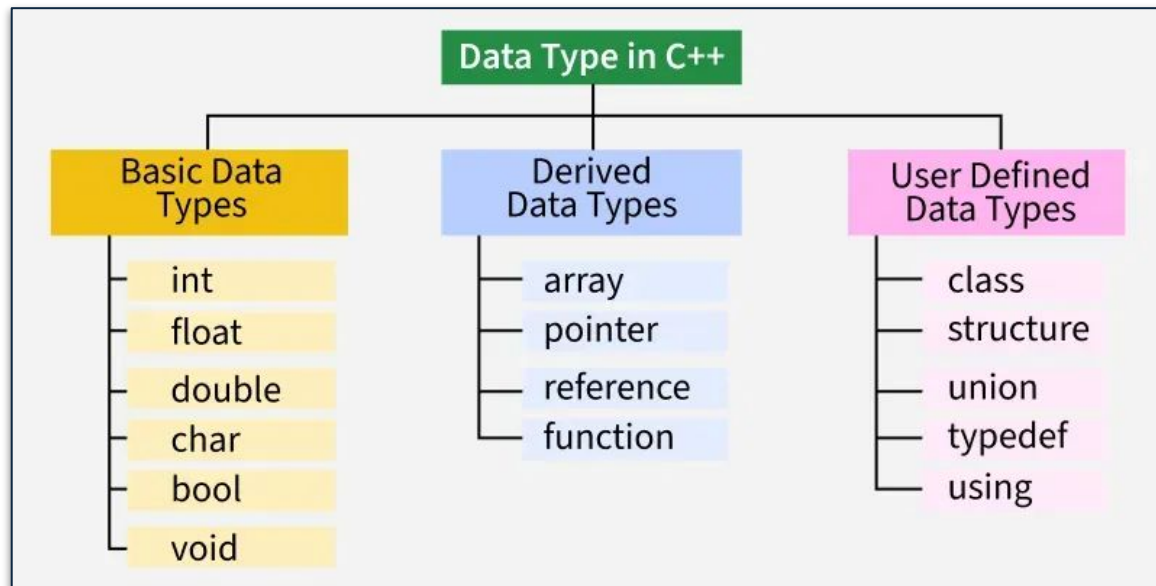


Lecture 2: Types and Structs

Stanford CS106L, Fall 2026

Preston Seay & Emily Ai



```
int main() {  
    struct {  
        string brand;  
        string model;  
        int year;  
    } myCar1, myCar2;  
  
    myCar1.brand = "BMW";  
    myCar1.model = "X5";  
    myCar1.year = 1999;  
}
```

Last Lecture

- Introductions!
- Why you should take 106L?
- Evolution of C++
- Course Logistics

Lecture 1: Welcome to CS106L!

CS106L, Fall 2026
Preston Seay & Emily Ai

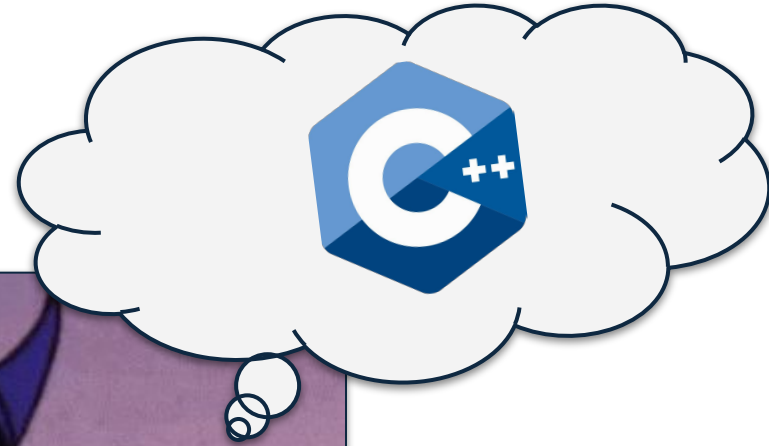
Slides are available at...

cs106l.stanford.edu

What's one thing you remember from last lecture?

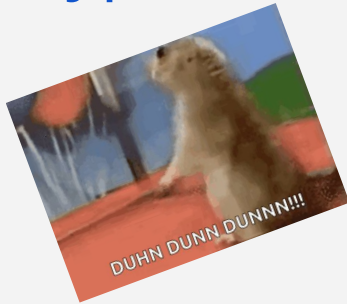
Pair up and discuss!

also discuss
what is your
spirit animal??



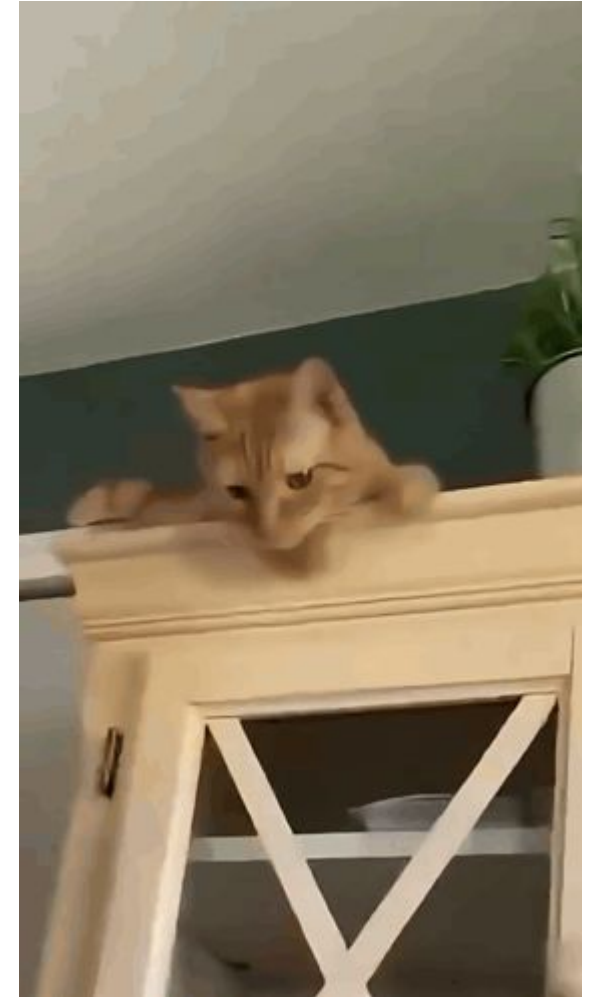
Today's Agenda

- Compile Time vs Run Time
- Statically Typed Languages
- Structs
- The **STD**
- Code demo
- Improving our code with **auto** and **using**





**Attendance Code:
cats**



Let's jump into!

Compiler VS Interpreter

Interpreted Languages

Source Code

```
print("Hello World")  
print("Welcome to ")  
for ch in "CS106L":  
    print(ch)
```

Interpreter

Machine Code

10110101

Output



A terminal window with a light gray sidebar containing icons for copy, download, home, and search. The main area is white and displays the text "Hello World".

```
Hello World
```

Interpreted Languages

Source Code

```
print("Hello World")  
print("Welcome to ")  
for ch in "CS106L":  
    print(ch)
```

Interpreter

Machine Code

```
10110101  
01011010
```

Output






Hello World
Welcome to

Interpreted Languages

Source Code

```
print("Hello World")  
print("Welcome to ")  
for ch in "CS106L":  
    print(ch)
```

Interpreter

Machine Code

```
10110101  
01011010  
10011101
```

Output



A terminal window with a light gray sidebar containing icons for copy, download, home, and a prompt. The main area shows the output of the program.

```
Hello World  
Welcome to
```

Interpreted Languages

Source Code

```
print("Hello World")  
print("Welcome to ")  
for ch in "CS106L":  
    print(ch)
```

Interpreter

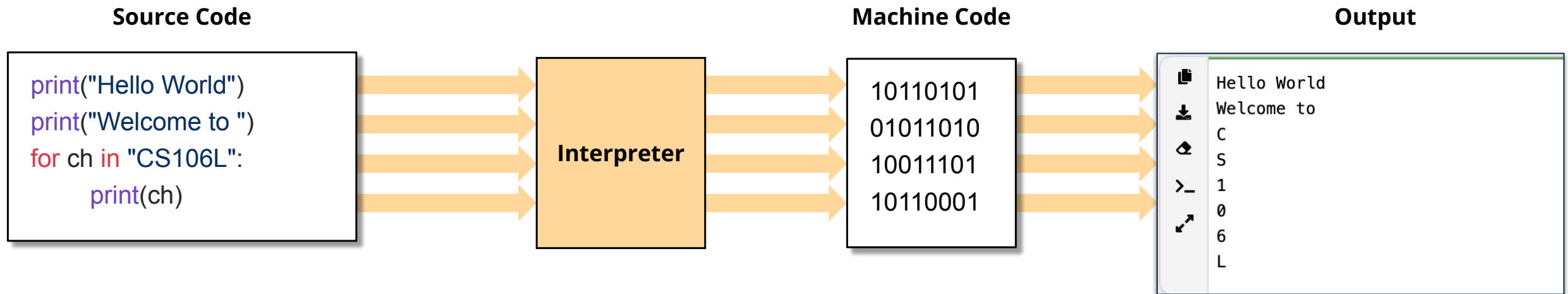
Machine Code

```
10110101  
01011010  
10011101  
10110001
```

Output

```
Hello World  
Welcome to  
C  
S  
1  
0  
6  
L
```

Interpreted Languages



THE BIG IDEA

The interpreted languages read each line of code
line-by-line, **translate** each line, and then **execute** it

Compiled Languages

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

Machine Code

```
10110101  
01011010  
10011101  
10110001
```

Compiled Languages

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

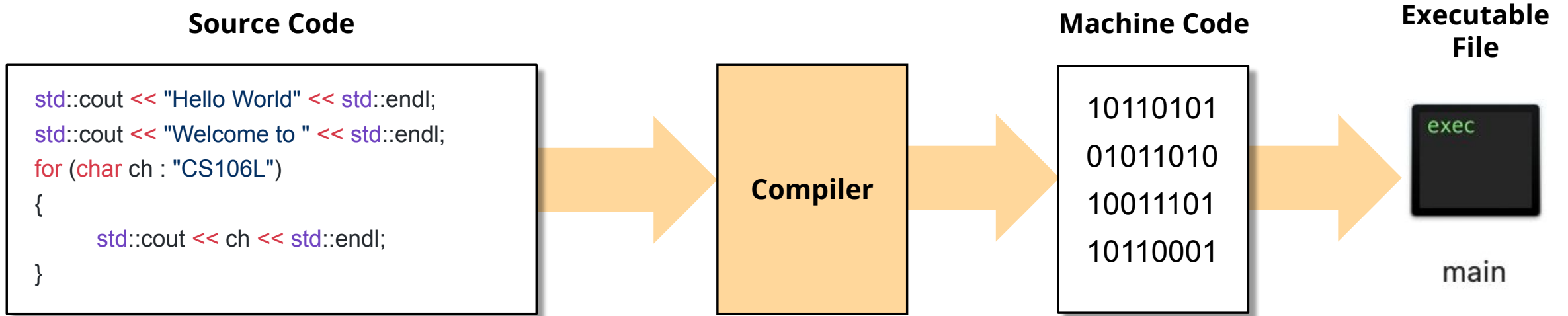
Machine Code

```
10110101  
01011010  
10011101  
10110001
```

Executable File

exec

main



Compiled Languages

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

Machine Code

```
10110101  
01011010  
10011101  
10110001
```

Executable File

exec

main

Executable File

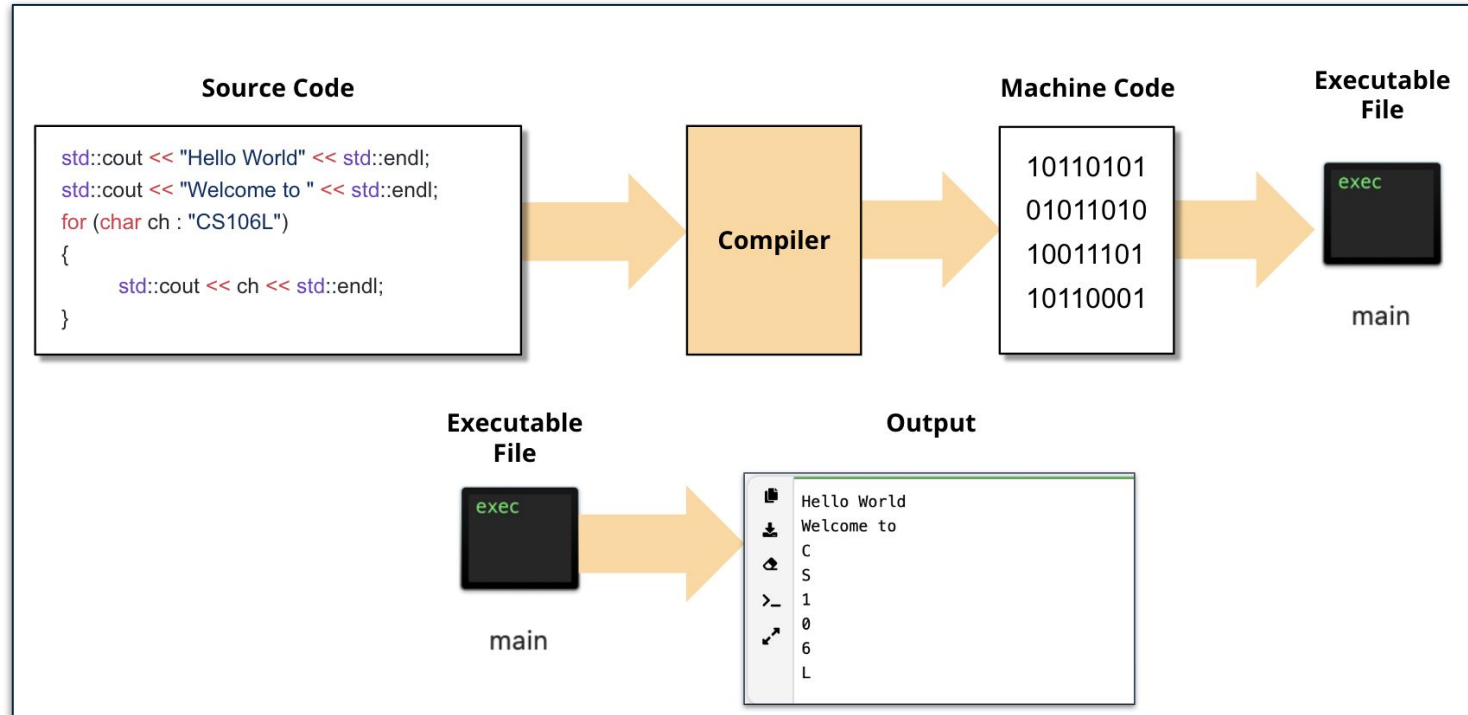
exec

main

Output

```
Hello World  
Welcome to  
C  
S  
1  
0  
6  
L
```

Compiled Languages



🧠 THE BIG IDEA 🧠

The compiler translates the **ENTIRE** program, packages it into an **executable file**, and then **executes** it

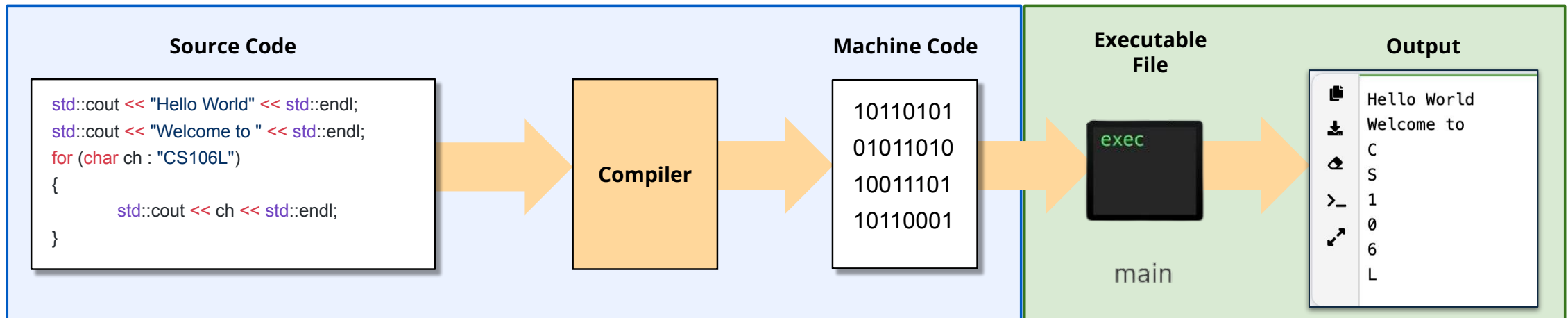
Compiled Languages: Compile Time V.S. Run Time



Compile Time

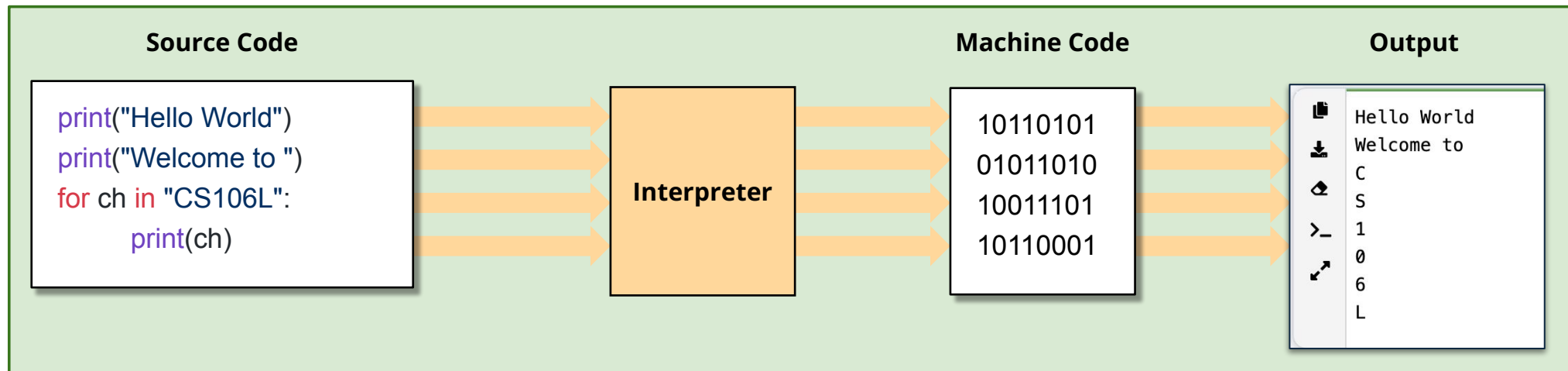


Run Time



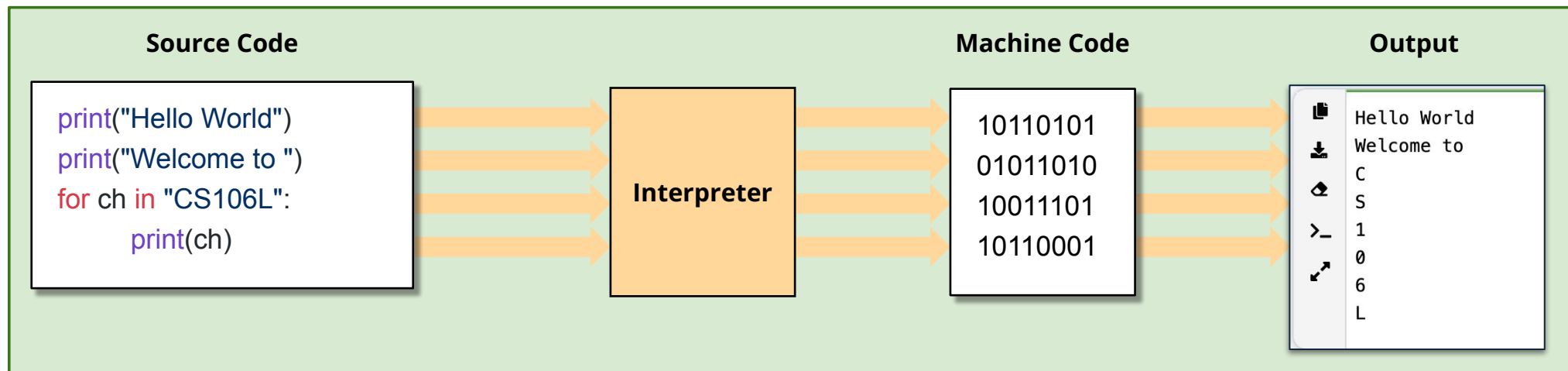
Interpreted Languages: Compile Time V.S. Run Time

 Run Time 



Interpreted Languages: Compile Time V.S. Run Time

 Run Time 



THE BIG IDEA

Interpreted languages all run in **run time**!

Compiled Languages run in first **compile time** then **run time**

What questions do you have?



bjarne_about_to_raise_hand

C++ is a compiled language

Q1: So we know the process of how C++ runs our code...

But when do we deal with errors?

Python

```
print("Running...")  
hello = "Hello ";  
world = "World!";  
print(hello * world)
```

```
$ python3 program.py
```

Running...

TypeError: can't multiply sequence by
non-int of type 'str'

C++

```
int main() {  
    std::cout << "Running..." << std::endl;  
    std::string hello = "Hello ";  
    std::string world = "World!";  
    std::cout << hello * world << std::endl;  
    return 0;  
}
```

```
$ g++ main.cpp
```

error: no match for 'operator*' (operand types are
'std::string' and 'std::string')

Python

```
print("Running...")  
hello = "Hello"  
world = "World!"  
print(hello * world)
```



10110101
01011010
10011101
10110001



Running...

Python

```
print("Running...")  
hello = "Hello"  
world = "World!"  
print(hello * world)
```

10110101
01011010
10011101
10110001

Running...

Python

```
print("Running...")  
hello = "Hello"  
world = "World!"  
print(hello * world)
```

10110101
01011010
10011101
10110001

Running...

Python

```
print("Running...")  
hello = "Hello"  
world = "World!"  
print(hello * world)
```

10110101
01011010
10011101
10110001

Running...

TypeError:
can't
multiply
sequence
by non-int
of type
'str'

Python

```
print("Running...")  
hello = "Hello ";  
world = "World!";  
print(hello * world)
```

Run Time Error

```
$ python3 program.py
```

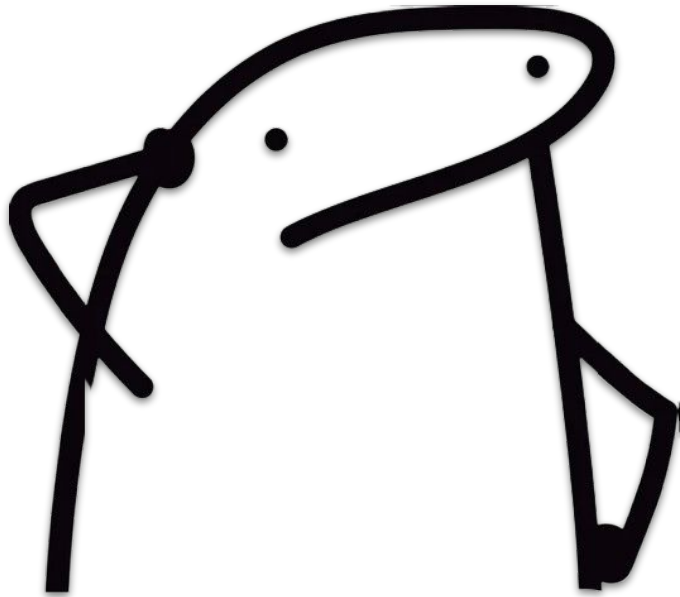
Running...

TypeError: can't multiply sequence by
non-int of type 'str'



C++

Any guesses for when this error occurs? Run time or compile time?



```
int main() {  
    std::cout << "Running..." << std::endl;  
    std::string hello = "Hello ";  
    std::string world = "World!";  
    std::cout << hello * world << std::endl;  
    return 0;  
}
```

```
$ g++ main.cpp
```

THE BIG IDEA

This error occurs during
compile time!

When we are translating, we see
that we try to multiply two
strings and the compiler goes
“hey that’s not allowed!!”

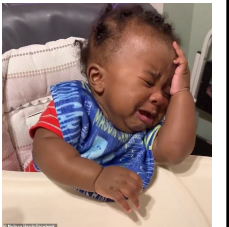
C++

```
int main() {  
    std::cout << "Running..." << std::endl;  
    std::string hello = "Hello ";  
    std::string world = "World!";  
    std::cout << hello * world << std::endl;  
    return 0;  
}
```

Compile Time Error

\$ g++ main.cpp

error: no match for 'operator*' (operand types are
'std::string' and 'std::string')



Python

```
print("Running...")  
hello = "Hello ";  
world = "World!";  
print(hello * world)
```

Run Time Error

```
$ python3 program.py
```

Running...

TypeError: can't multiply sequence by
non-int of type 'str'



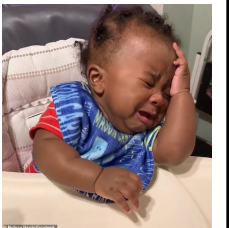
C++

```
int main() {  
    std::cout << "Running..." << std::endl;  
    std::string hello = "Hello ";  
    std::string world = "World!";  
    std::cout << hello * world << std::endl;  
    return 0;  
}
```

Compile Time Error

```
$ g++ main.cpp
```

error: no match for 'operator*' (operand types are
'std::string' and 'std::string')



What questions do you have?



bjarne_about_to_raise_hand

Types are super important!!

Types

- A **type** refers to the “category” of a variable
- C++ comes with built-in types
 - **int** 106
 - **double** 71.4
 - **string** “Welcome to CS106L!”
 - **bool** true / false
 - **size_t** 12 // Non-negative

Hopefully this
sounds familiar :D



Aside: `int` vs `size_t`

- `int` is a signed integer type
- `size_t` is an unsigned integer type
- Functions like `string.size()`, and `vector.size()` return a `size_t` rather than an `int`

```
int x = -5;      // valid
size_t y = -5;   // allowed, but becomes a huge
                 positive number
// in a 64 bit system, equivalent to:
18446744073709551611
```

```
// mixing them isn't good practice and can cause
errors
for (int i = 0; i < s.size(); i++) {
    ...
}
```

We know that the compiler checks for types before generating machine code.

This means...

C++ is a **statically typed language**

Dynamic Typing

Python (Dynamic Typing)

```
a = 3
```

```
b = "test"
```

```
def foo(c):
```

```
    d = 106
```

```
    d = "hello world!"
```

The interpreter assigns variables a **type** at runtime based on the variable's value at that time

Dynamic Typing

Python (Dynamic Typing)

```
a = 3
```

```
b = "test"
```

```
def foo(c):
```

```
    d = 106
```

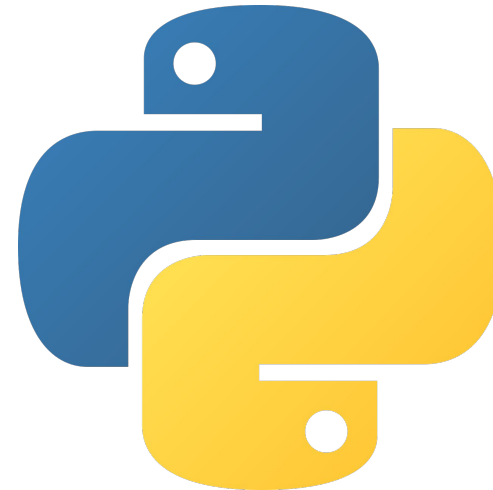
```
    d = "hello world!"
```

The interpreter assigns variables a **type** at runtime based on the variable's value at that time

Oh you are just switching things up with me! That's okay! I'll catch on!

So d was originally an **integer**, and now it's a **string** :D

No problemo!



Dynamic Typing

Python (Dynamic Typing)

```
a = 3
b = "test"

def foo(c):
    d = 106
    d = "hello world!"
```



The interpreter assigns variables a **type at runtime based on the variable's value at that time**

Static Typing

C++ (Static Typing)

```
int a = 3;
string b = "test";

void foo(string c)
{
    int d = 106;
    d = "hello world!";
}
```



- Every variable must declare a type
- Once declared, the type **cannot change**

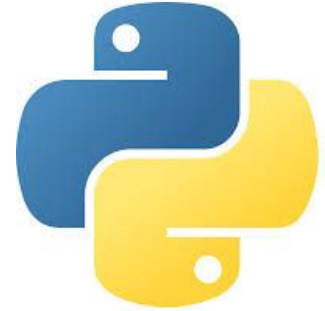
Why static typing?

- More efficient
- Easier to understand and reason about
- Better error checking

Better error checking

```
def add_3(x):  
    return x + 3
```

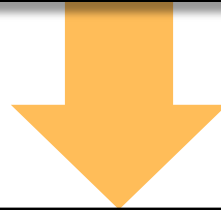
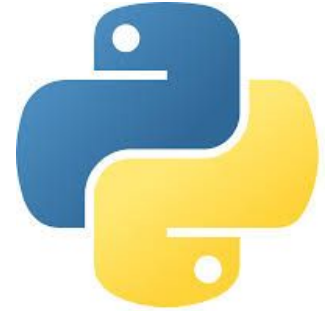
```
add_3("CS106L") # Oops, that's a string. Runtime error!
```



Better error checking

```
def add_3(x):  
    return x + 3
```

```
add_3("CS106L") # Oops, that's a string. Runtime error!
```



```
int add_3(int x) {  
    return x + 3;  
}
```

```
add_3("CS106L"); // Can't pass a string when int expected. Compile time error!
```



What questions do you have?



bjarne_about_to_raise_hand

Your turn 🖐️🖐️💪💪

- TODO: Fill in the blanks underneath with the correct type
- NOTE: `(int) x` **casts** `x` to an int by dropping decimals
 - E.g. `(int) 5.7 = 5`

_____ `a = "test";`

_____ `b = 3.2 * 5 - 1;`

_____ `c = 5 / 2;`

_____ `d(int foo) { return foo / 2; }`

_____ `e(double foo) { return foo / 2; }`

_____ `f(double foo) { return (int)(foo + 0.5); }`

_____ `g(double c) { std::cout << c << std::endl; }`

Your turn

```
std::string a = "test";
```

```
double b = 3.2 * 5 - 1;
```

```
int c = 5 / 2; // What does this equal?
```

```
int d(int foo) { return foo / 2; }
```

```
double e(double foo) { return foo / 2; }
```

```
int f(double foo) { return (int)(foo + 0.5); } // What's this?
```

```
void g(double c) { std::cout << c << std::endl; }
```

Aside: Function Overloading

Defining two functions with the same name but different parameters

```
double axotl(int x) {           // (1)
    return (double) x + 3;      // typecast: int → double
}
```

```
double axotl(double x) {       // (2)
    return x * 3;
}
```

```
axotl(2);           // uses version ____, returns ____
axotl(2.0);         // uses version ____, returns ____
```

Aside: Function Overloading

Defining two functions with the same name but different parameters

```
double axolotl(int x) {           // (1)
    return (double) x + 3;        // typecast: int → double
}
```

```
double axolotl(double x) {       // (2)
    return x * 3;
}
```

```
axolotl(2);           // uses version (1), returns 5.0
axolotl(2.0);         // uses version (2), returns 6.0
```

C++ is a compiled, statically typed language

What questions do you have?



bjarne_about_to_raise_hand

Structs

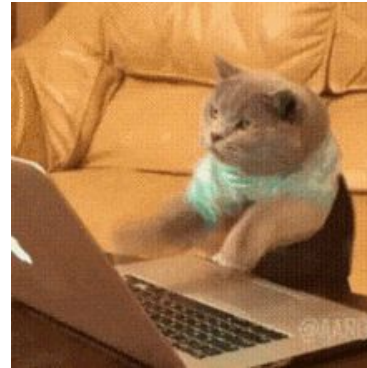
Keeping track of students

- Every student ID has a few properties
 - A name (**string**)
 - A SUNet (**string**)
 - An ID # (**int**)



Okay let's make generating IDs into a function!!

```
return type issueNewID() {  
    yada yada code yada yada  
  
    return our ID stuff (ID #, name, sunet)  
}
```



this looks like the most legit function I've ever seen 😎😎 (jk..)

A fundamental problem

```
return type issueNewID() {  
    // How can we return all three things?  
    // What should our return type be? 😞😞  
  
    // In Python this would look like...  
    // return "Stanford Tree", "theTREE", 0000002  
}
```

How do we return more than one value? :OO

Introducing... structs!



Structs bundle data together

```
struct StanfordID {  
    string name;           // These are called fields  
    string sunet;          // Each has a name and type  
    int idNumber;  
};  
  
StanfordID id;             // Initialize struct  
id.name = "THE Stanford Tree"; // Access field with '.'  
id.sunet = "theTREE";  
id.idNumber = 0000002;
```

Returning multiple values

```
StanfordID issueNewID() {  
    StanfordID id;  
  
    id.name = "THE Stanford Tree";  
    id.sunet = "theTREE";  
    id.idNumber = 00000002;  
  
    return id;  
}
```



Default Initialization

```
StanfordID id;  
id.name = "THE Stanford Tree";  
id.sunet = "theTREE";  
id.idNumber = 0000002;
```

Uniform Initialization

```
StanfordID id;  
id.name = "THE Stanford Tree";  
id.sunet = "theTREE";  
id.idNumber = 0000002;
```

**We'll learn more
about this next time!**



// Order depends on field order in struct. '=' is optional

```
StanfordID tree = { "THE Stanford Tree", "theTREE", 0000002 };  
StanfordID lelandjr { "Leland Stanford Jr", "thejunior", 5430282 };
```



Using uniform initialization

```
StanfordID issueNewID() {  
    StanfordID id;  
    id.name = "THE Stanford Tree";  
    id.sunet = "theTREE";  
    id.idNumber = 0000002;  
    return id;  
}
```



```
StanfordID issueNewID() {  
    StanfordID id = { "THE Stanford Tree", "theTREE", 0000002 };  
    return id;  
}
```

THE BIG IDEA

A **struct** bundles **named variables** into a new type

What questions do you have?



bjarne_about_to_raise_hand

What is an std !!?  

std — The C++ Standard Library

- Built-in types, functions, and more provided by C++
- We prefix standard library names with `std::`
- You need to `#include` the relevant file
 - `#include <string>` → `std::string`
 - `#include <utility>` → `std::pair`
 - `#include <iostream>` →
`std::cout, std::endl`

```
1
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      std::string name = "Emily";
8      std::cout << name << std::endl;
9
10     return 0;
11 }
```

std — The C++ Standard Library

- `using namespace std;`
- Considered bad style as it can introduce ambiguity
 - (What would happen if we defined our own `sort`?)

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

// You write your own sort function
void sort(int a, int b) {
    cout << "emily's sort >:D!";
}

int main() {
    sort(3, 5); // which sort? emily's or std::sort?
}
```

std — The C++ Standard Library

- See the official standard at cppreference.com!
- Avoid cplusplus.com...
 - It is outdated and filled with ads 😭

To use **std::cout**, you must **#include** it

std::cout is defined in a header file called iostream

```
#include <iostream>
```

```
// Now we can use `std::cout` in our code.
```

```
std::cout << "Hello World!";
```

What does **#include** do?

```
#include <iostream>
```

```
std::cout << "Hello World!";
```

iostream

```
namespace std {  
  
    class ostream {  
    public:  
        ostream& operator<<(const char* text);  
        ostream& operator<<(int value);  
    };  
  
    extern ostream cout;  
  
}
```

What does **#include** do?

```
namespace std {  
  
class ostream {  
public:  
    ostream& operator<<(const char* text);  
    ostream& operator<<(int value);  
};  
  
extern ostream cout;  
  
}  
  
std::cout << "Hello World!";
```

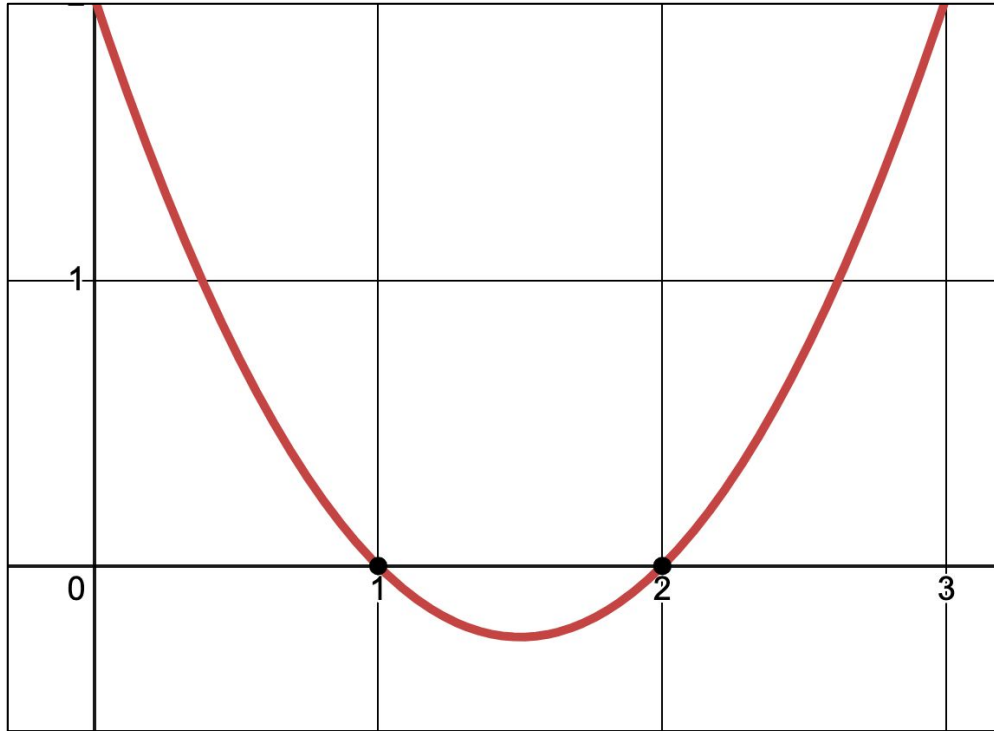
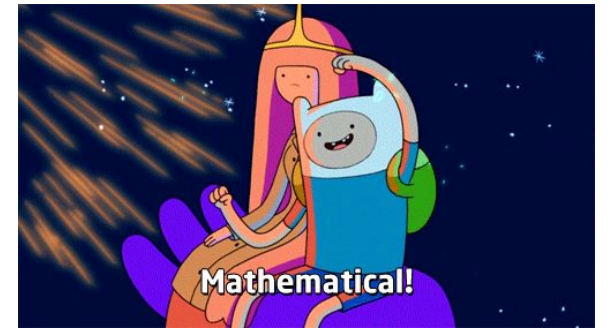
What questions do you have?



bjarne_about_to_raise_hand

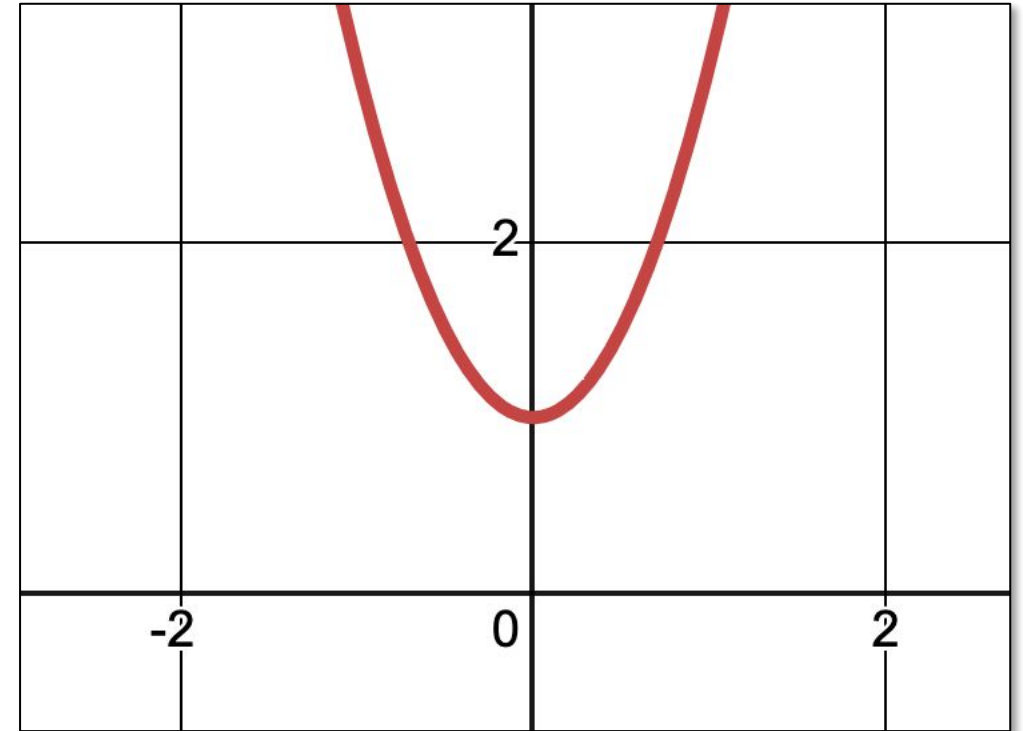
Code Demo

Solving a Quadratic Equation



$$x^2 - 3x + 2 = 0$$

$$x = 1, x = 2$$



$$2x^2 + 1 = 0$$

no solution

Solving a Quadratic Equation

- If we have $ax^2 + bx + c = 0$
- Solutions are $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- If $b^2 - 4ac$ is negative, there are no solutions

What are the solutions
(if any)?

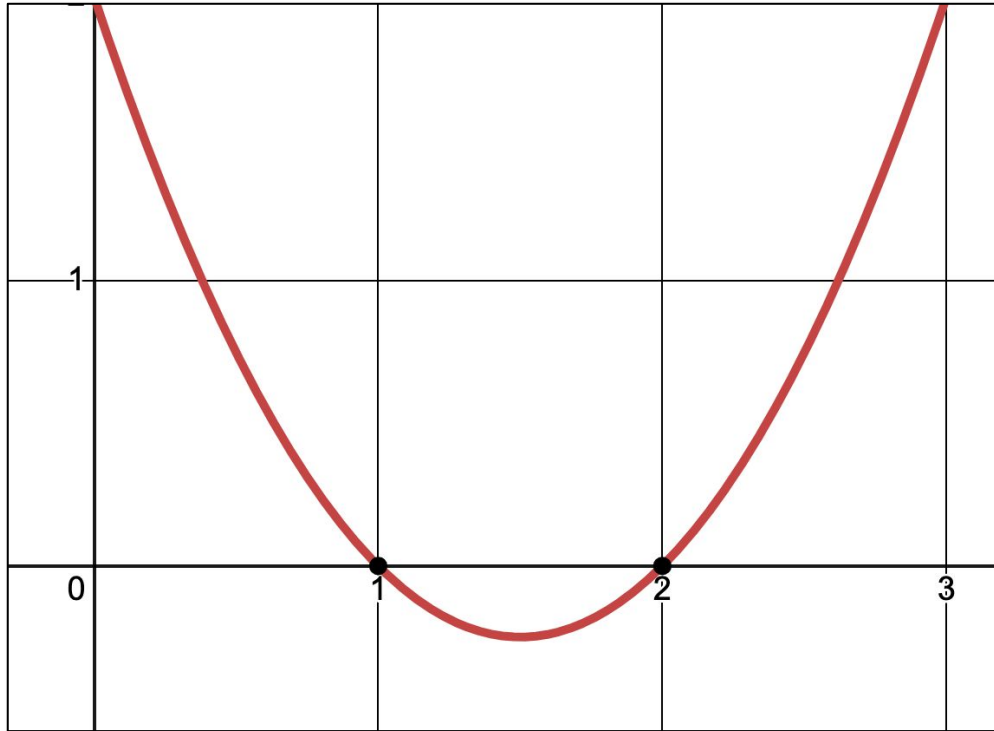
Return Value

```
std::pair<bool, std::pair<double, double>> solveQuadratic(double a, double b, double c);
```

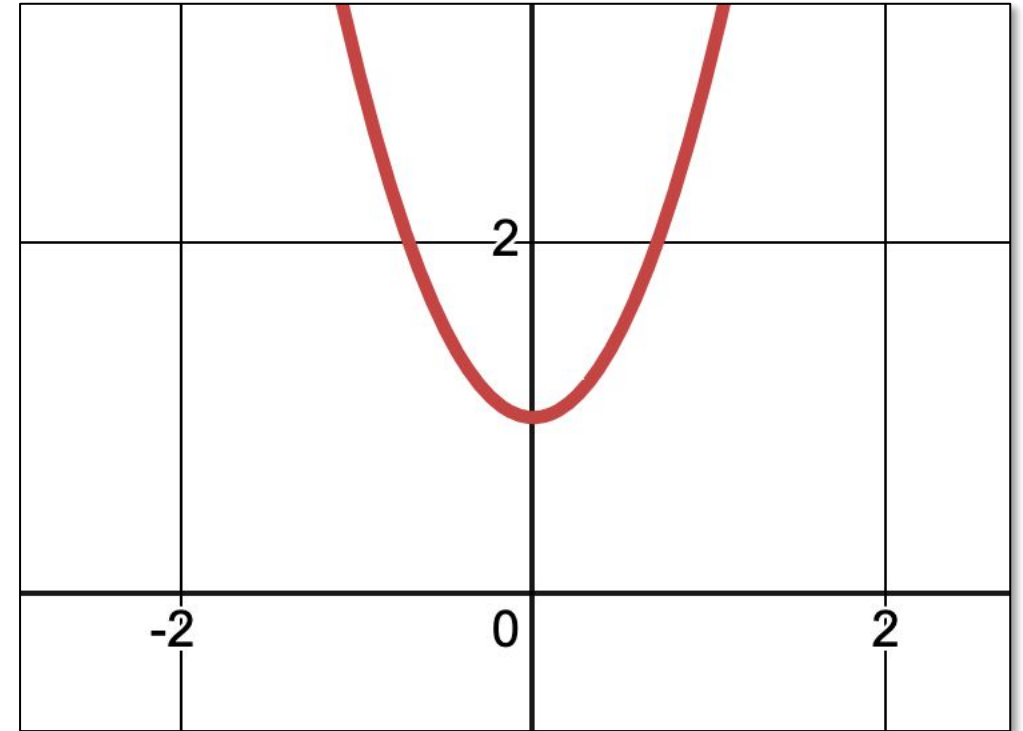
Is there a
solution?

Coefficients

`std::pair<bool, std::pair<double, double>>`



`{ true, { 1.0, 2.0 } }`



`{ false, doesnt_matter }`

e.g. `{ false, { 0.0, 0.0 } }`

Solving a Quadratic Equation

- If we have $ax^2 + bx + c = 0$
- Solutions are $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- If $b^2 - 4ac$ is negative, there are no solutions
- **Your task:** Write a function to solve a quadratic equation:

```
std::pair<bool, std::pair<double, double>> solveQuadratic(double a, double b, double c);
```



The `sqrt` function from the `<cmath>` header can calculate the square root

Let's code this together 

Improving Our Code

The `using` keyword

The **using** keyword

- Typing out long type names gets tiring
- We can create **type aliases** with the **using** keyword

```
std::pair<bool, std::pair<double, double>> solveQuadratic(double a, double b, double c);
```



```
using Zeros = std::pair<double, double>;  
using Solution = std::pair<bool, Zeros>;  
Solution solveQuadratic(double a, double b, double c);
```



using is kind of like a variable for types!

The **auto** keyword

The `auto` keyword

- The `auto` keyword tells the compiler to infer the type

```
std::pair<bool, std::pair<double, double>> result = solveQuadratic(a, b, c);
```



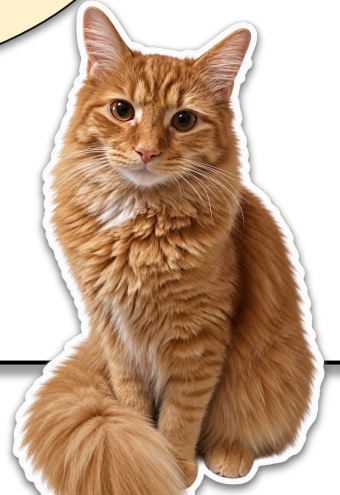
```
auto result = solveQuadratic(a, b, c);
```

// This is exactly the same as the above!

// **result** still has type `std::pair<bool, std::pair<double, double>>`

// We just told the compiler to figure this out for us!

The compiler checks for the declared return type of `solveQuadratic` and fills it in for `auto` :O



auto is still statically typed!

```
auto i = 1; // int inferred
```

```
i = "hello!"; // ❌ Doesn't compile
```

Which one is clearer?

```
std::pair<bool, std::pair<double, double>> result = ...;  
auto result = ...;
```

Which one is clearer?

```
auto i = 1;
```

```
int i = 1;
```

What questions do you have?



bjarne_about_to_raise_hand

Recap

Recap

- C++ is a compiled, statically typed language
- Structs bundle data together into a single object
- `#include` from the C++ Standard Library to use built-in types
 - And use the `std::` prefix too!
- Quality of life features to improve your code
 - `using` creates type aliases
 - `auto` infers the type of a variable

See you all on Tuesday!! :)

Have a great weekend :D