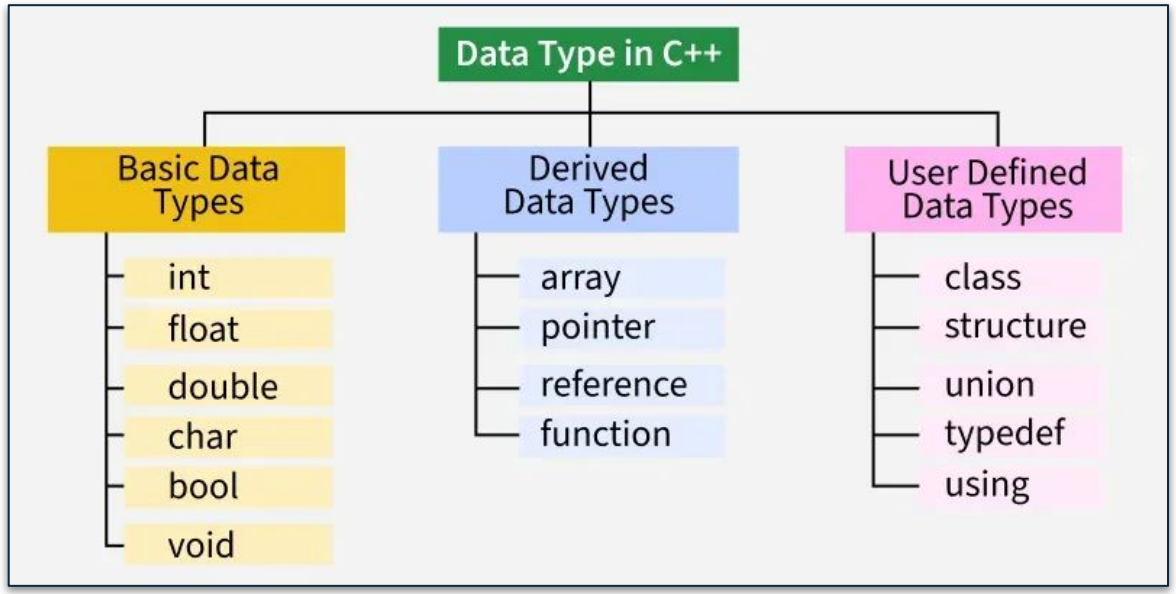
Lecture 2: Types and Structs

Stanford CS106L, Fall 2026

Rachel Fernandez, Thomas Poimenidis




```cpp
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?
- Course Logistics
- Evolution of C++

Lecture 1:
Welcome to CS106L!

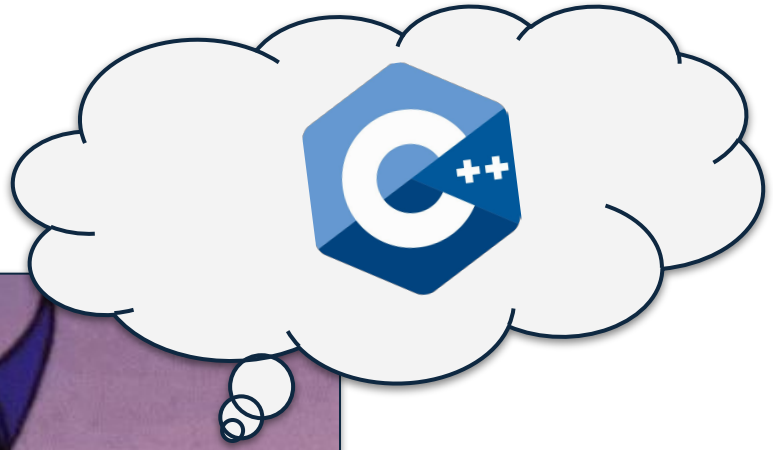
CS106L, Spring 2025

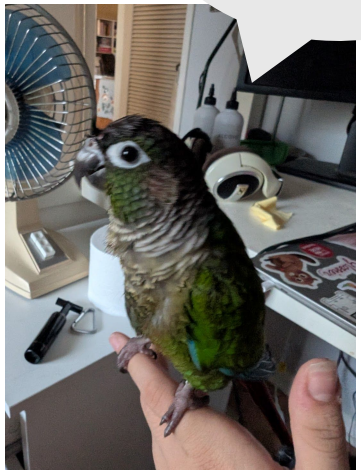
Slides are available at...

cs106l.stanford.edu

What's one thing you remember from last lecture?

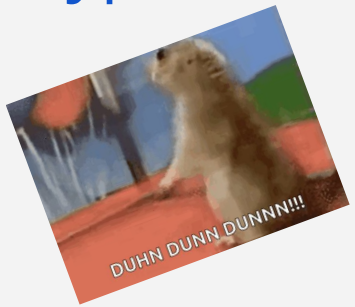
Pair up and discuss!





Today's Agenda

- Compile Time vs Run Time
- Statically Typed Languages
- Structs
- The **STD**

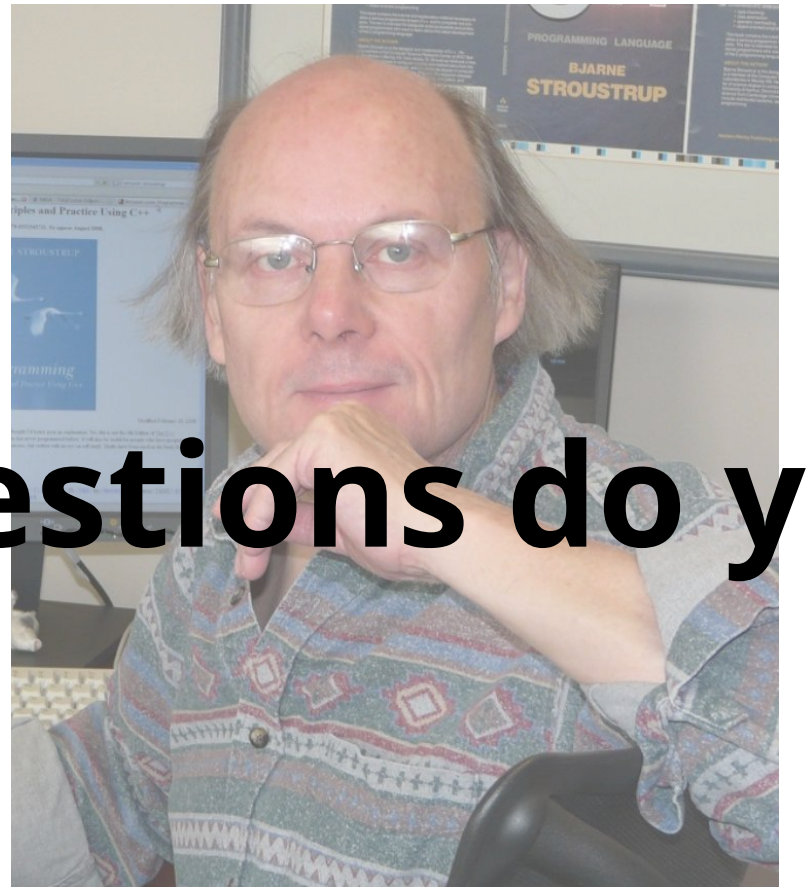


- Code demo
- Improving our code with `auto` and `using`

We'll cover a LOT of material in this class

Please ask questions!!!

What questions do you have?

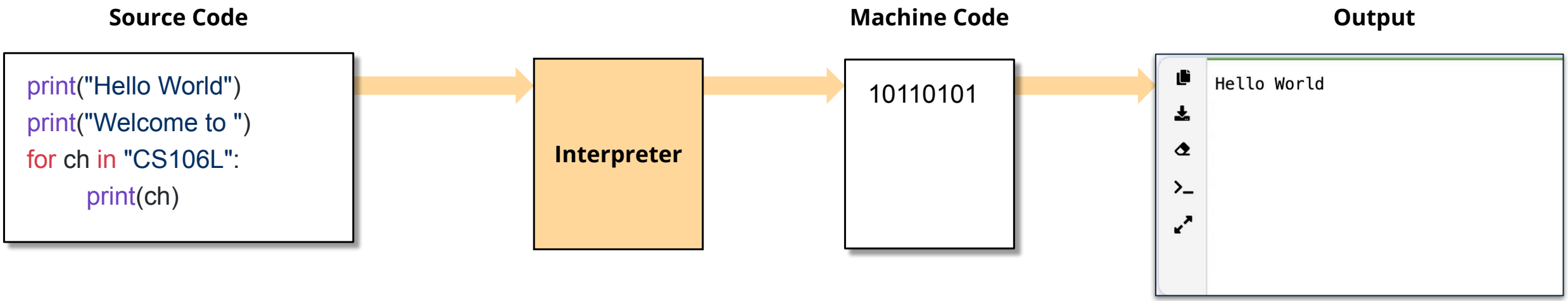


bjarne_about_to_raise_hand

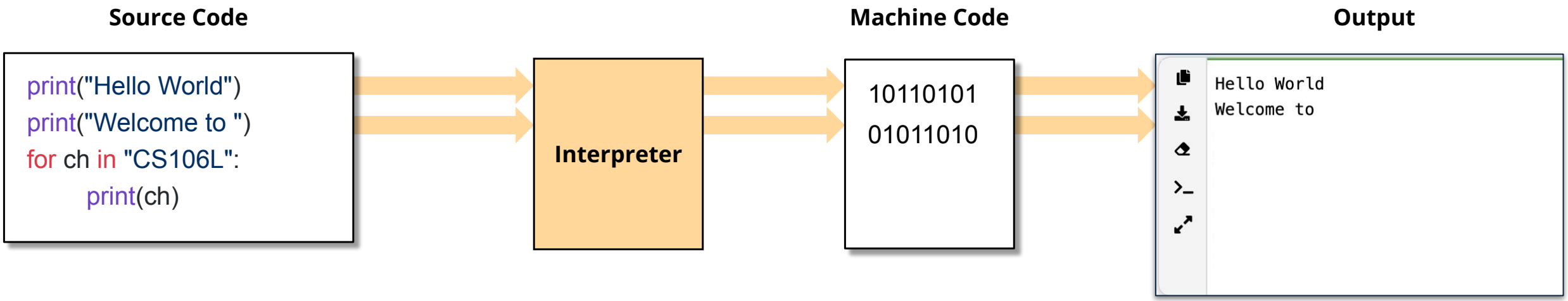
Let’s recap!

Compiler VS Interpreter

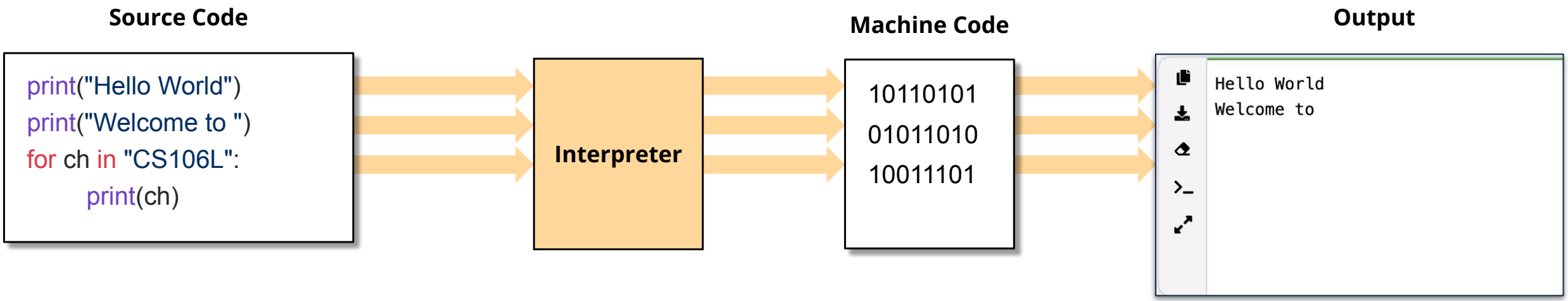
Interpreted Languages



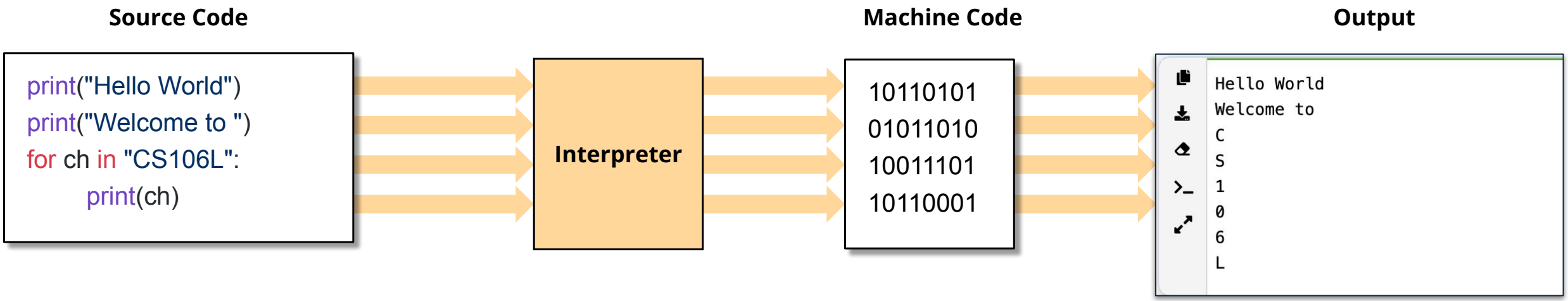
Interpreted Languages



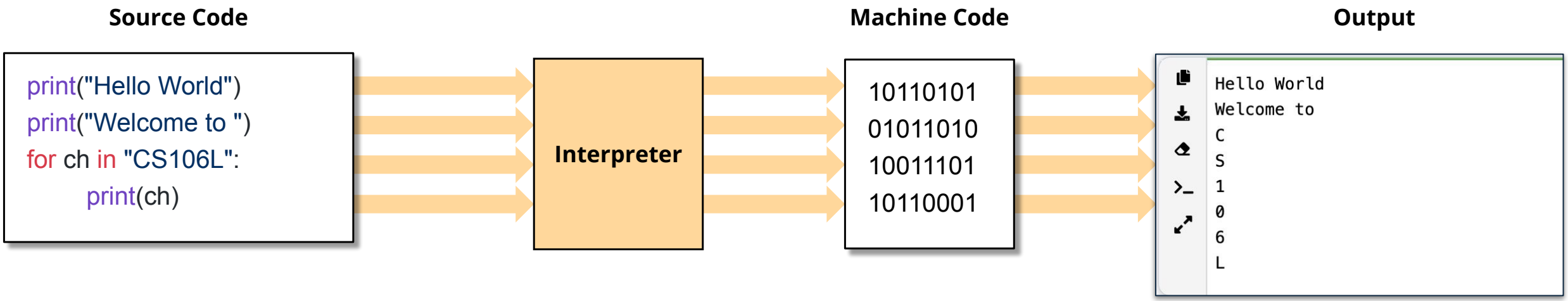
Interpreted Languages



Interpreted Languages



Interpreted Languages



🧠 THE BIG IDEA 🧠

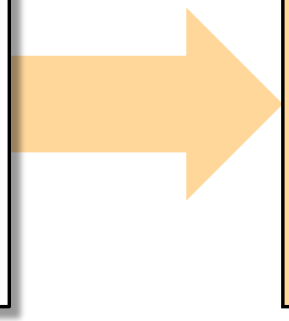
The interpreted languages read each line of code

line-by-line, **translate** each line, and then **execute** it

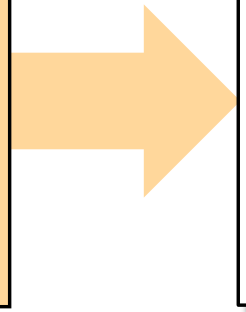
Compiled Languages

Source Code

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



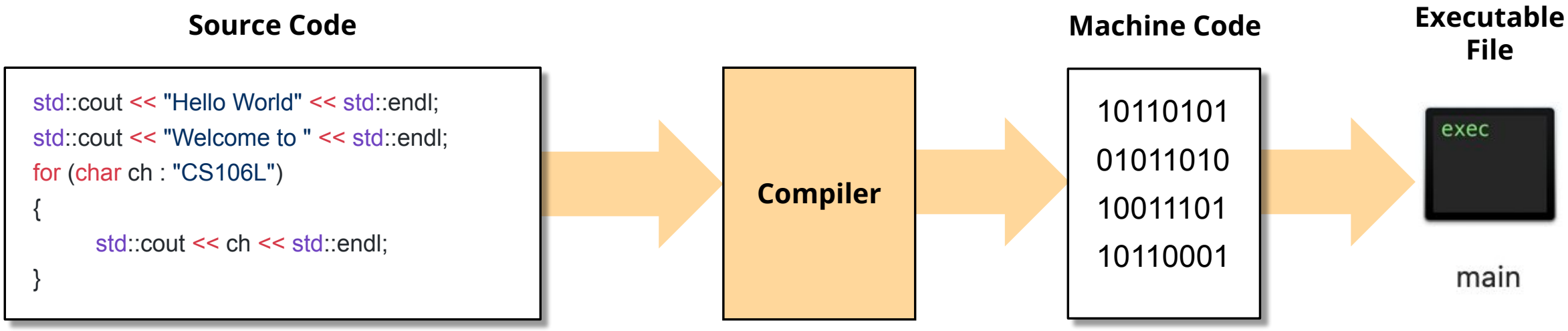
Compiler



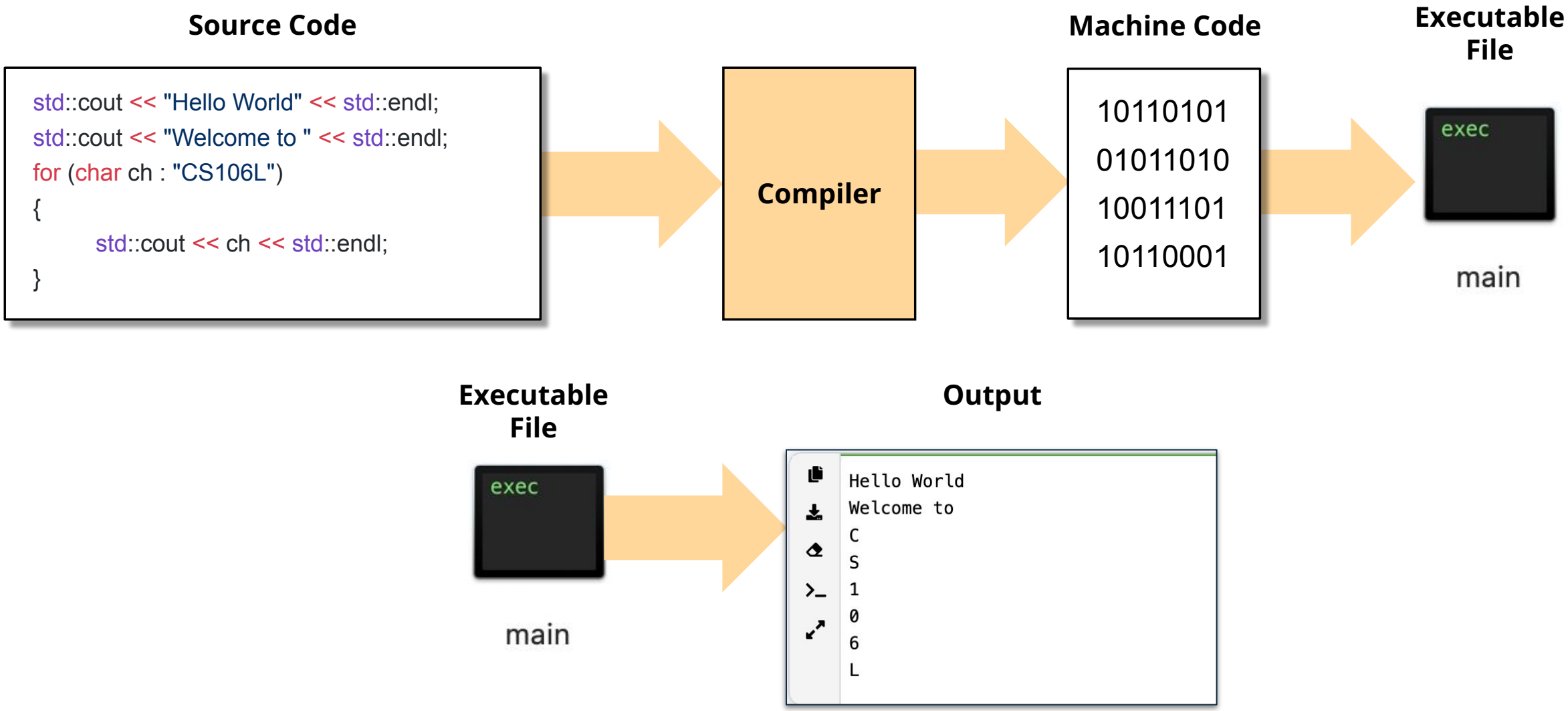
Machine Code

```
10110101
01011010
10011101
10110001
```

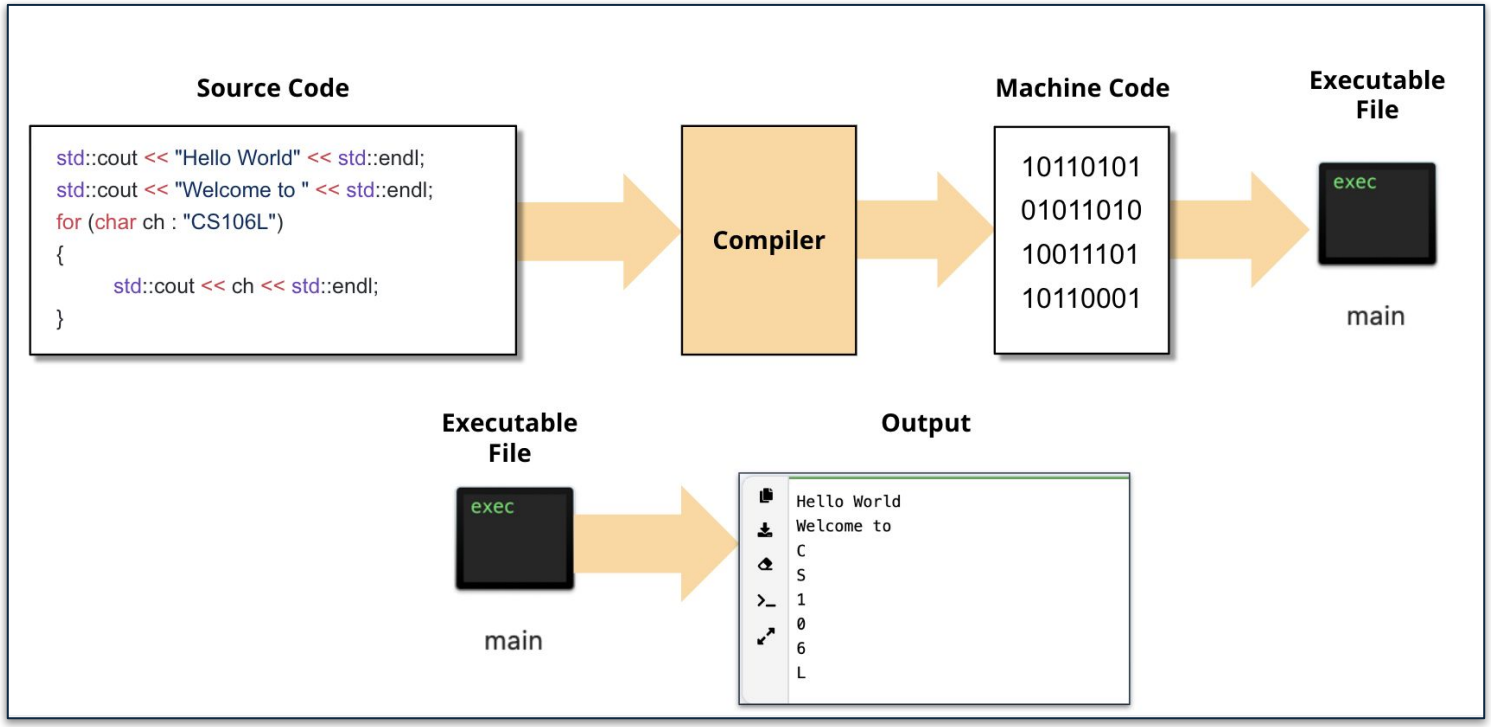
Compiled Languages



Compiled Languages



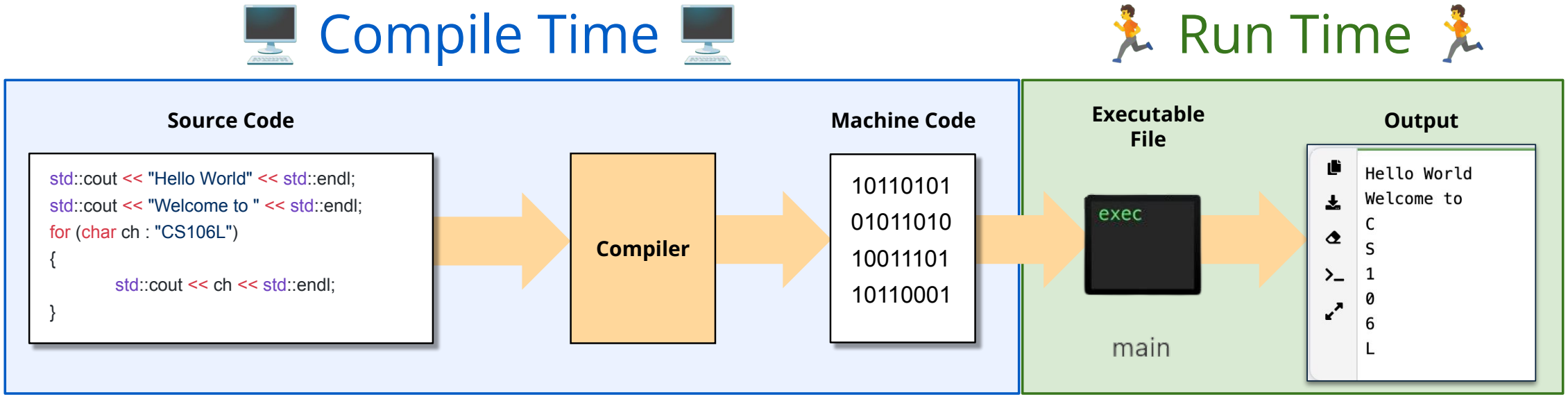
Compiled Languages



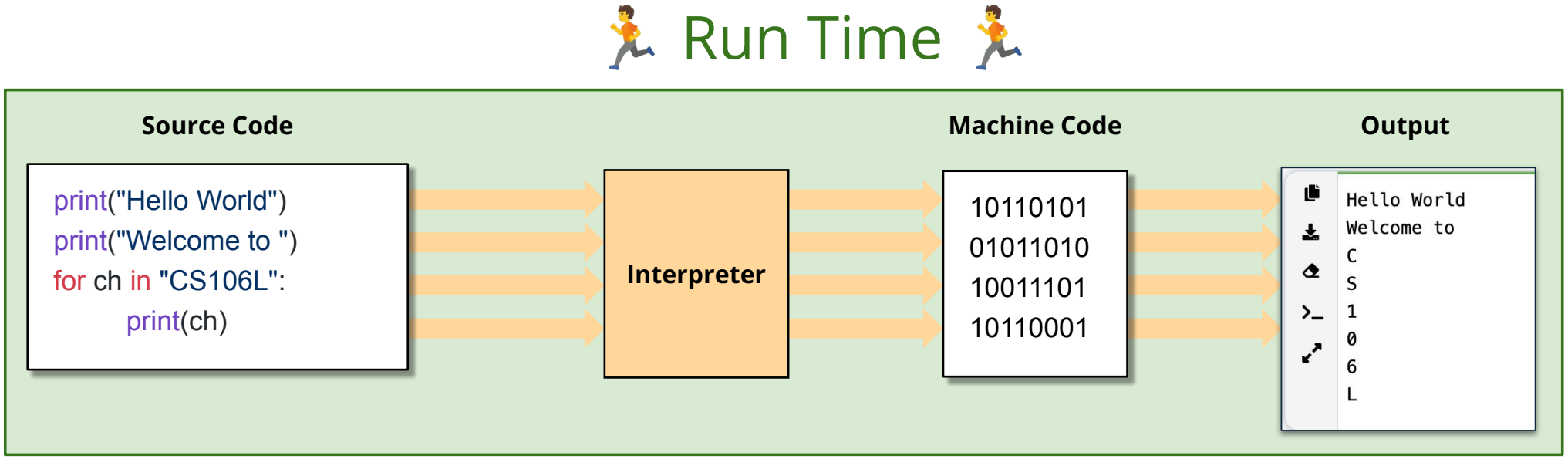
🧠 THE BIG IDEA 🧠

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

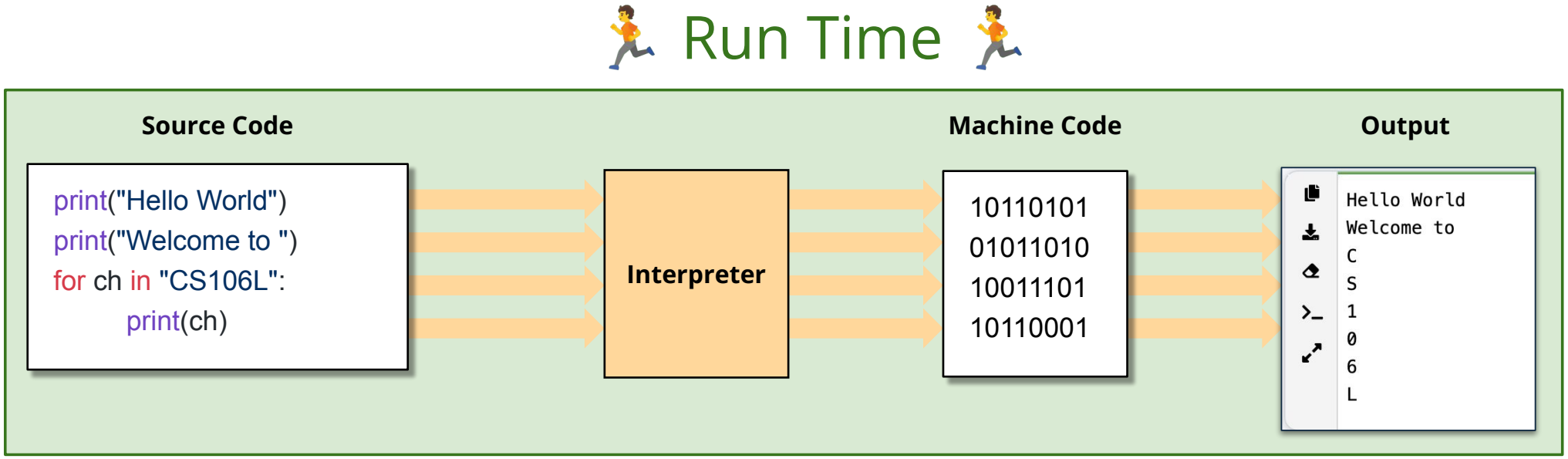
Compiled Languages: Compile Time V.S. Run Time



Interpreted Languages: Compile Time V.S. Run Time



Interpreted Languages: Compile Time V.S. Run Time

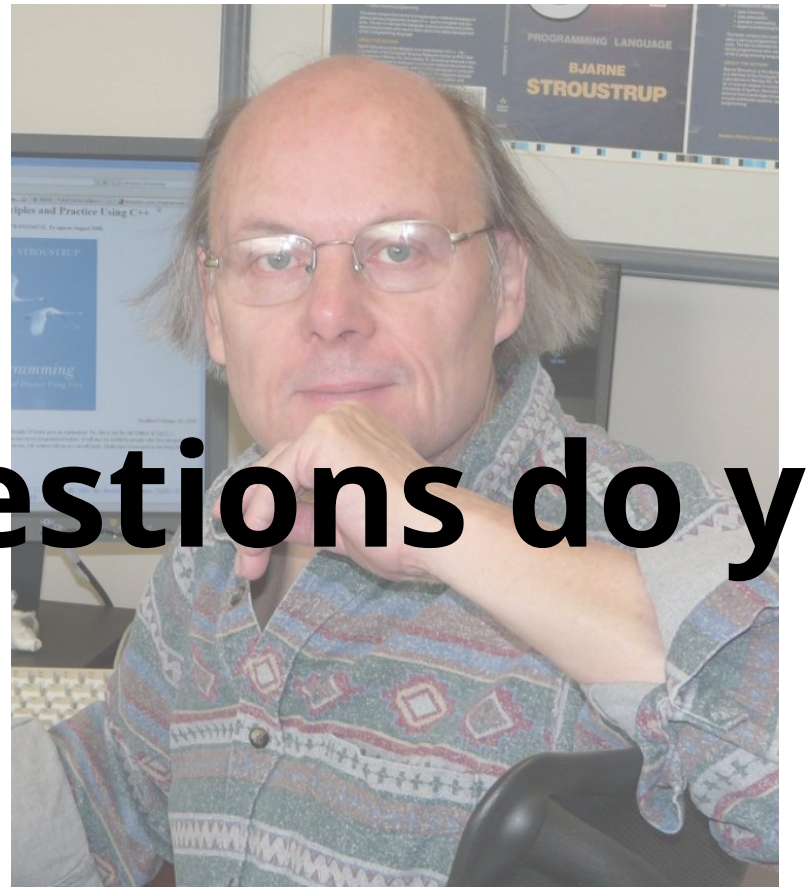


THE BIG IDEA

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'
```

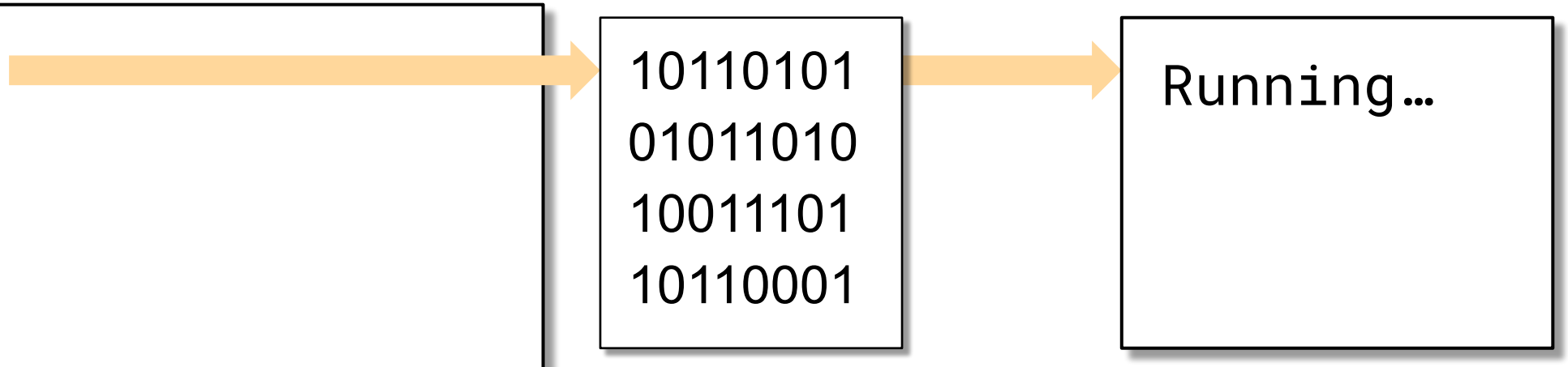
C++

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

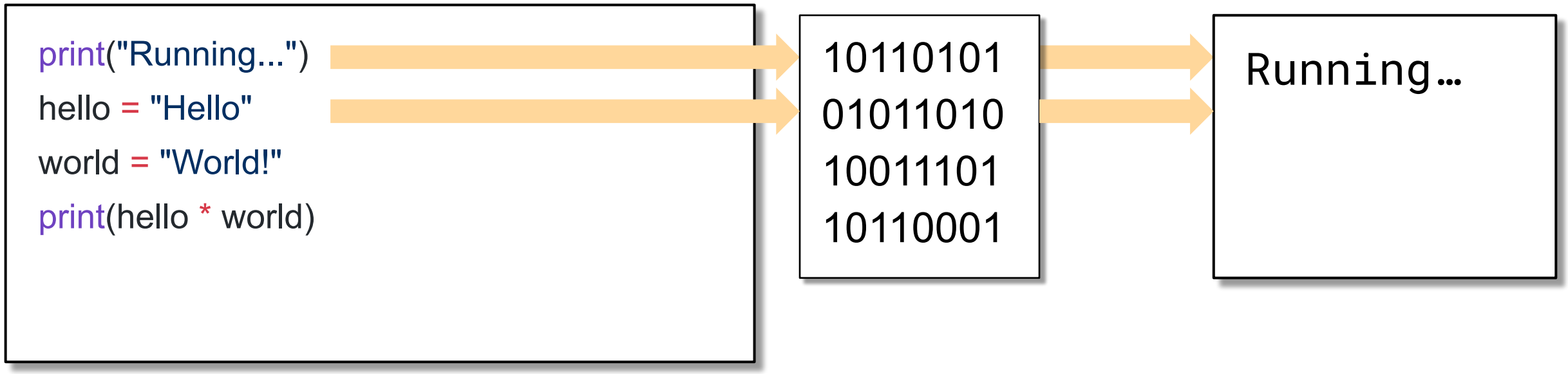
```
$ g++ main.cpp

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

Python

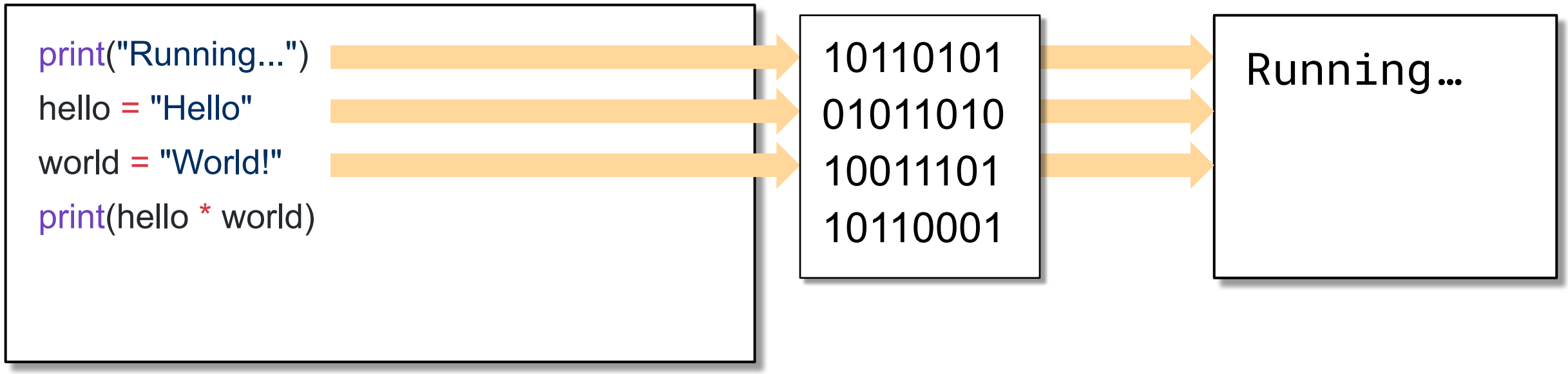
```
print("Running...")
hello = "Hello"
world = "World!"
print(hello * world)
```



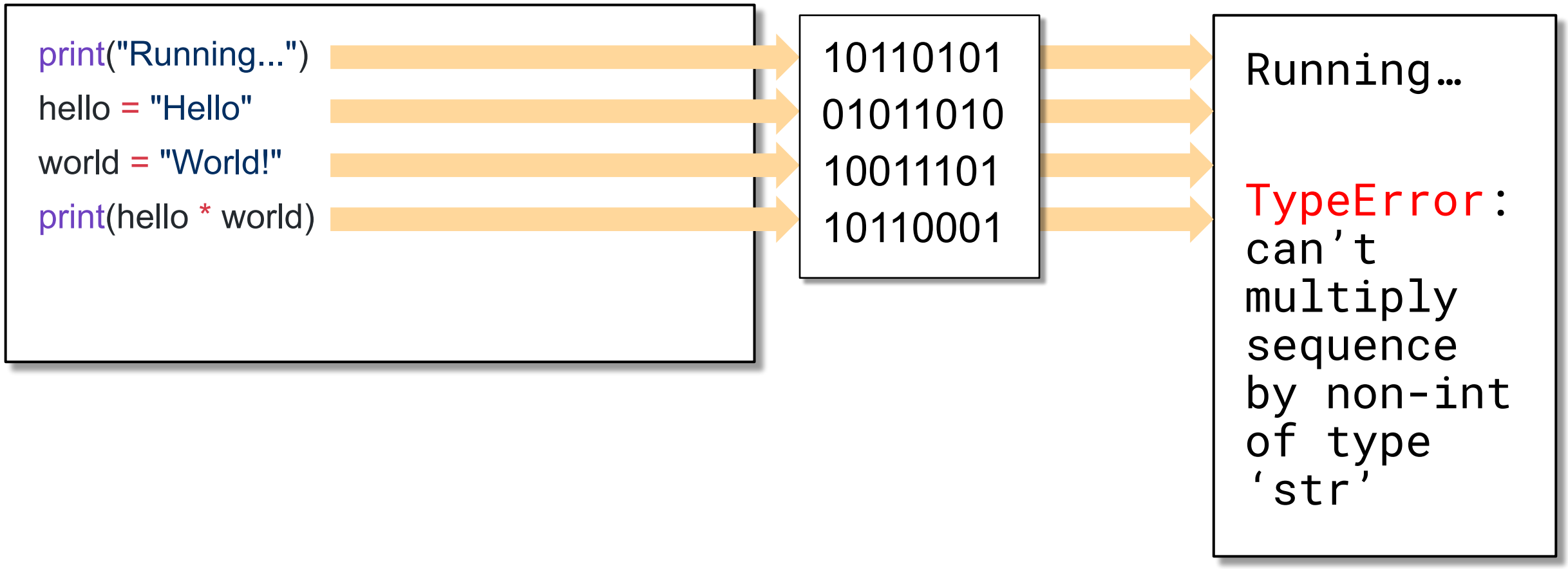
Python


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

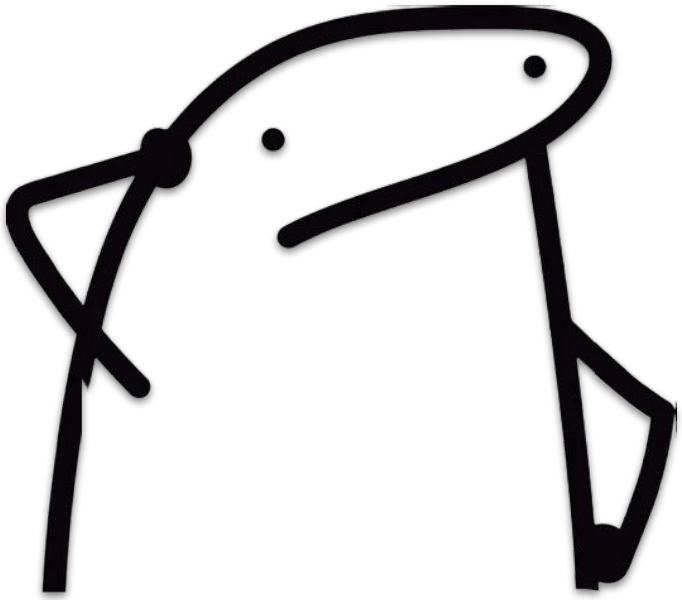
```
print("Running...")
hello = "Hello ";
world = "World!";
print(hello * world)
```

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?



```cpp
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
```

C++

🧠 THE BIG IDEA 🧠

This error occurs during

compile time!

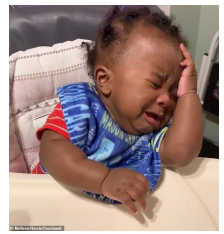
When we are translating, we see that we try to multiply two strings and the compiler goes “hey that’s not allowed!!”

```
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

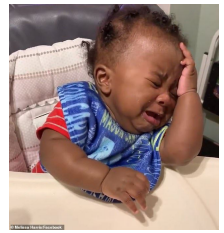
```
$ python3 program.py

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')
```



C++ compilers can be noisy... why?

```
rtmap.cpp: In function `int main()':
rtmap.cpp:19: invalid conversion from `int' to `
std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:19: initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
_Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
std::pair<const int, double>*]'
rtmap.cpp:20: invalid conversion from `int' to `
std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:20: initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
_Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
std::pair<const int, double>*]'
```

[Source]

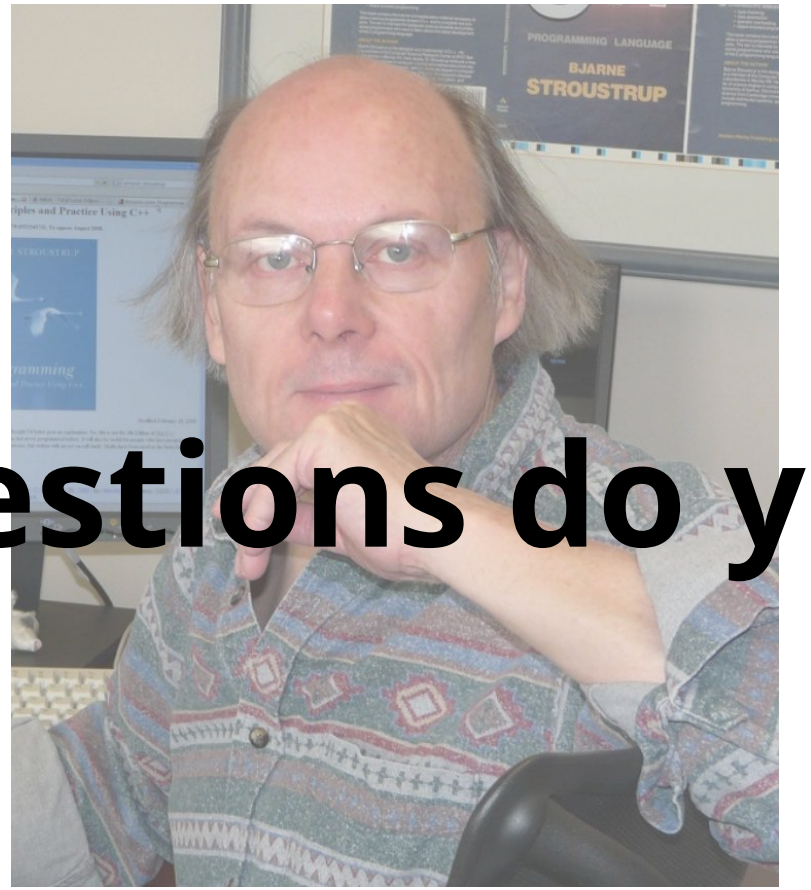
C++ compilers can be noisy... why?

```
rtmap.cpp: In function `int main()':
rtmap.cpp:19: invalid conversion from `int' to `
std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:19: initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
_Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
std::pair<const int, double>*]'
rtmap.cpp:20: invalid conversion from `int' to `
std::_Rb_tree_node<std::pair<const int, double> >*'
rtmap.cpp:20: initializing argument 1 of `std::_Rb_tree_iterator<_Val, _Ref,
_Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*) [with _Val =
std::pair<const int, double>, _Ref = std::pair<const int, double>&, _Ptr =
std::pair<const int, double>*]'
```

Well the compiler is processing all of our types!

[Source]

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

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?

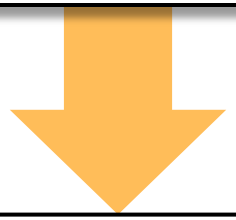
- More efficient
- Easier to understand and reason about
- Better error checking

Better error checking

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

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

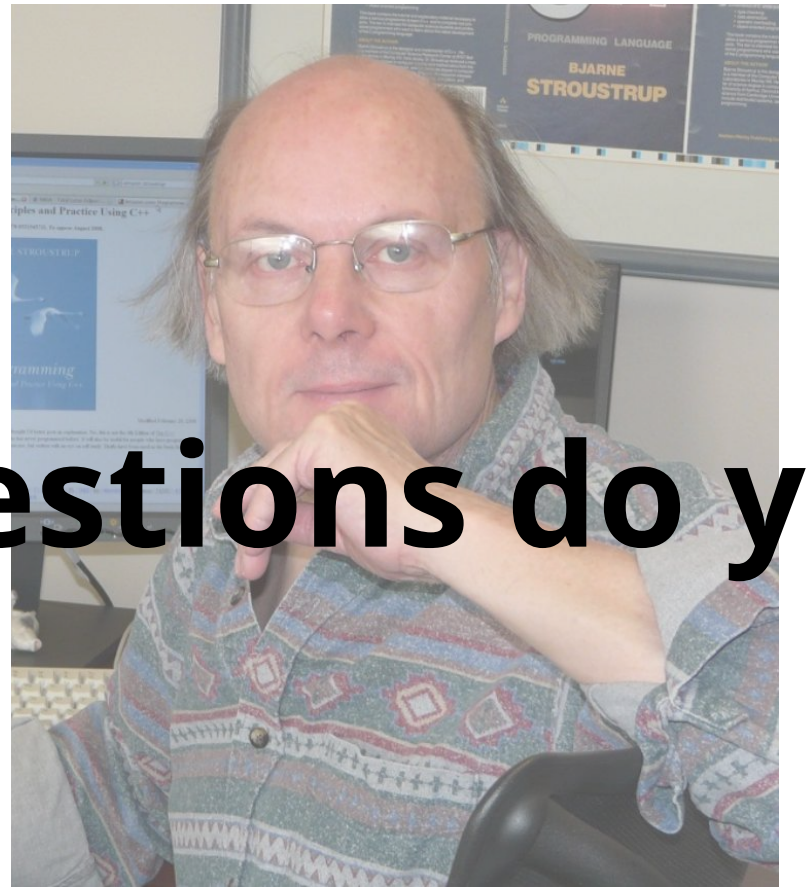
Better error checking

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

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



```cpp
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

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

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

axolotl(2);          // uses version ___, returns _______
axolotl(2.0);   // uses version ___, returns _______
```

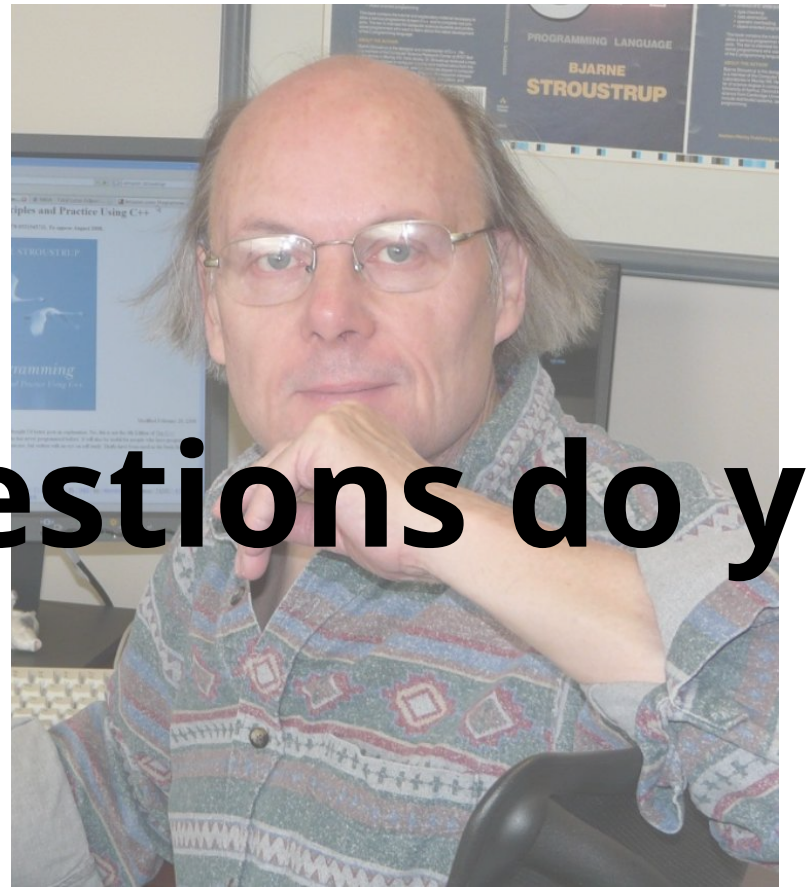
Aside: Function Overloading

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

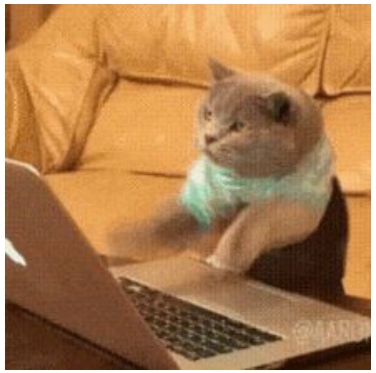
Structs

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 = 0000002;

    return id;
}
```

Uniform 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;
```


```
// Order depends on field order in struct. '=' is optional
StanfordID tree = { "THE Stanford Tree", "theTREE", 0000002 };
StanfordID lelandjr { "Leland Stanford Jr", "thejunior", 5430282 };
```

Using list initialization

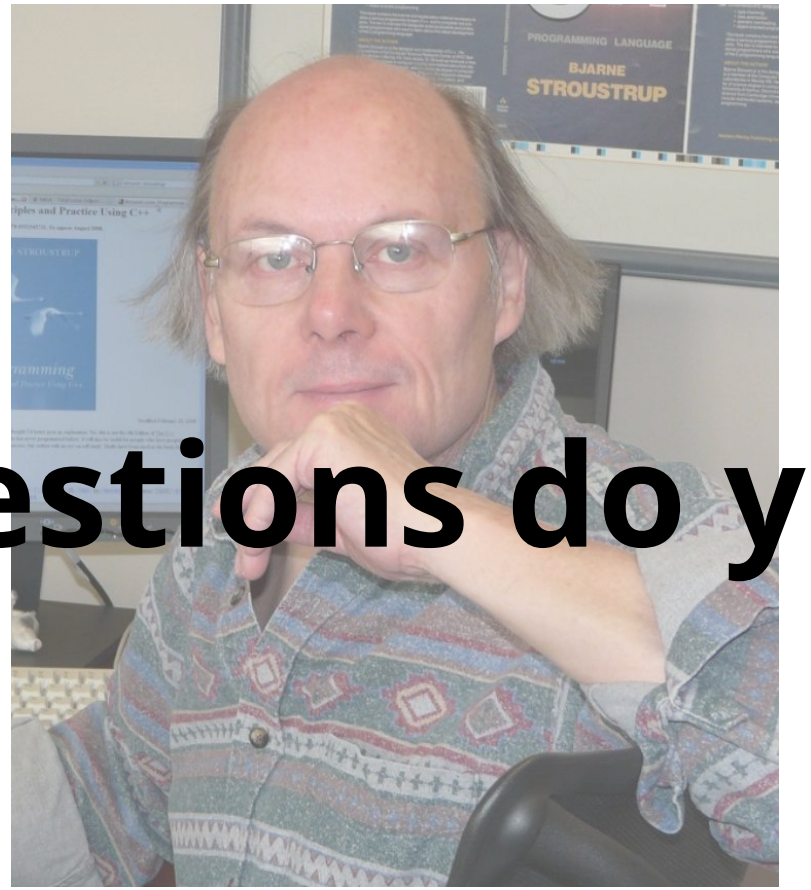
```
StanfordID issueNewID() {
    StanfordID id;
    id.name = "THE Stanford Tree";
    id.sunet = "theTREE";
    id.idNumber = 0000002;
    return id;
}
```

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

🧠 THE BIG IDEA 🧠

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

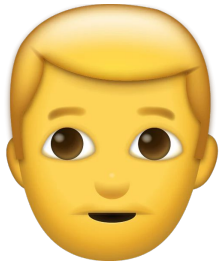
What questions do you have?



bjarne_about_to_raise_hand

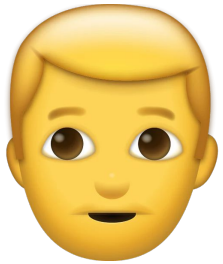
Many Possible Structs

```
struct Name {
        string first;
        string last;
};

Name rf = { "Rachel", "Fernandez" };
```



Many Possible Structs

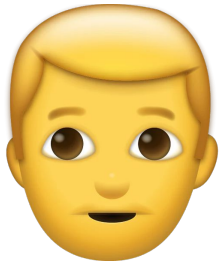
```
struct Name {
    string first;
    string last;
};

Name rf = { "Rachel", "Fernandez" };
```



```
struct Order {
    string item;
    int quantity;
};

Order dozen = { "Eggs", 12 };
```

Many Possible Structs

```
struct Name {
    string first;
    string last;
};

Name rf = { "Rachel", "Fernandez" };
```

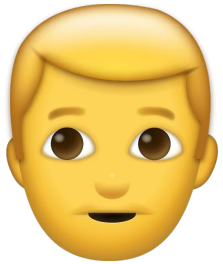


```
struct Order {
    string item;
    int quantity;
};

Order dozen = { "Eggs", 12 };
```

```
struct Point {
    double x;
    double y;
};

Point origin { 0.0, 0.0 };
```

Many Possible Structs

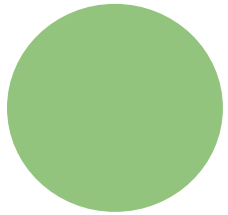
```
struct Name {
    string first;
    string last;
};

Name rf = { "Rachel", "Fernandez" };
```



```
struct Order {
    string item;
    int quantity;
};

Order dozen = { "Eggs", 12 };
```

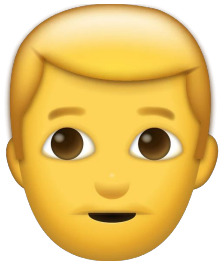
```
struct Point {
    double x;
    double y;
};

Point origin { 0.0, 0.0 };
```

```
struct Circle {
    Point center;
    double radius;
};

Circle circle { {0, 0} , 50000000 };
```



Many Possible Structs

```
struct Name {
    string first;
    string last;
};

Name rf = { "Rachel", "Fernandez" };
```

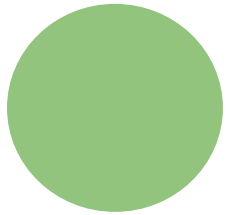


```
struct Order {
    string item;
    int quantity;
};

Order dozen = { "Eggs", 12 };
```

Notice anything?

```
struct Point {
    double x;
    double y;
};

Point origin { 0.0, 0.0 };
```

```
struct Circle {
    Point center;
    double radius;
};

Circle circle { {0, 0} , 50000000 };
```



Many Possible Structs

```
struct Name {
    string first;
    string last;
};

Name jrb = { "Rachel", "Fernandez" };
```

```
struct Order {
    string item;
    int quantity;
};

Order dozen = { "Eggs", 12 };
```

Notice anything?

```
struct Point {
    double x;
    double y;
};

Point origin { 0.0, 0.0 };
```

```
struct Circle {
    Point center;
    double radius;
};

Circle circle { {0, 0} , 50000000 };
```

Erm these all look a bit similar!!

We can use std::pair!

std::pair

```
struct Order {
        std::string item;
        int quantity;
};

Order dozen = { "Eggs", 12 };
```

std::pair

```
struct Order {
        std::string item;
        int quantity;
};

Order dozen = { "Eggs", 12 };
```

```
std::pair<std::string, int> dozen { "Eggs", 12 };
std::string item = dozen.first;                    // "Eggs"
int quantity = dozen.second;                       // 12
```

std::pair is a template

(We'll learn more about this later)

```
template <typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
};
std::pair<std::string, int>
```

std::pair is a template

(We'll learn more about this later)

```
struct pair {
    std::string first;
    int second;
};
```

There’s something we need to discuss…

What is an std !!? 🦠😷

std — The C++ Standard Library

- Built-in types, functions, and more provided by C++
- You need to #include the relevant file
  - #include <string> → std::string
  - #include <utility> → std::pair
  - #include <iostream> → std::cout, std::endl
- We prefix standard library names with std::
  - If we write using namespace std; we don't have to, but this is considered bad style as it can introduce ambiguity
    - (What would happen if we defined our own string?)

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::pair, you must #include it

std::pair is defined in a header file called utility

```
#include <utility>

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

std::pair<double, double> point { 1.0, 2.0 };
```

What does #include do?

```
#include <utility>

std::pair<double, double> p { 1.0, 2.0 };
```

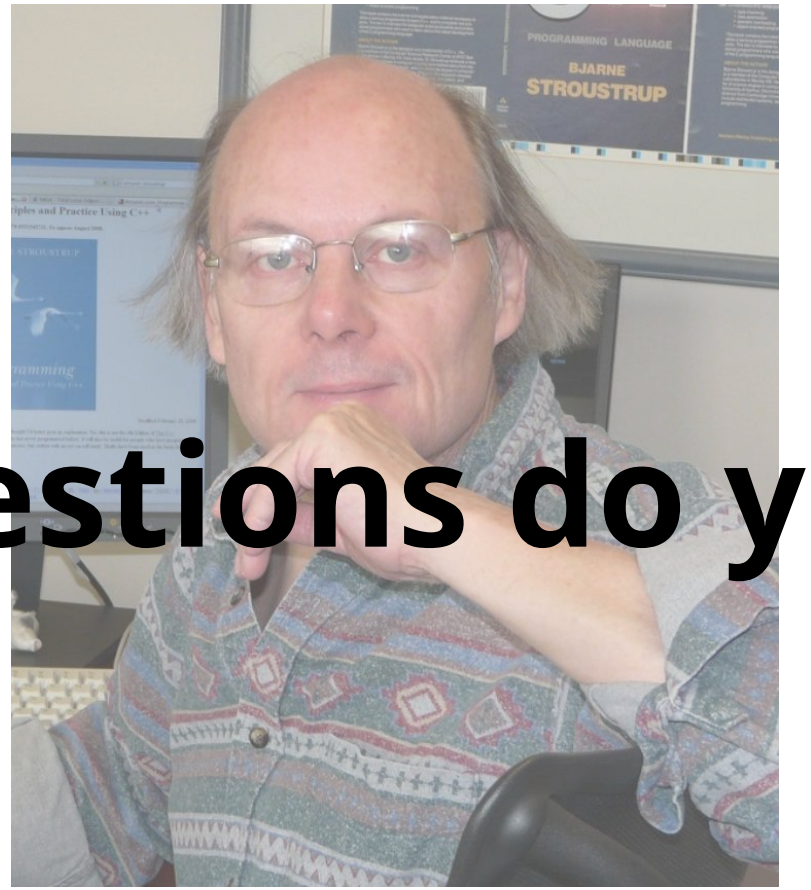
utility

```
namespace std {

    template
    <typename T1, typename T2>
    struct pair {
        T1 first;
        T2 second;
    };

    // Other utility code...
}
```

What does #include do?

```
namespace std {

    template <typename T1, typename T2>
    struct pair {
        T1 first;
        T2 second;
    };

    // Other utility code...
}

std::pair<double, double> p { 1.0, 2.0 };
```

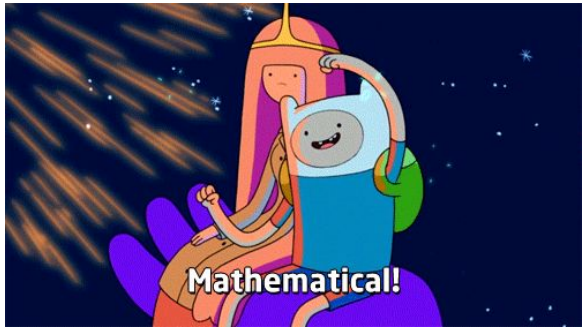
What questions do you have?

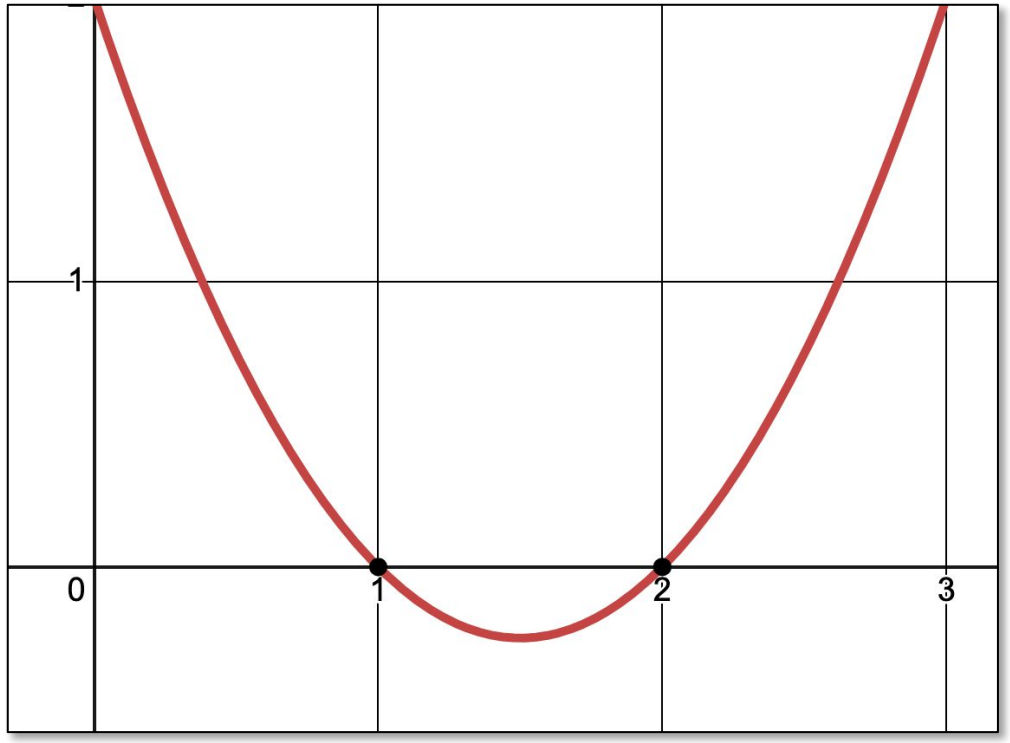


bjarne_about_to_raise_hand

Code Demo

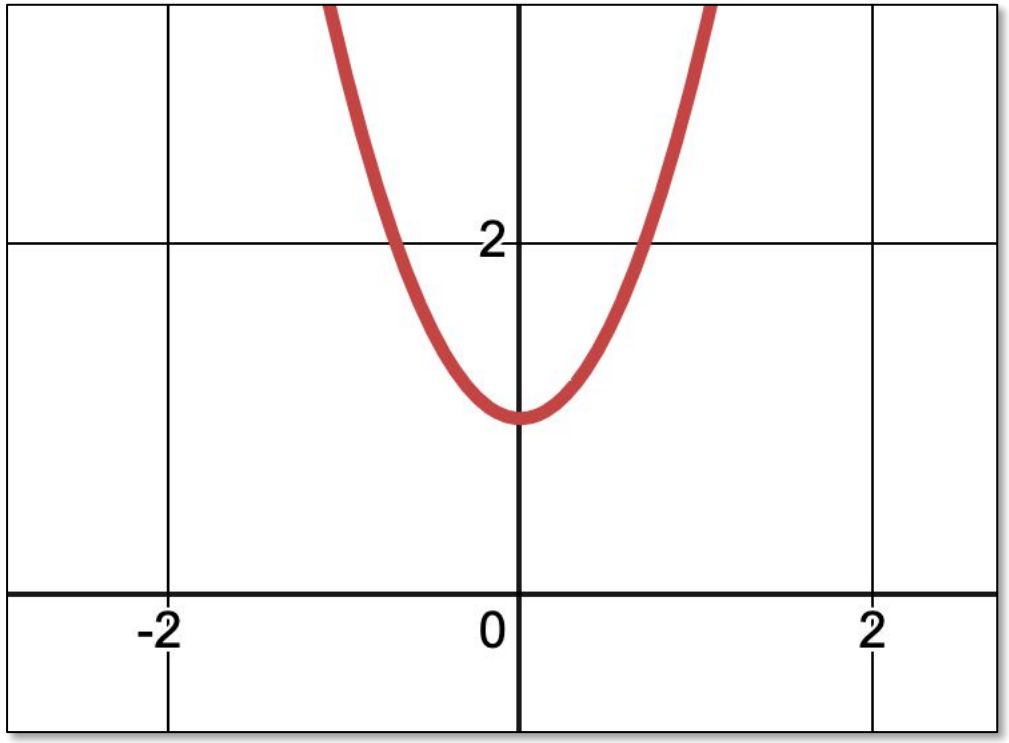
Solving a Quadratic Equation





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

$$x = 1, x = 2$$

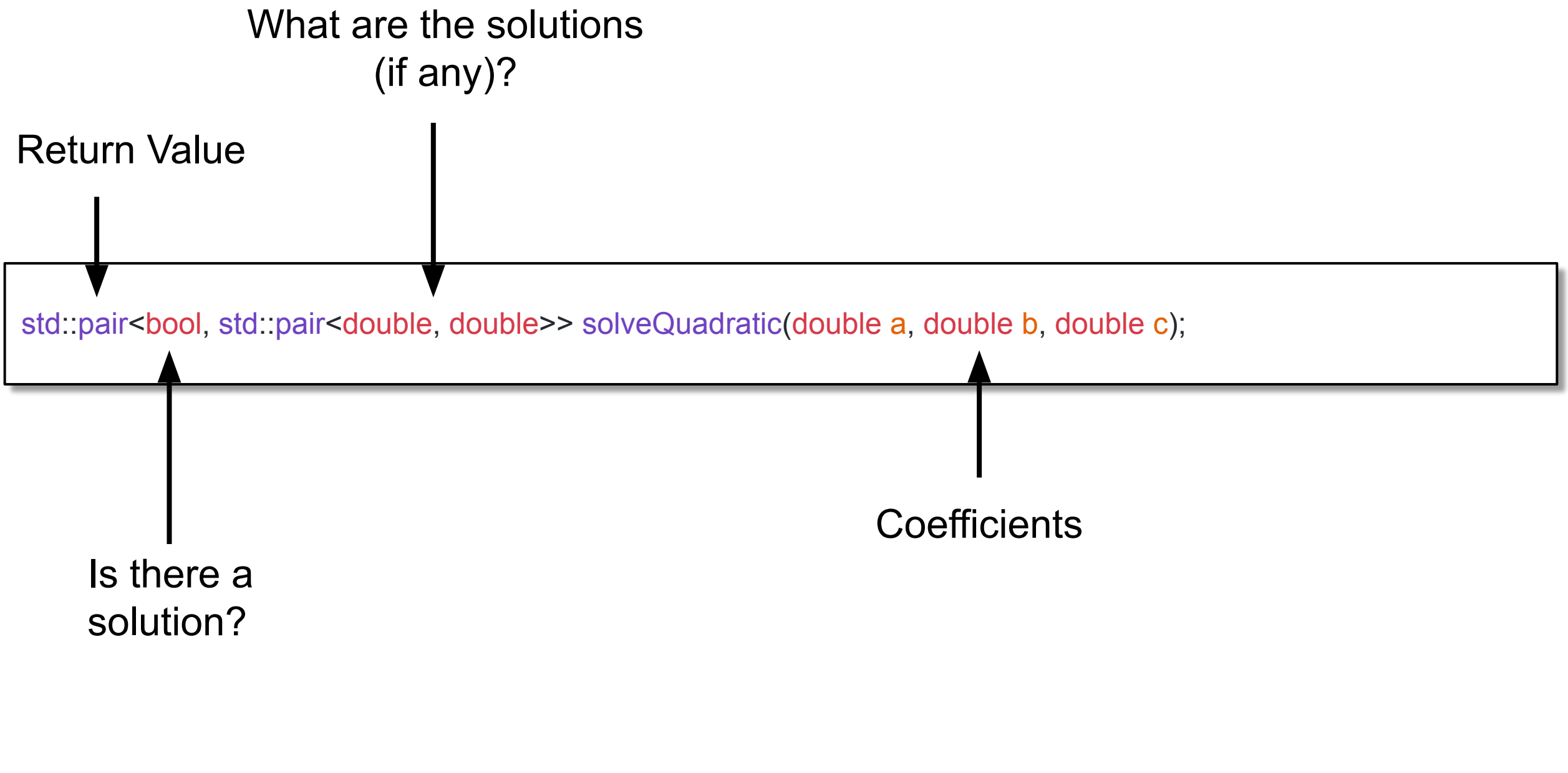


$$2x^2 + 1 = 0$$

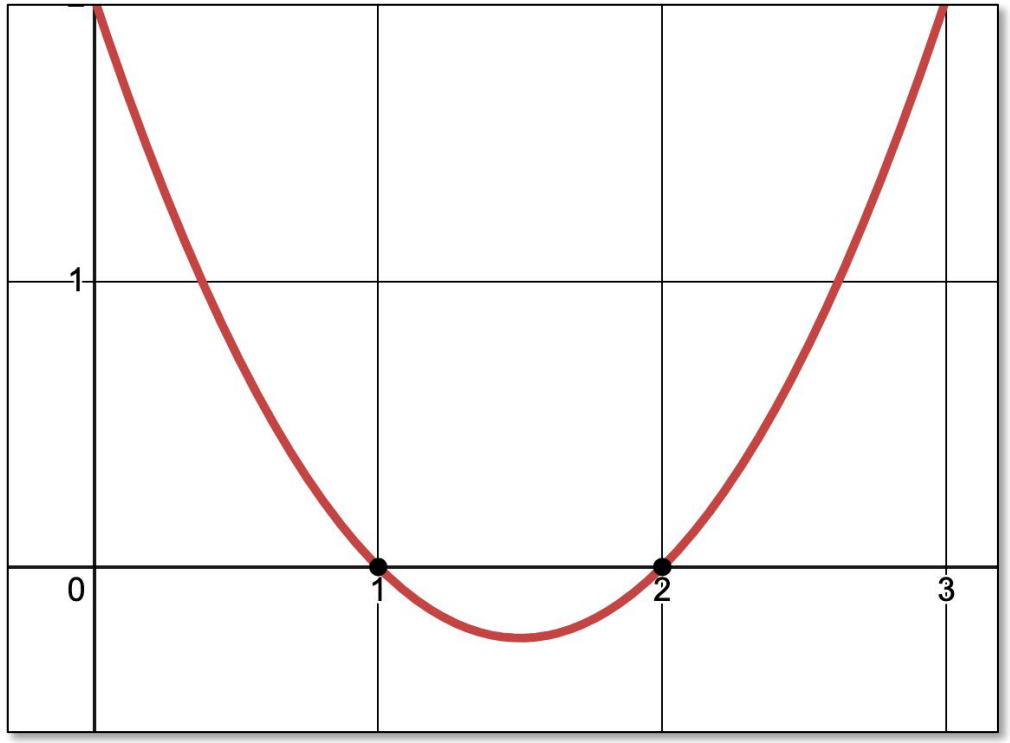
$$no\ solution$$

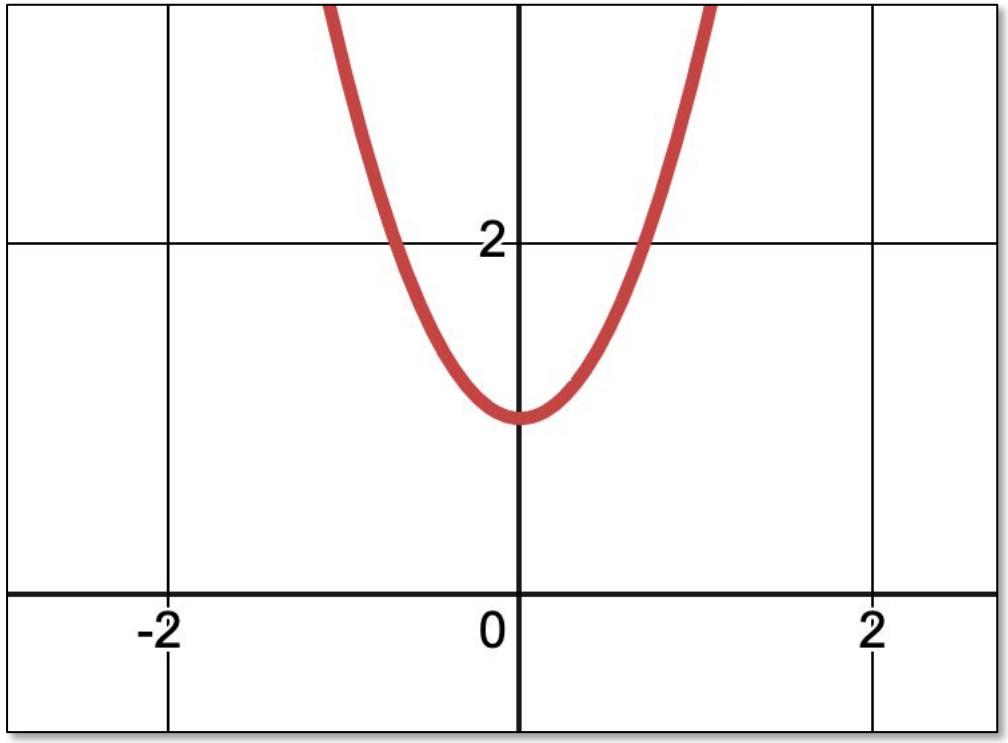
Solving a Quadratic Equation

- 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
```

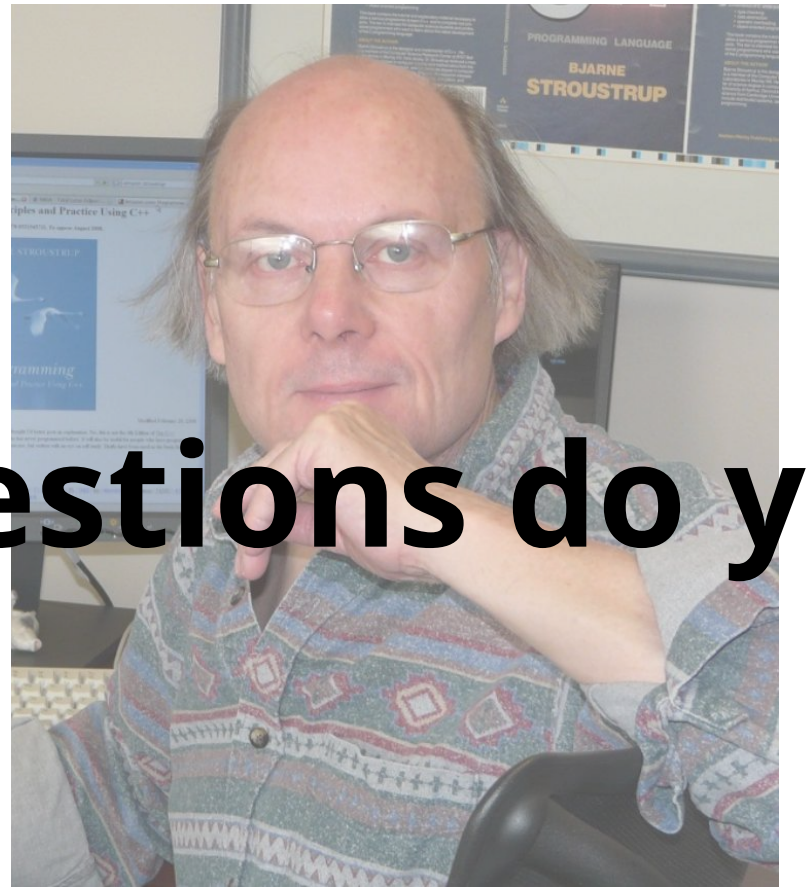
Which one is clearer?

```
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
- **std::pair** is a general purpose struct with two fields
- `#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