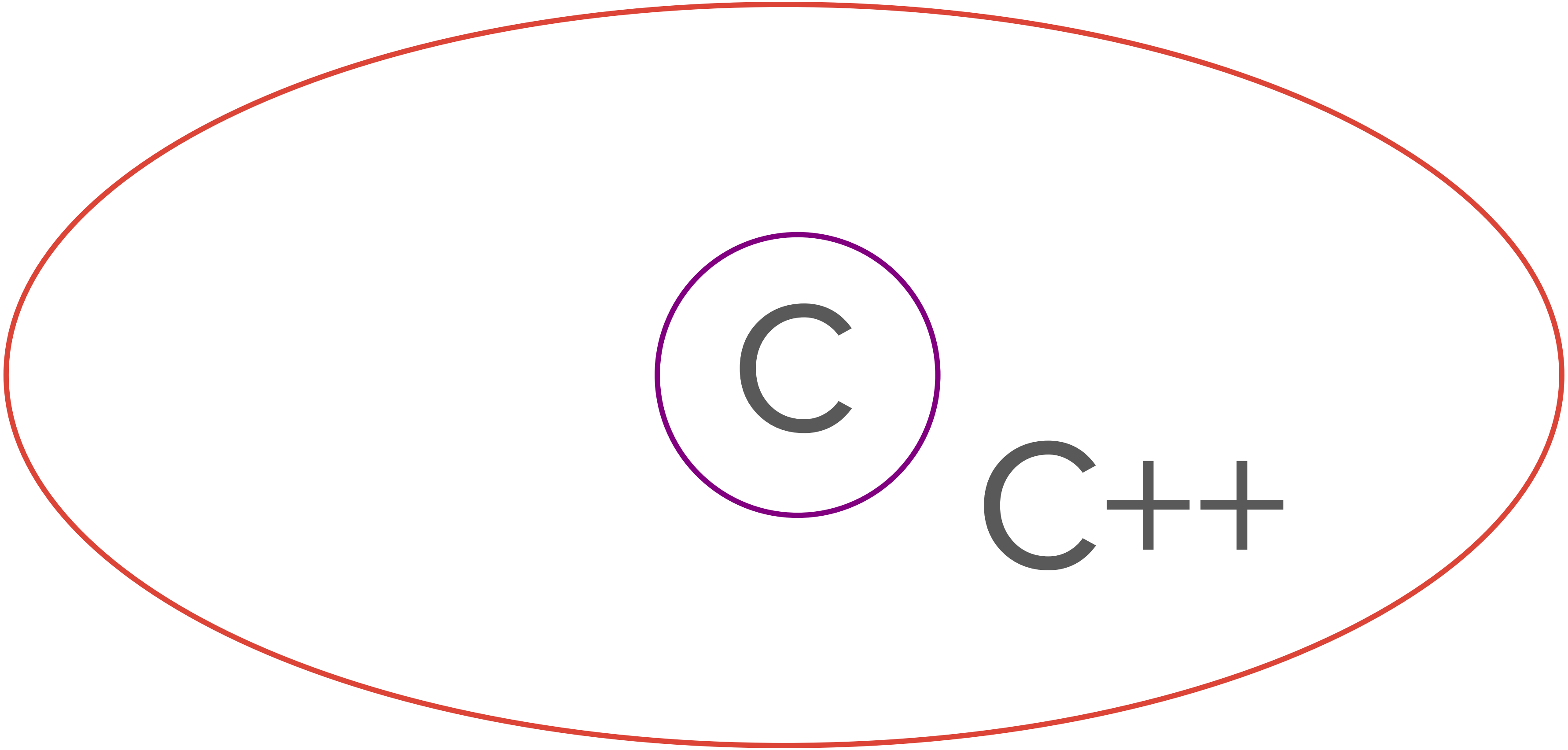
BUSINESSWEEK JUNE 11, 2015
BY PAUL FORD

©

~~B~~

Credit: https://www.ncl.ac.uk/computing/about/history/1970s/

©



C

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++ 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++ 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
    - Even though it's old, it still gets updated every ~3 years

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

“High level”

Python

C++

C

Machine code

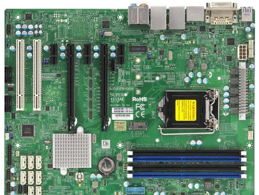
“Low level”

YOUR THOUGHTS / THE REAL WORLD

2+2=4

PROGRAMMING LANGUAGE

```
int sum = 0;
int num_busters = 2;
int num_perrys = 2;
sum = num_busters + num_perrys
```



C++ Benefits and Drawbacks

Benefits

C++ Benefits and Drawbacks

Benefits

- C++ is **fast**
  - Get ready for the Python vs C++ speed showdown during Assignment 1!

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++ 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

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++ 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

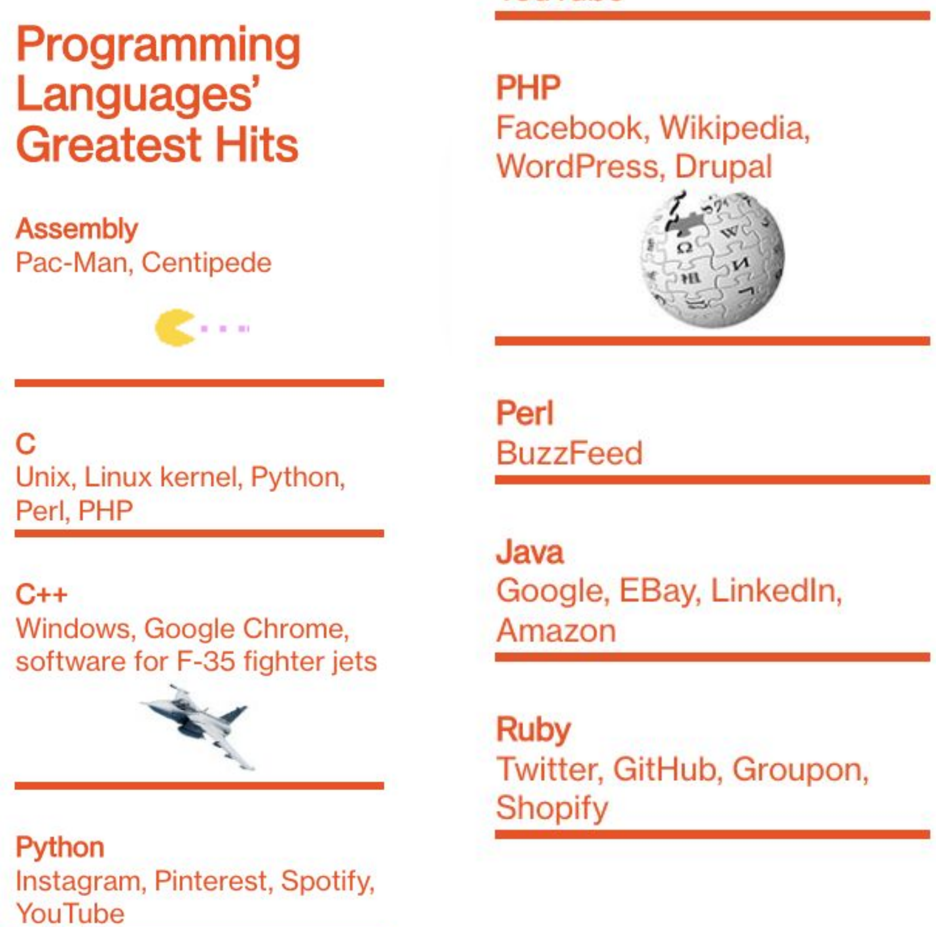
C++ Benefits and Drawbacks

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**
  - C++ will let you make mistakes (especially related to memory)



Programming Languages' Greatest Hits

Assembly
Pac-Man, Centipede

C
Unix, Linux kernel, Python, Perl, PHP

C++
Windows, Google Chrome, software for F-35 fighter jets

Python
Instagram, Pinterest, Spotify, YouTube

PHP
Facebook, Wikipedia, WordPress, Drupal

Perl
BuzzFeed

Java
Google, EBay, LinkedIn, Amazon

Ruby
Twitter, GitHub, Groupon, Shopify

Credit: Paul Ford

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

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.

 */
```

Include libraries

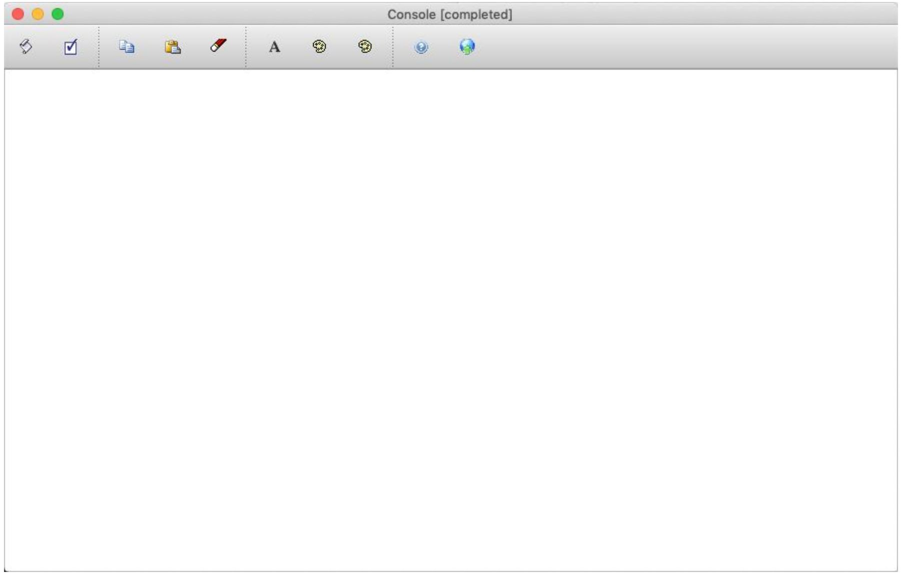
- **What is a library?**
  - It’s a bunch of code that other people have written, packaged up nicely so we can reuse it!
  - In C++, a library includes two files (.h header file, .cpp file)
  - In python, they’re called modules
- **Standard library**
  - Comes with the programming language
- **Anyone can write a library and publish it**
  - CS106
  - You could write a library!
  - Open-source

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 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

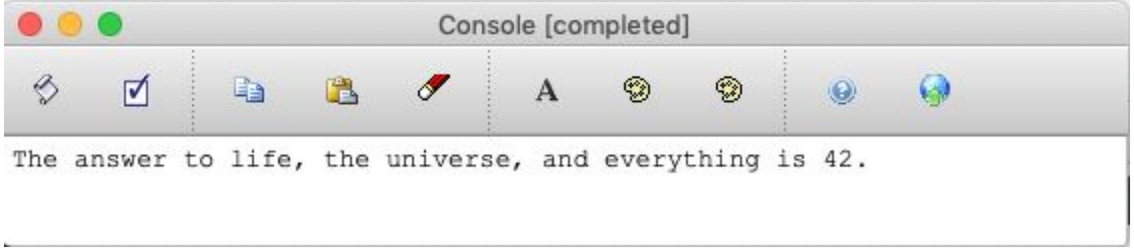
- The console is the main venue that we will use in this class to communicate information from a program to the user of the program.



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;
```

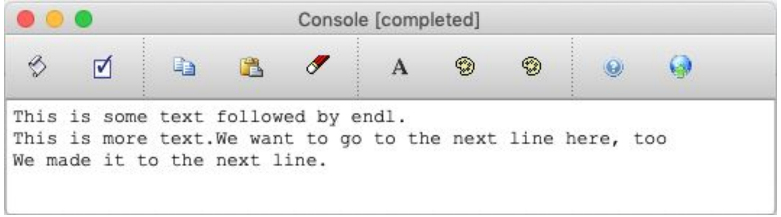


```
The answer to life, the universe, and everything is 42.
```

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;
```

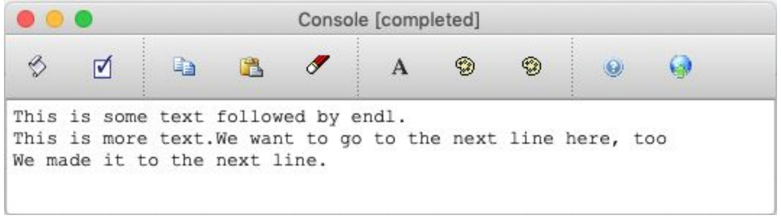


```
This is some text followed by endl.
This is more text.We want to go to the next line here, too
We made it to the next line.
```

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;
```



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

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

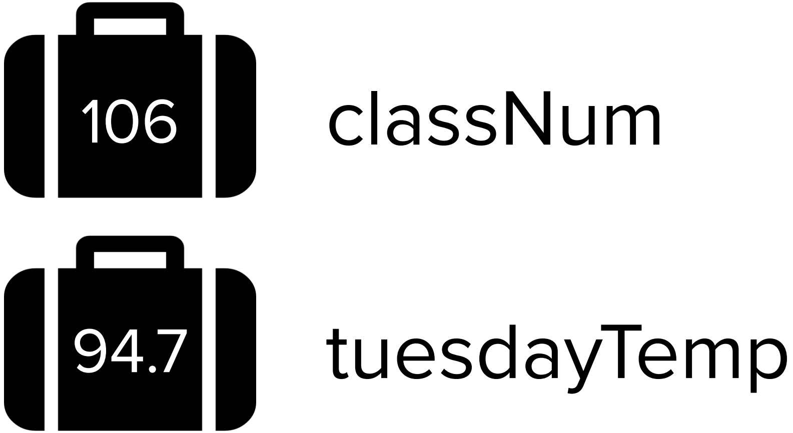
Variables and Types

Variables

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

Variables

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



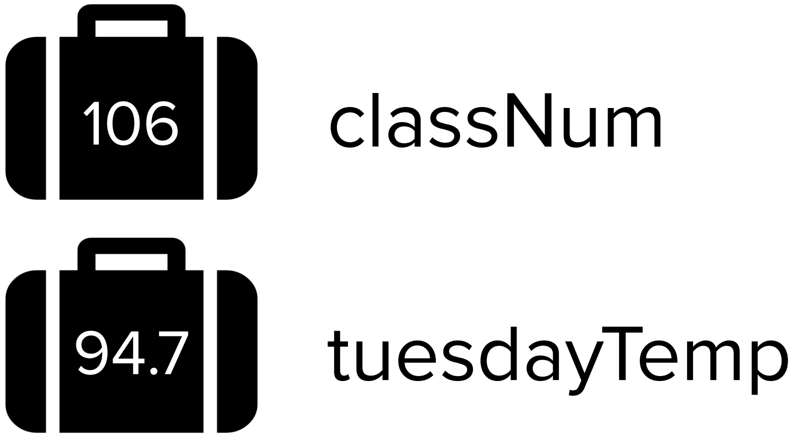
106 classNum

94.7 tuesdayTemp

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.



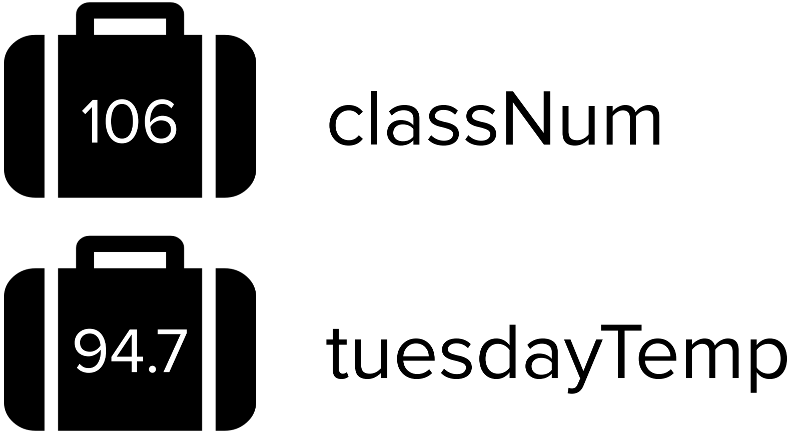
106 classNum

94.7 tuesdayTemp

Variables

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

Note: C++ uses the camelCase naming convention



106 classNum

94.7 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.

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)`

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` (or `long`)
  - `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` (or `long`)
  - `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`** **(**or **`long`)**
  - **`double`**
  - **`string`**
  - **`char`**
- **In C++, *all types must be explicitly defined when the variable is created, and a variable cannot change its type*.**

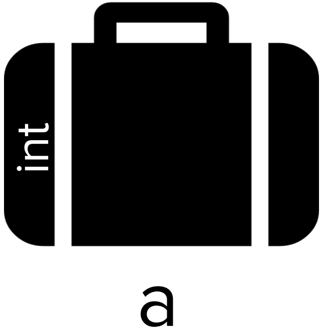
Key takeaway

Types in C++

In C++, all types must be explicitly defined when the variable is created, and a variable cannot change its type.

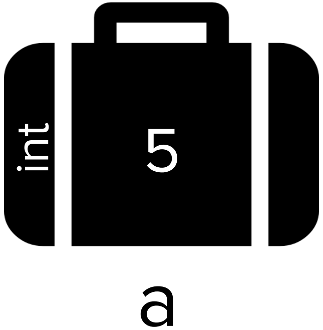
Typed Variables

```
int a; // declare a new integer variable
```



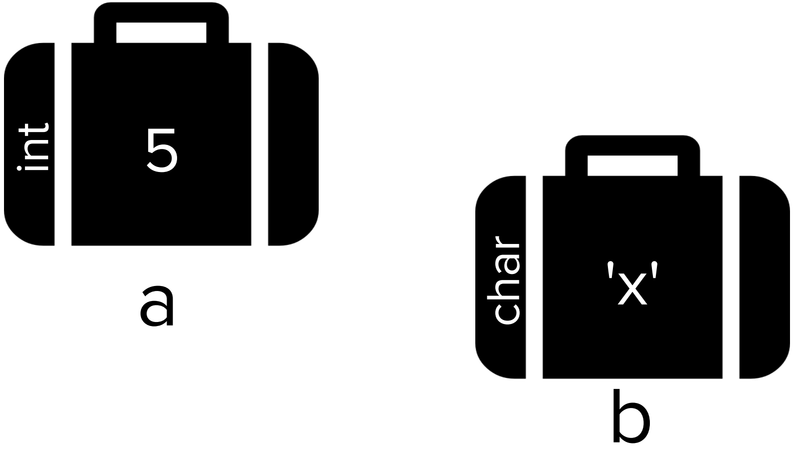
Typed Variables

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



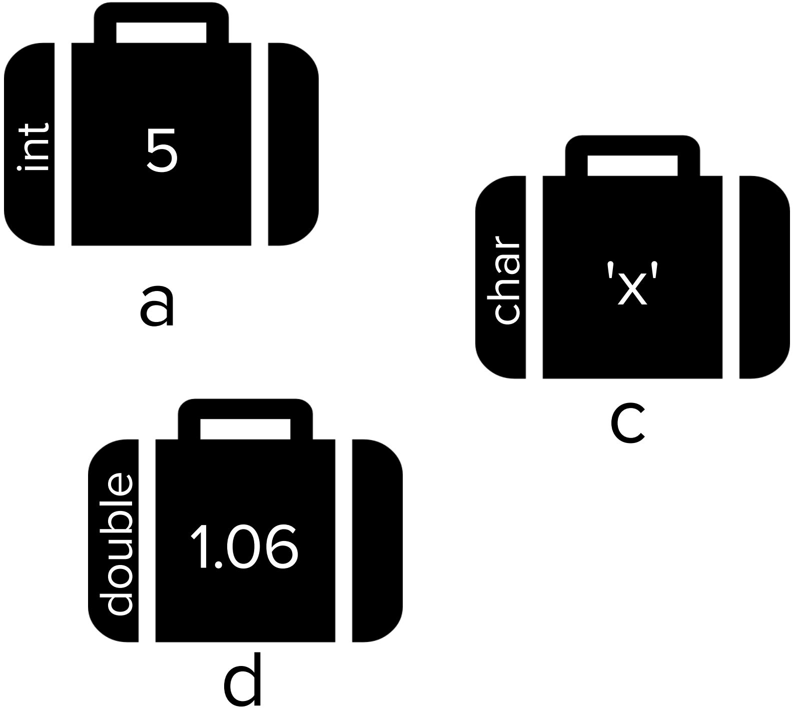
Typed Variables

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

char b = 'x'; // b is a char ("character")
```



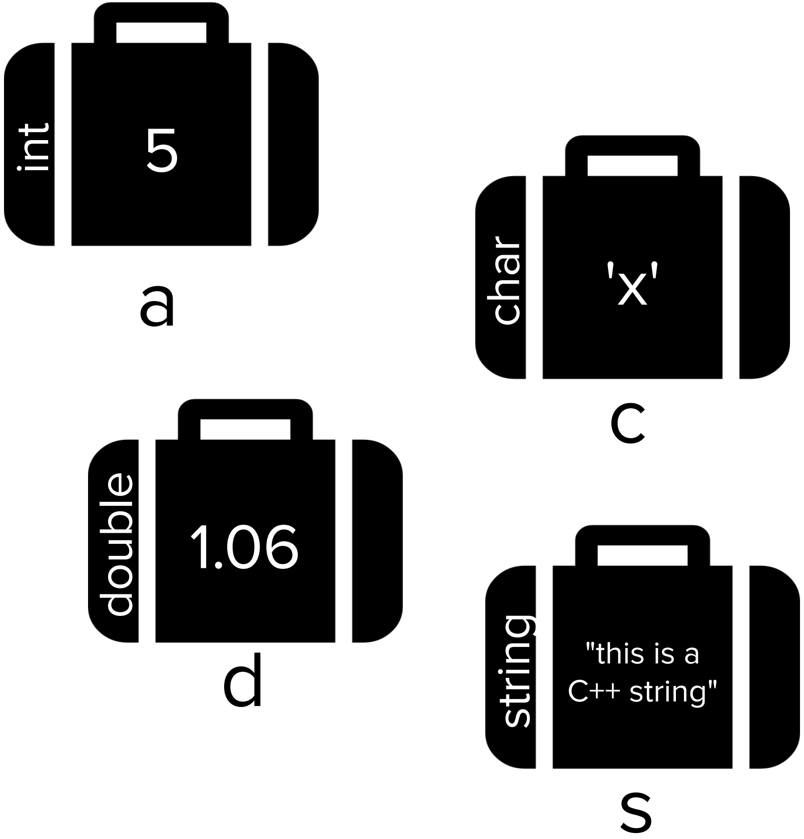
Typed Variables

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

char c = 'x'; // b is a char ("character")
double d = 1.06; // d is a double, a type
used to represent decimal numbers
```



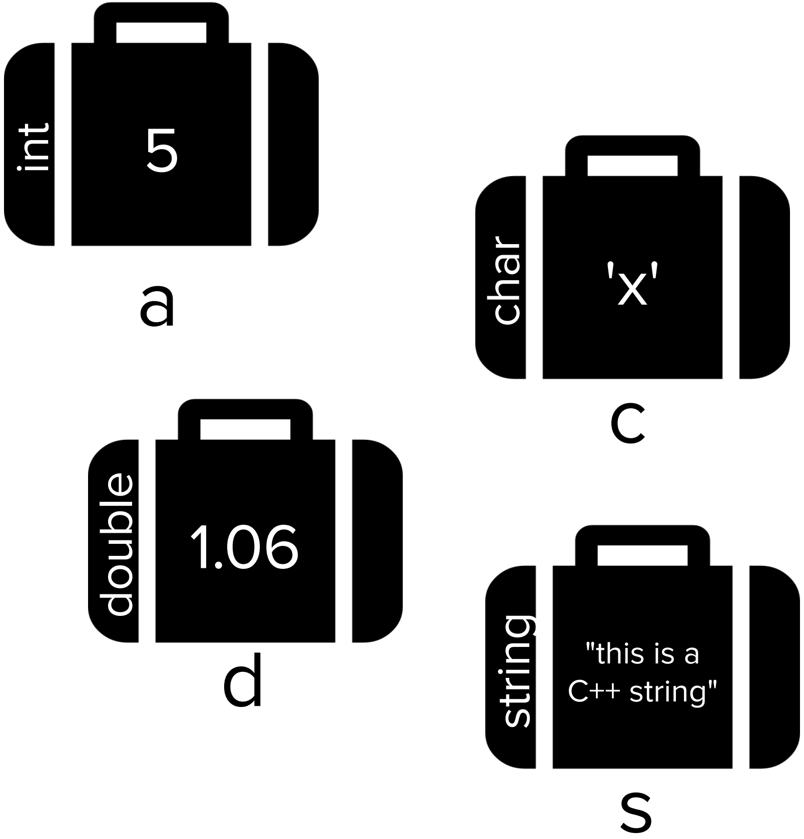
Typed Variables

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

char c = 'x'; // b 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";
```



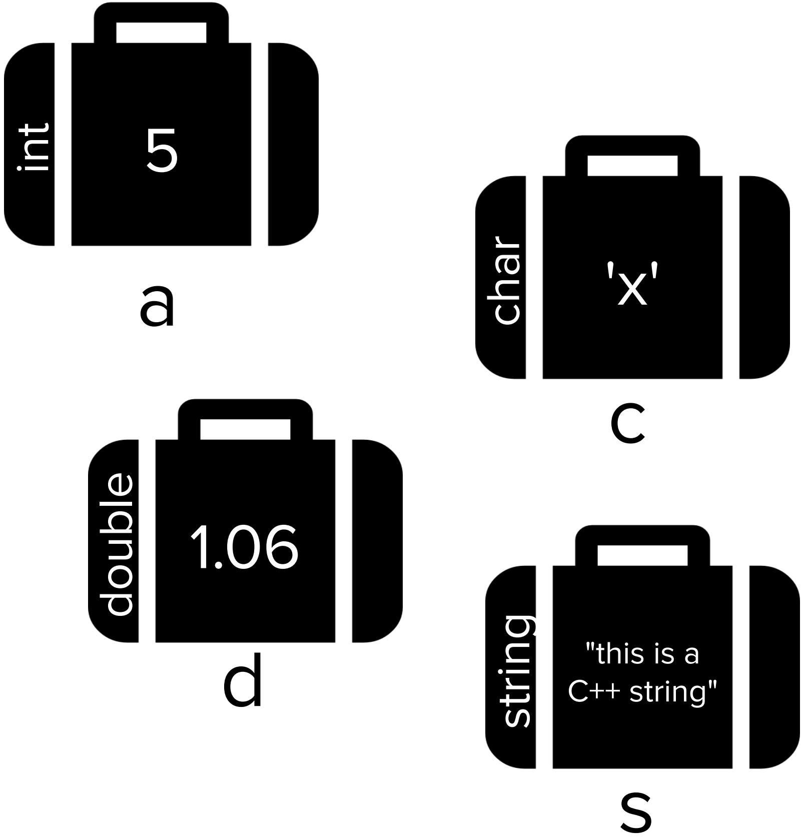
Typed Variables

```
int a; // declare a new integer variable
a = 5; // initialize the variable value
char c = 'x'; // b 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
```



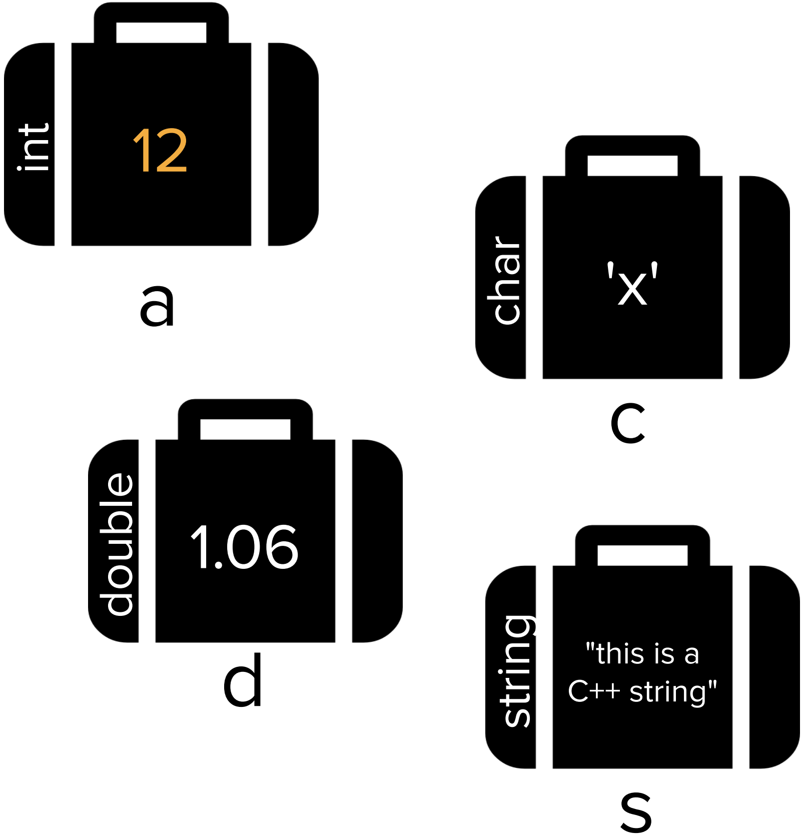
Typed Variables

```
int a; // declare a new integer variable
a = 5; // initialize the variable value
char c = 'x'; // b 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
```



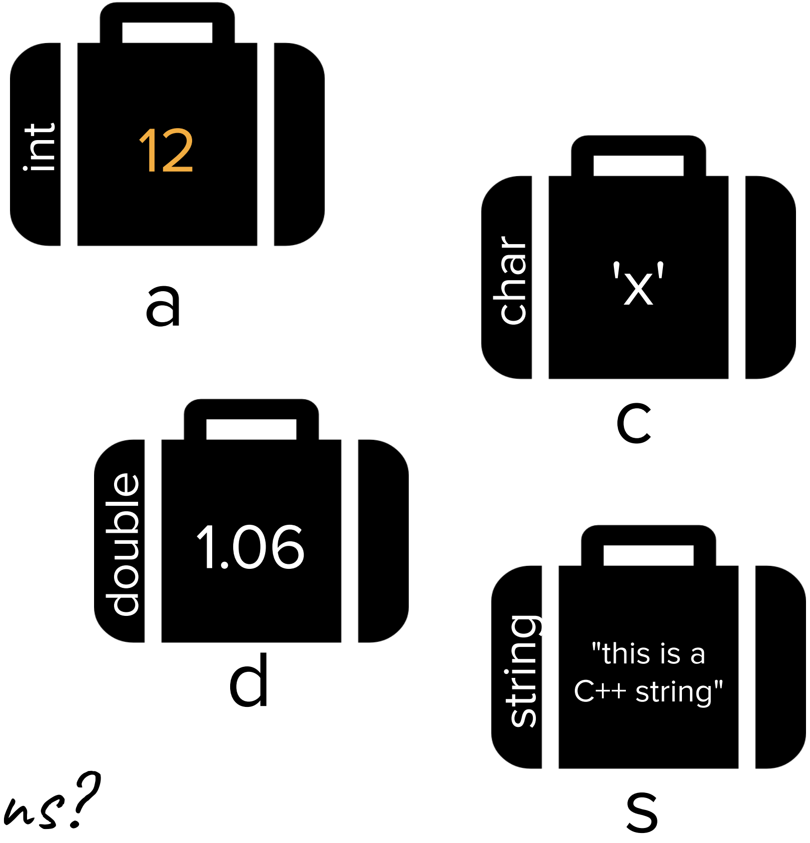
Typed Variables

```
int a; // declare a new integer variable
a = 5; // initialize the variable value
char c = 'x'; // b 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
```



Typed Variables

```
int a; // declare a new integer variable
a = 5; // initialize the variable value
char c = 'x'; // b 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
```



Questions?

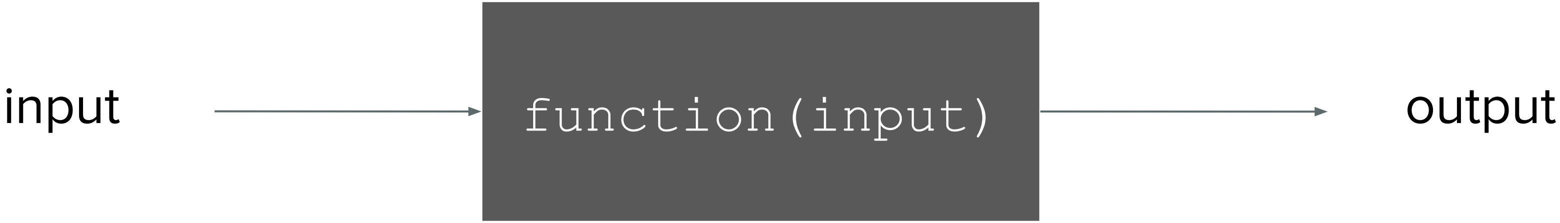
Mid-Lecture Announcements Break!

Announcements

- If you have WiFi and power, finish Assignment 0 by Friday at 11:59 pm PDT.
  - If you're running into issues with Qt Creator, come to the Qt Installation Help Session Thursday.
- Assignment 1 will be released Thursday (tomorrow), and after that lecture is over, you will have the skills you need to get started on pt 1!
  - YEAH hours Friday 12:15 pm Hewlett 102
- **We will be sending a lot of updates on Ed today regarding the status of sections, lecture, and the website.**
- Thanks for being flexible! Stay safe!

Functions and Parameters

Anatomy of a function

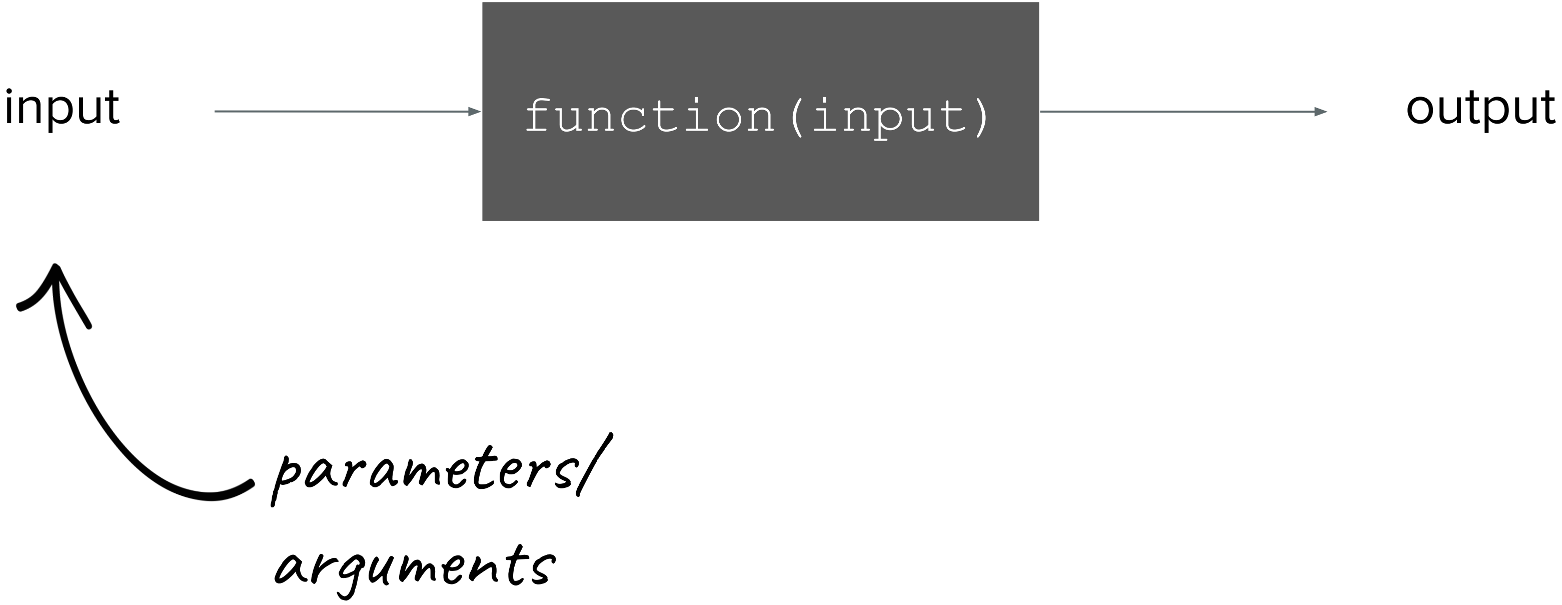


input

`function(input)`

output

Anatomy of a function



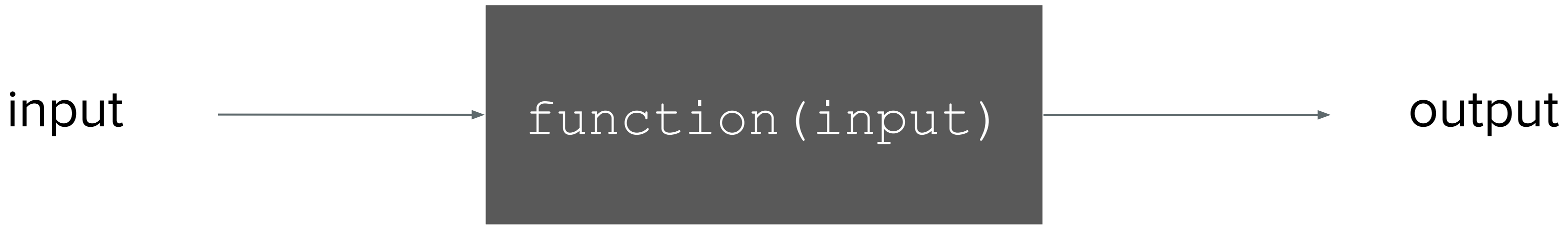
input

`function(input)`

output

parameters/
arguments

Anatomy of a function



input

`function(input)`

output

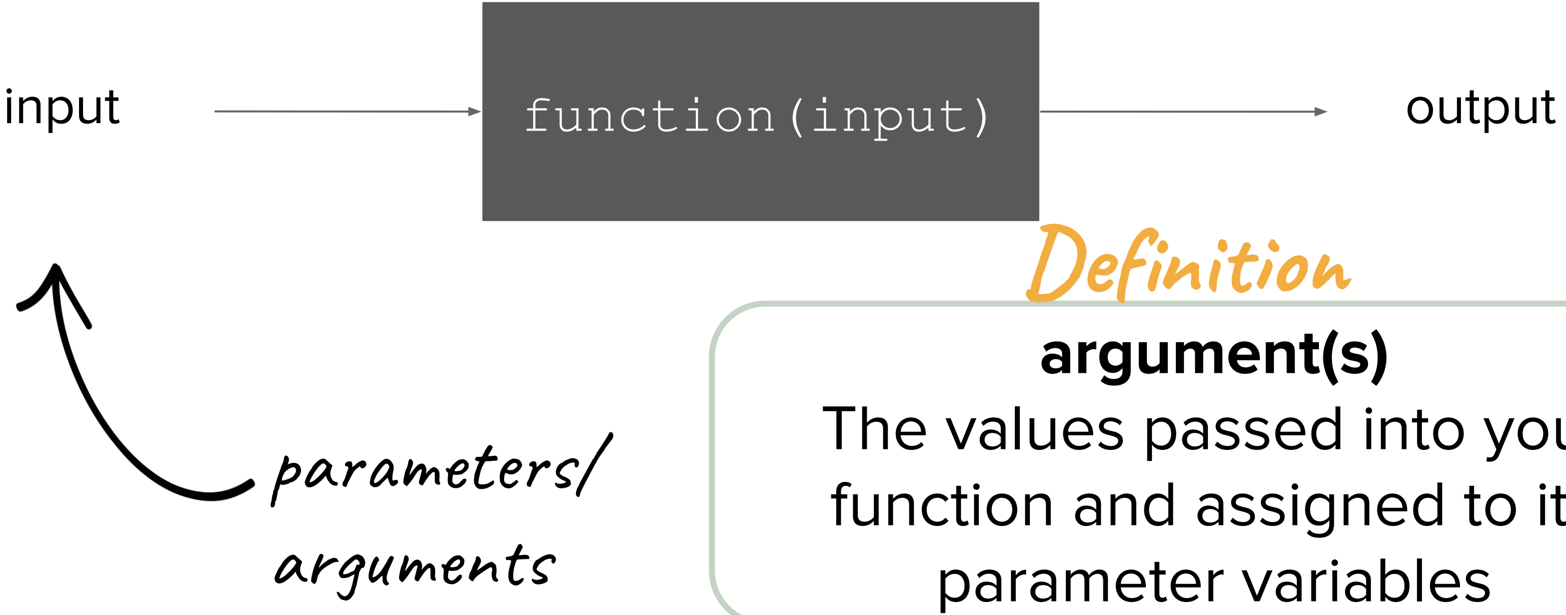
parameters/ arguments

Definition

parameter(s)

One or more variables that your function expects as input

Anatomy of a function

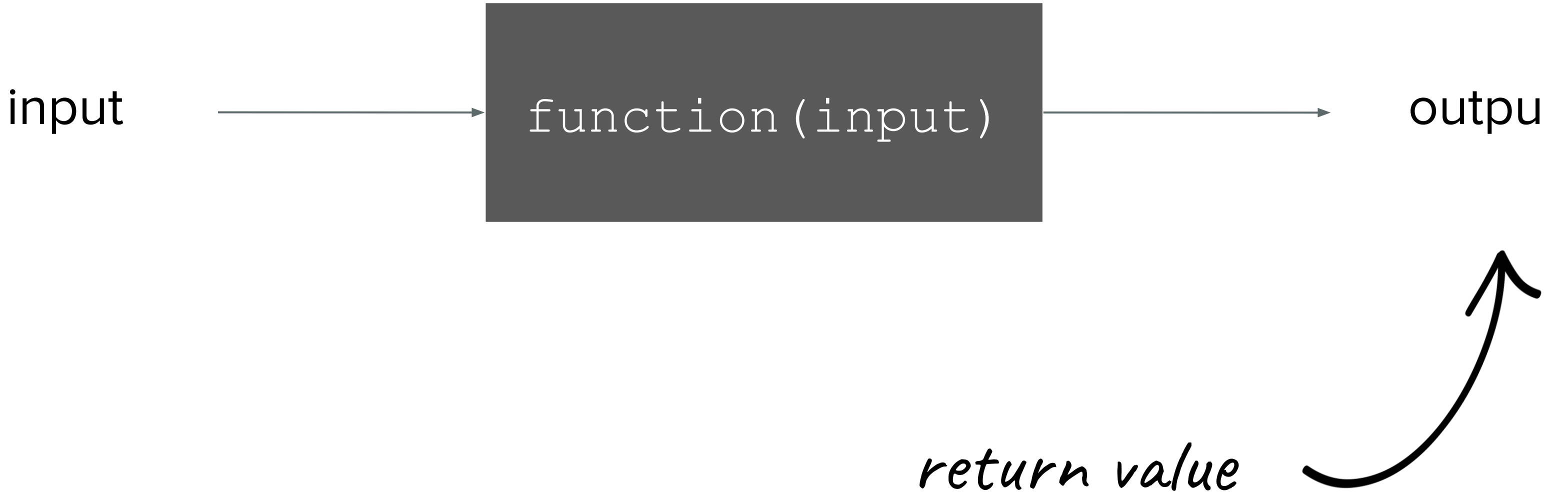


parameters/
arguments

Definition

argument(s)
The values passed into your function and assigned to its parameter variables

Anatomy of a function



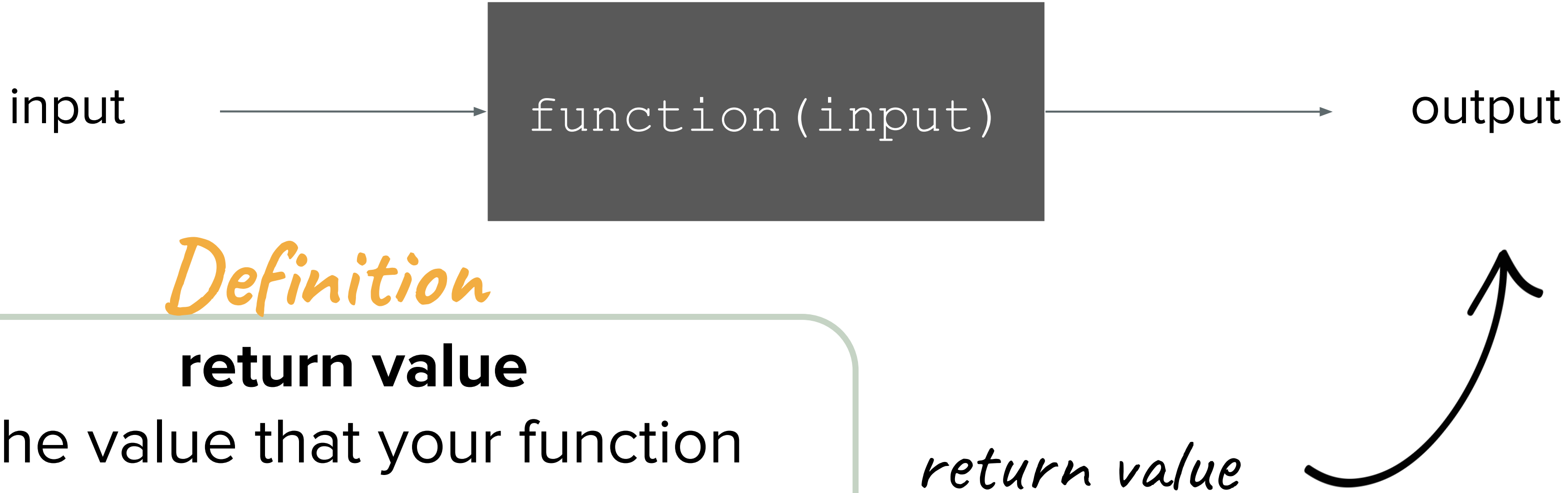
input

`function(input)`

output

return value

Anatomy of a function



input

`function(input)`

output

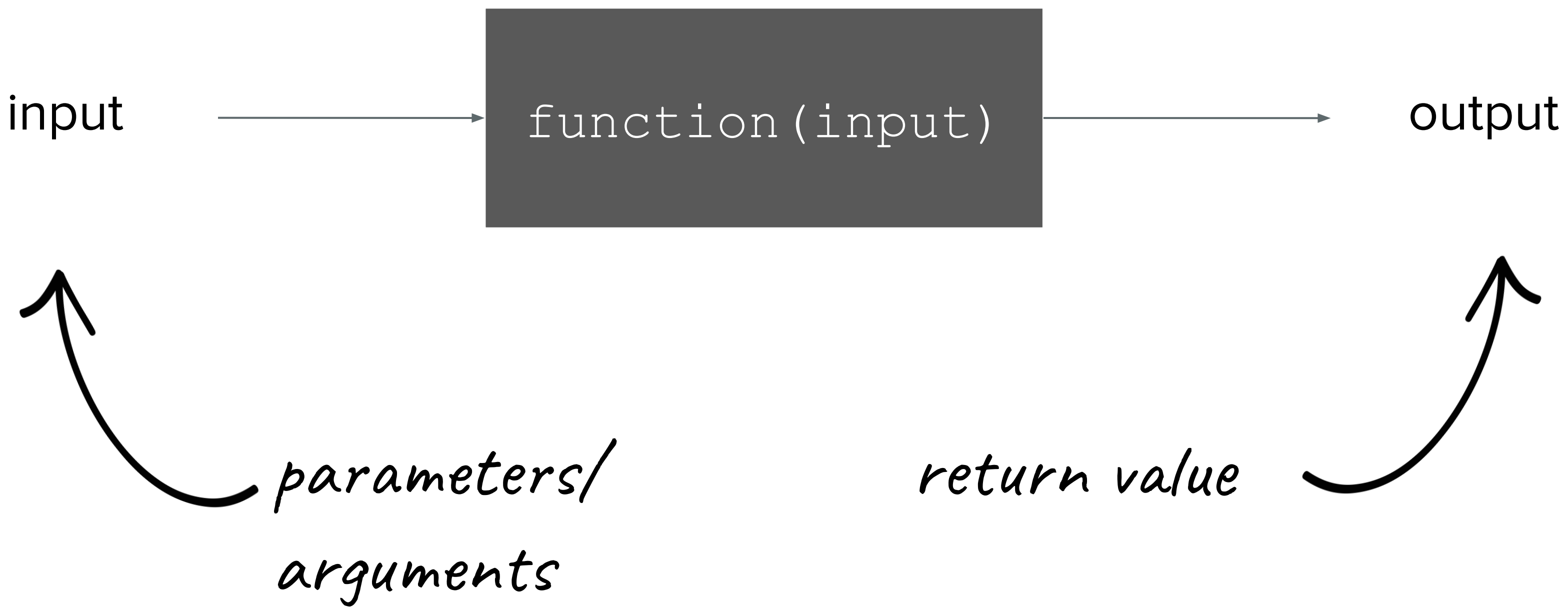
return value

Definition

return value

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

Anatomy of a function



input

`function(input)`

output

parameters/ arguments

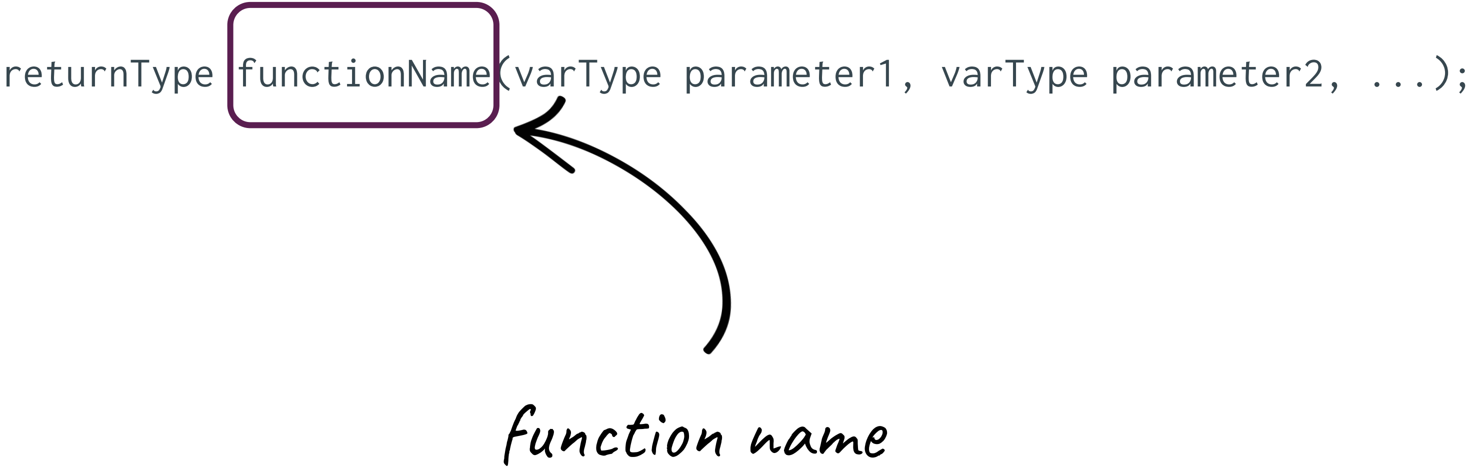
return value

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)

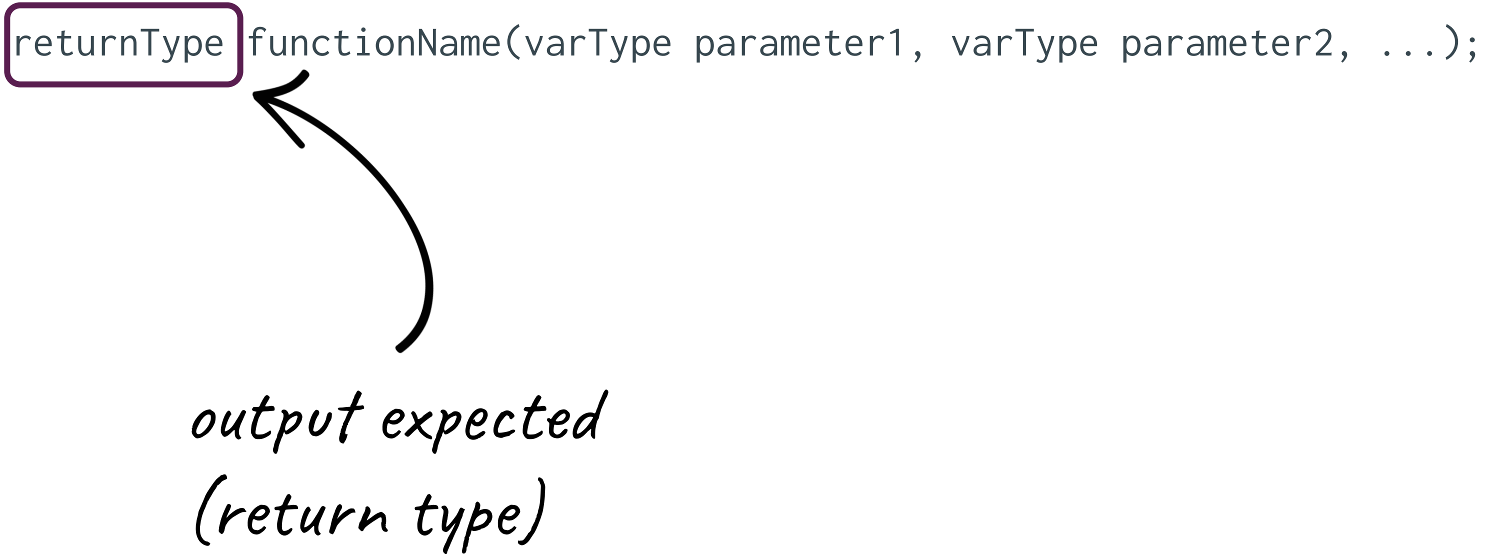
Anatomy of a function

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

input expected (parameters)

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.

Anatomy of a function

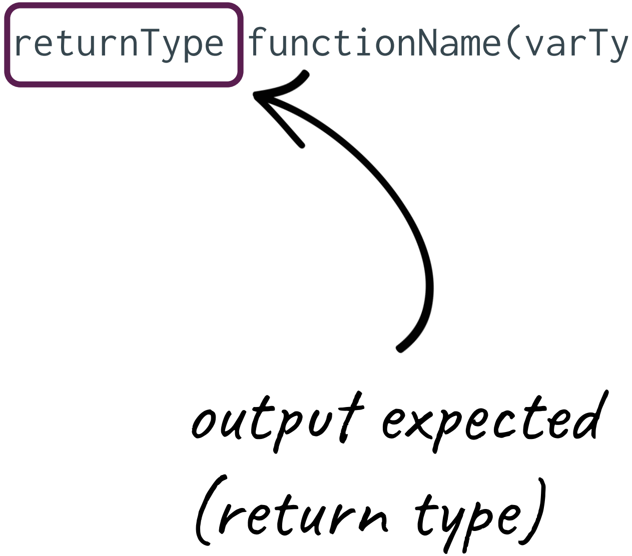


```
returnType functionName(varType parameter1, varType parameter2, ...);
```

output expected
(return type)

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;
}
```

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.

Function Example

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

callee

(function that got called)

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

caller

(function that made the call)

Function Example

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

parameters

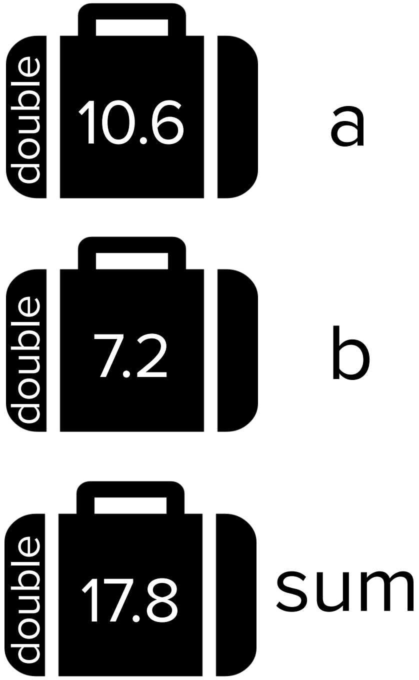
return value

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

arguments

Function Example

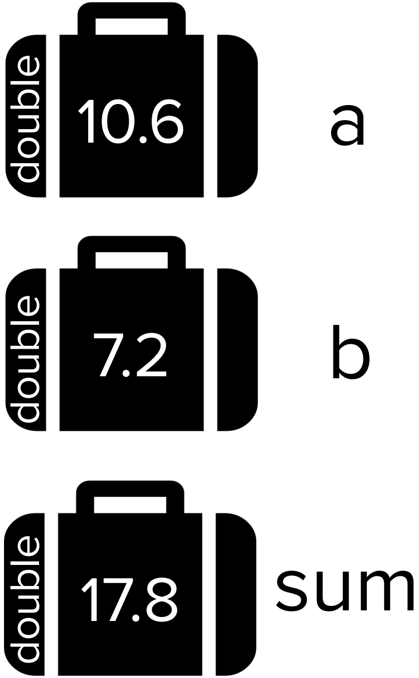
```
double average(double a, double b){
   double sum = a + b;
   return sum / 2;
}


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



Function Example

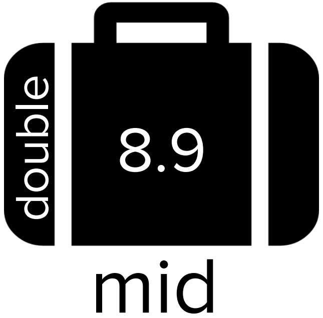
These variables only exist inside `average()`!

```
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;
}
```



Function Example

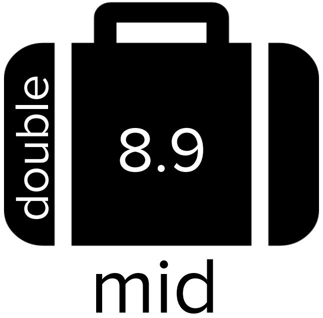
```
double average(double a, double b){
   double sum = a + b;
   return sum / 2;
}


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



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

```cpp
// 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;
    return 0;
}
```

Take a guess!

What is the console output of this block of code?

Pass by Value

```cpp
// 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;
    return 0;
}
```

```
myValue: 5 result: 10
```

Why is this the case?

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;
    return 0;
}
```

- 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 in the next lecture!

Control Flow

- **conditional statements:** if/else
- **loops:** while loops, for loops

are tools that help us control the flow

Boolean Expressions

Expression	Meaning
`a < b`	**a** is less than **b**
`a <= b`	**a** is less than or equal to **b**
`a > b`	**a** is greater than **b**
`a >= b`	**a** is greater than or equal to **b**
`a == b`	**a** is equal to **b**
`a != b`	**a** is not equal to **b**

Operator	Meaning
`a && b`	Both **a** AND **b** are `true`
`a || b`	Either **a** OR **b** are `true`
`!a`	If **a** is `true`, returns `false`, 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.

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

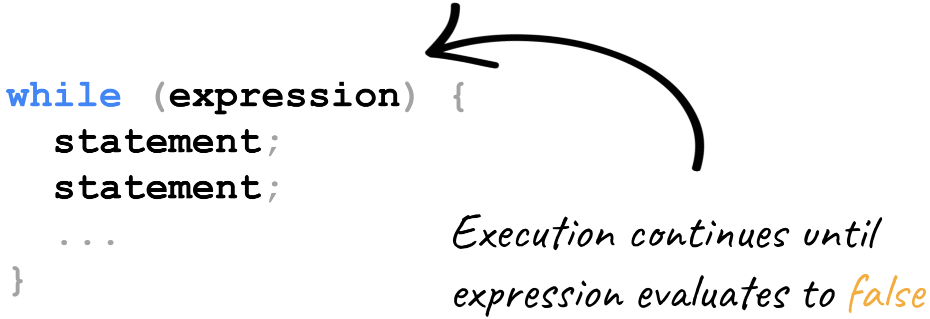
- Loops allow you to repeat the execution of a certain block of code multiple times

`while` loops

- 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` 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
```

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` 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;
  ...
}
```

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;
}
```

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;
   ...
}
```

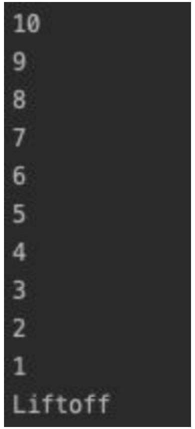
```
for (int i = 0; i < 3; i++) {
   cout << i << endl;
}
```

```
Output:
0
1
2
```

Exercise

Try it for yourself!

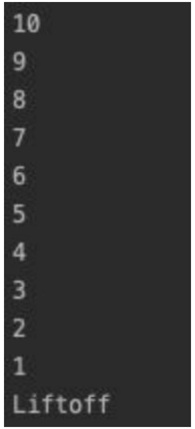
Write a program that prints out the calls for a spaceship that is about to launch. Countdown the numbers from 10 to 1 and then write “Liftoff.”



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

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.”



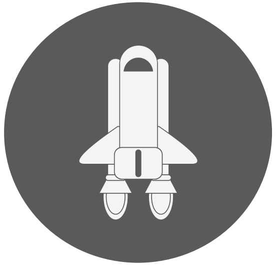
```
10
9
8
7
6
5
4
3
2
1
Liftoff
```

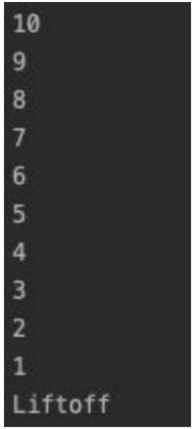
```python
def main():
    for i in range(10, 0, -1):
        print(i)
    print ("Liftoff")

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

Python

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
```

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

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

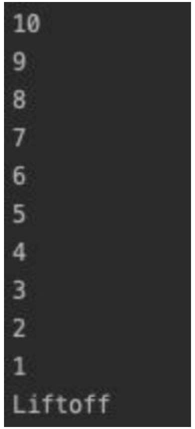
Python

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

int main() {
    /* TODO: Your code goes here! */

    return 0;

}
```

C++

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
```

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

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

Python

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

int main() {
    /* TODO: Your code goes here! */

    return 0;

}
```

C++

What’s next?

Strings, Testing, C++ Review