Structure of a C++ Program

```cpp
// Include other libraries, similar to Python's "import"
#include <iostream>
#include <utility>
#include <cmath>

// Main logic of your program goes here
int main() {
	std::cout << "Hello World" << std::endl;
	std::cout << "Welcome to " << std::endl;
	for (char ch : "CS106L")
	{
		std::cout << ch << std::endl;
	}
}
```

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

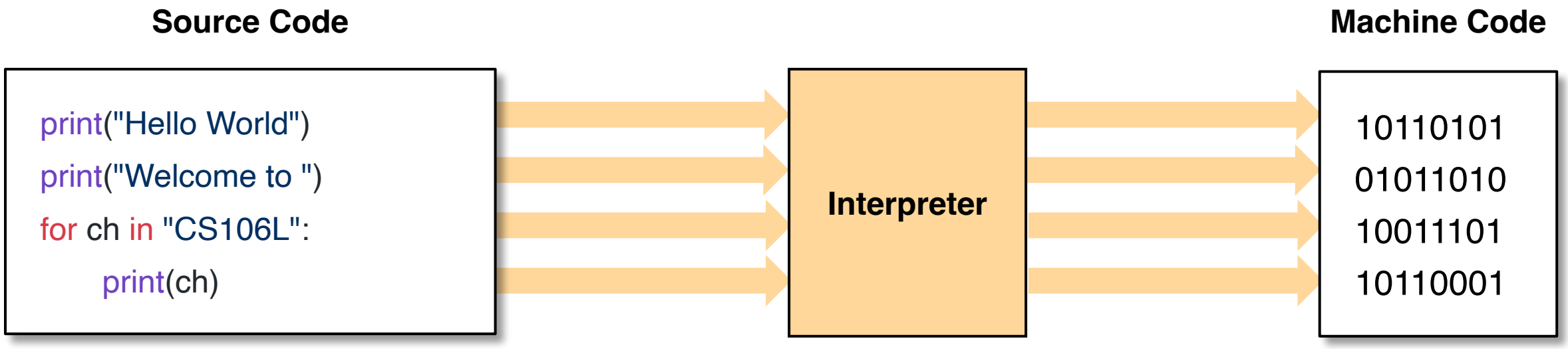
Python

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

C++

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

Interpreted Languages

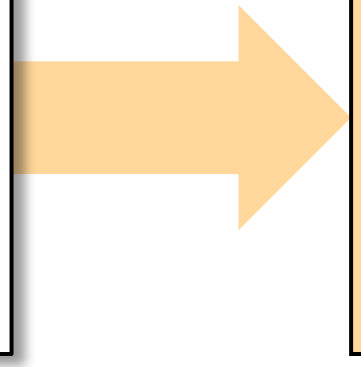



```
$ python3 main.py # python3 is the interpreter
```

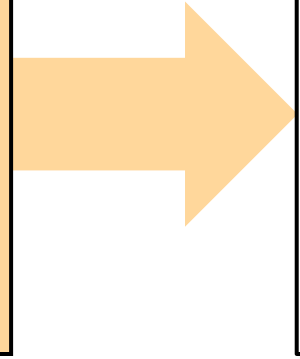
Compiled Languages

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
```

```
$ g++ main.cpp –o main  # g++ is the compiler, outputs binary to main
$ ./main                # This actually runs our program
```

Why compile over interpret?

- It allows us to generate more efficient machine code!
  - Interpreters only see one small part of code at a time
  - Compilers see everything
- However, compilation takes time!

Compile time vs. runtime

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

Compile Time

```
10110101
01011010
10011101
10110001
```

Runtime

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')
```

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

Static Typing

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

Python (Dynamic Typing)

```
a = 3
b = "test"

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

C++ (Static Typing)

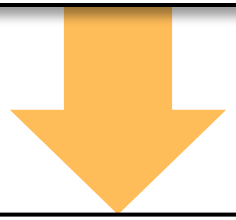
```
int a = 3;
string b = "test";

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

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!
```

Aside: Function Overloading

Defining two functions with the same name but different signatures

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

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

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

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 = "Jacob Roberts-Baca"; // Access field with '.'
id.sunet = "jtrb";
id.idNumber = 6504417;
```

Returning multiple values

```
StanfordID issueNewID() {
    StanfordID id;
    id.name = "Jacob Roberts-Baca";
    id.sunet = "jtrb";
    id.idNumber = 6504417;
    return id;
}
```

List Initialization

```
StanfordID id;
id.name = "Jacob Roberts-Baca";
id.sunet = "jtrb";
id.idNumber = 6504417;
```

```
// Order depends on field order in struct. ‘=‘ is optional
StanfordID jrb = { "Jacob Roberts-Baca", "jtrb", 6504417 };
StanfordID fi { ”Fabio Ibanez", ”fibanez", 6504418 };
```

Using list initialization

```
StanfordID issueNewID() {
    StanfordID id = { "Jacob Roberts-Baca", "jtrb", 6504417 };
    return id;
}
```

```
StanfordID issueNewID() {
    return { "Jacob Roberts-Baca", "jtrb", 6504417 };
}
```

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

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

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 };
```

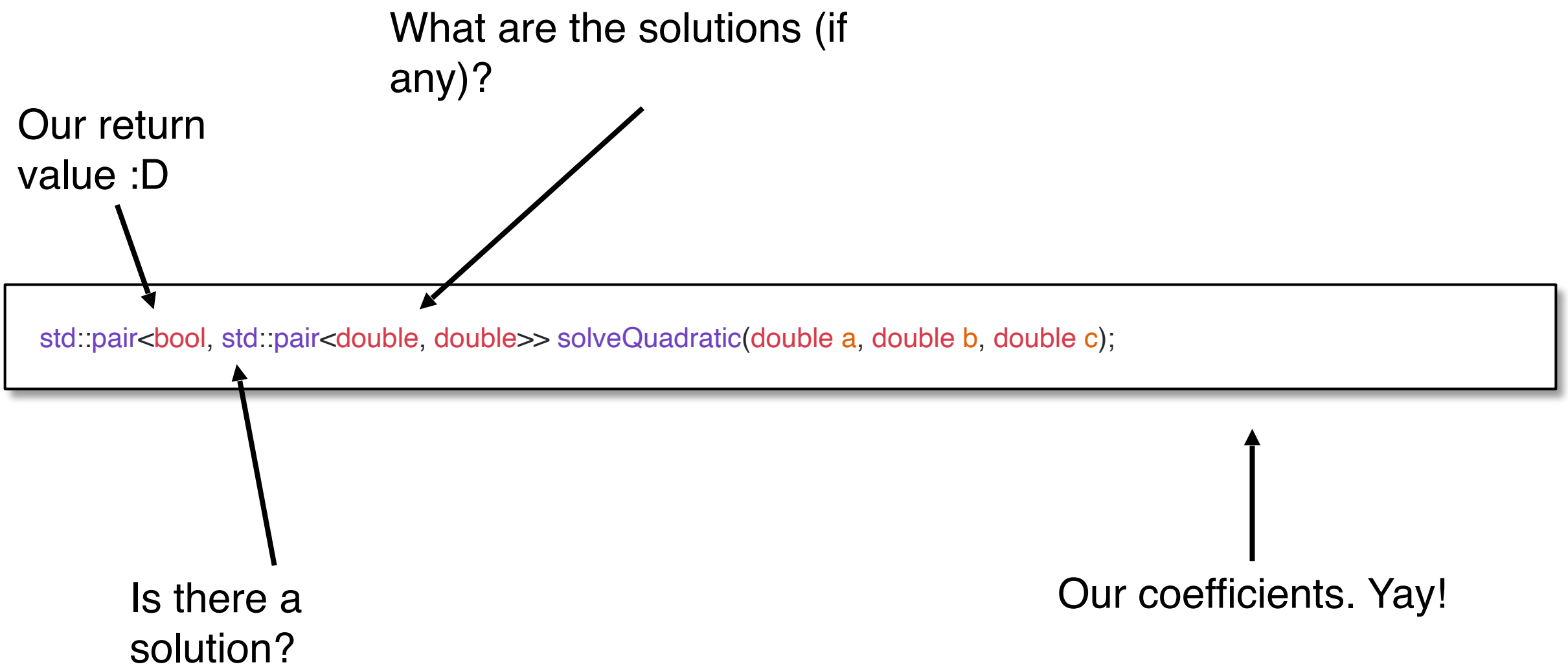
Solving a Quadratic Equation

- 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


What are the solutions (if any)?
Our return value :D
std::pair<bool, std::pair<double, double>> solveQuadratic(double a, double b, double c);
Is there a solution?
Our coefficients. Yay!

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>;
```

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!
```

auto is still statically typed!

```
auto i = 1;     // int inferred
i = "hello!";   // ❌ Doesn't compile
```

```
#include <iostream>
#include <cmath>
#include <utility>

using Zeros = std::pair<double, double>;
using Solution = std::pair<bool, Zeros>;

Solution solveQuadratic(double a, double b, double c)
{
  double discrim = b * b - 4 * a * c;
  if (discrim < 0) return { false, { 106, 250 }};
  discrim = sqrt(discrim);
  return { true, { (-b - discrim) / (2 * a), (-b + discrim) / (2 * a) }};
}

int main() {
 double a, b, c;
  std::cout << "a: "; std::cin >> a;
  std::cout << "b: "; std::cin >> b;
  std::cout << "c: "; std::cin >> c;

  auto result = solveQuadratic(a, b, c);
  if (result.first) {
    auto solutions = result.second;
    std::cout << "Solutions: " << solutions.first << ", " << solutions.second << std::endl;
  } else {
    std::cout << "No solutions" << std::endl;
  }

  return 0;
}
```

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