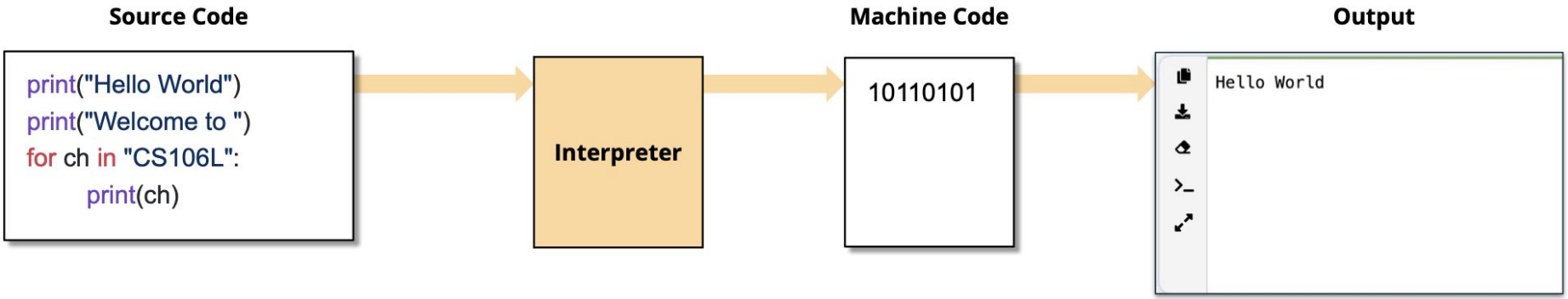
Lec-2: C++ Program Structure

Bernard Nongpoh

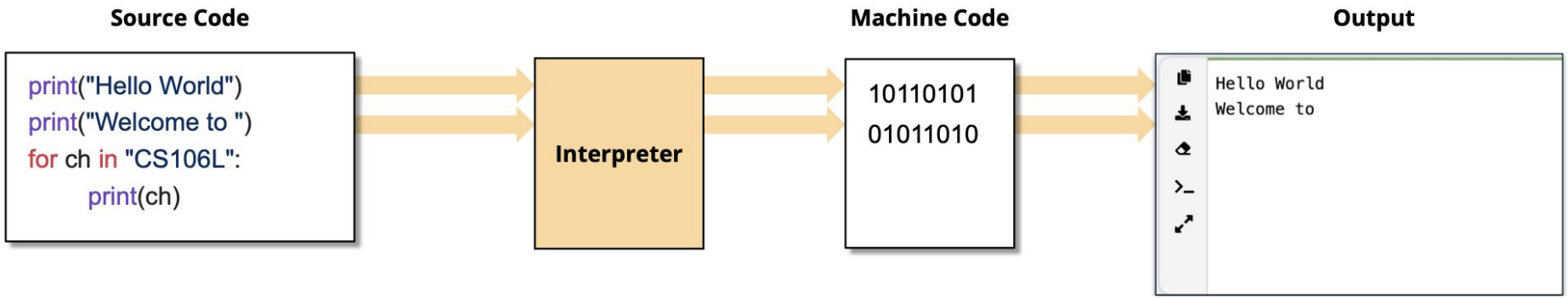
Compiler versus Interpreter

Interpreter Languages



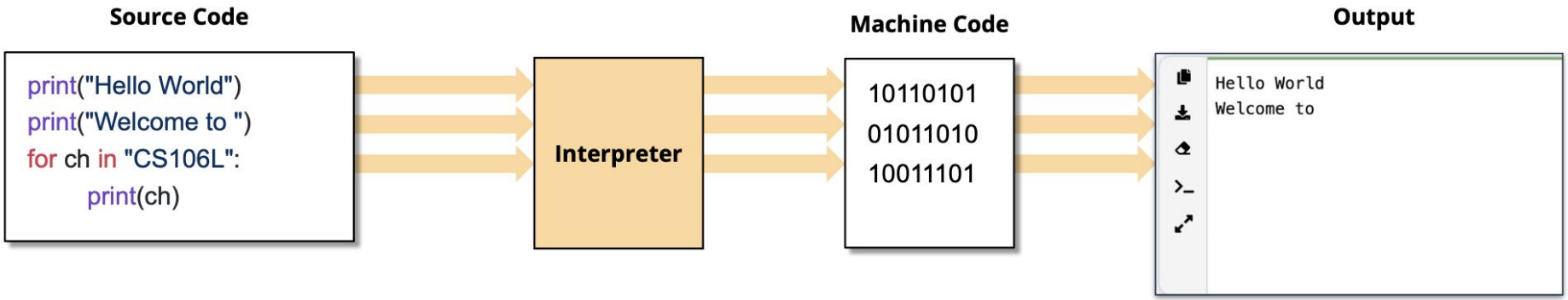
Source

Interpreter Languages



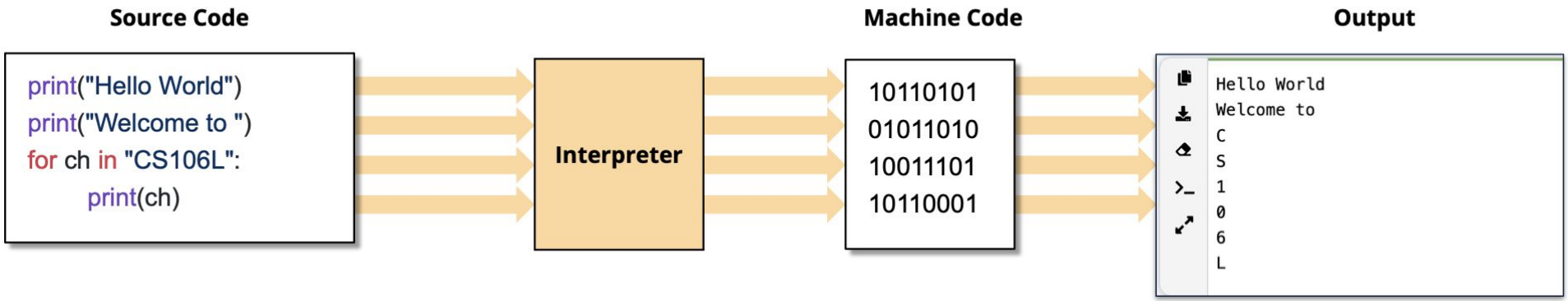
Source

Interpreter Languages



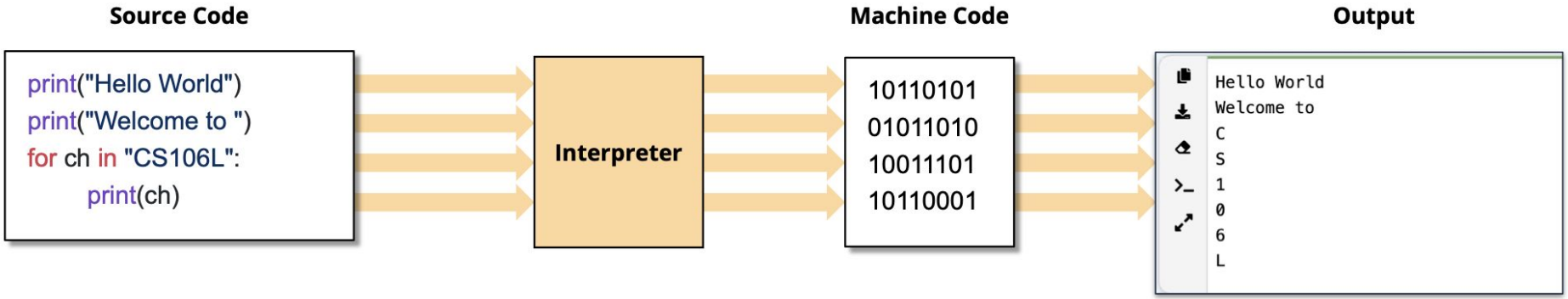
Source

Interpreter Languages



Source

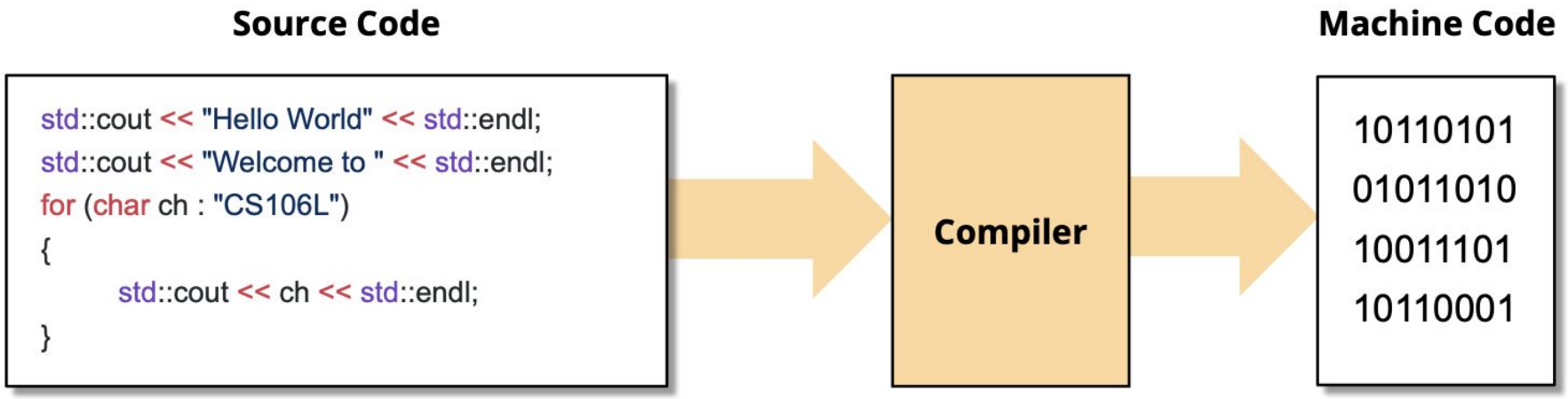
Interpreter Languages



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

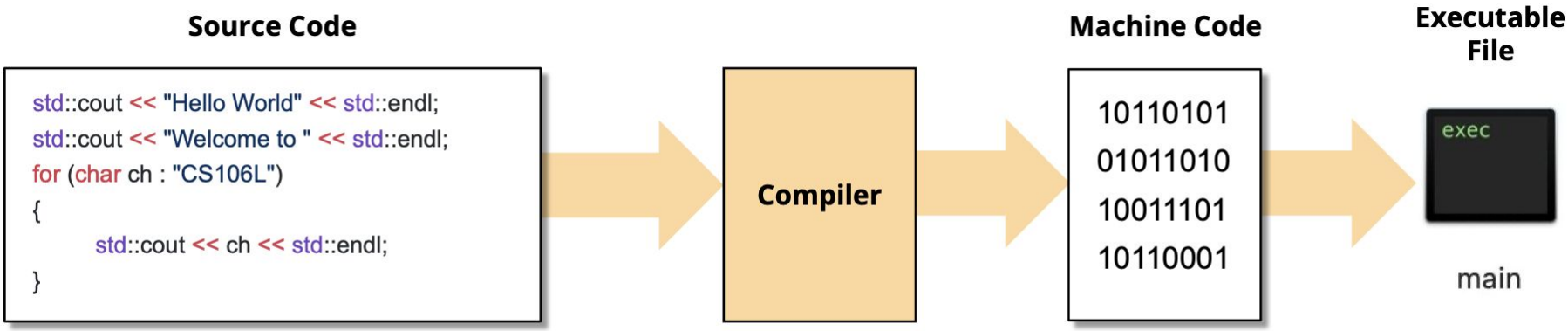
Source

Compiled Languages



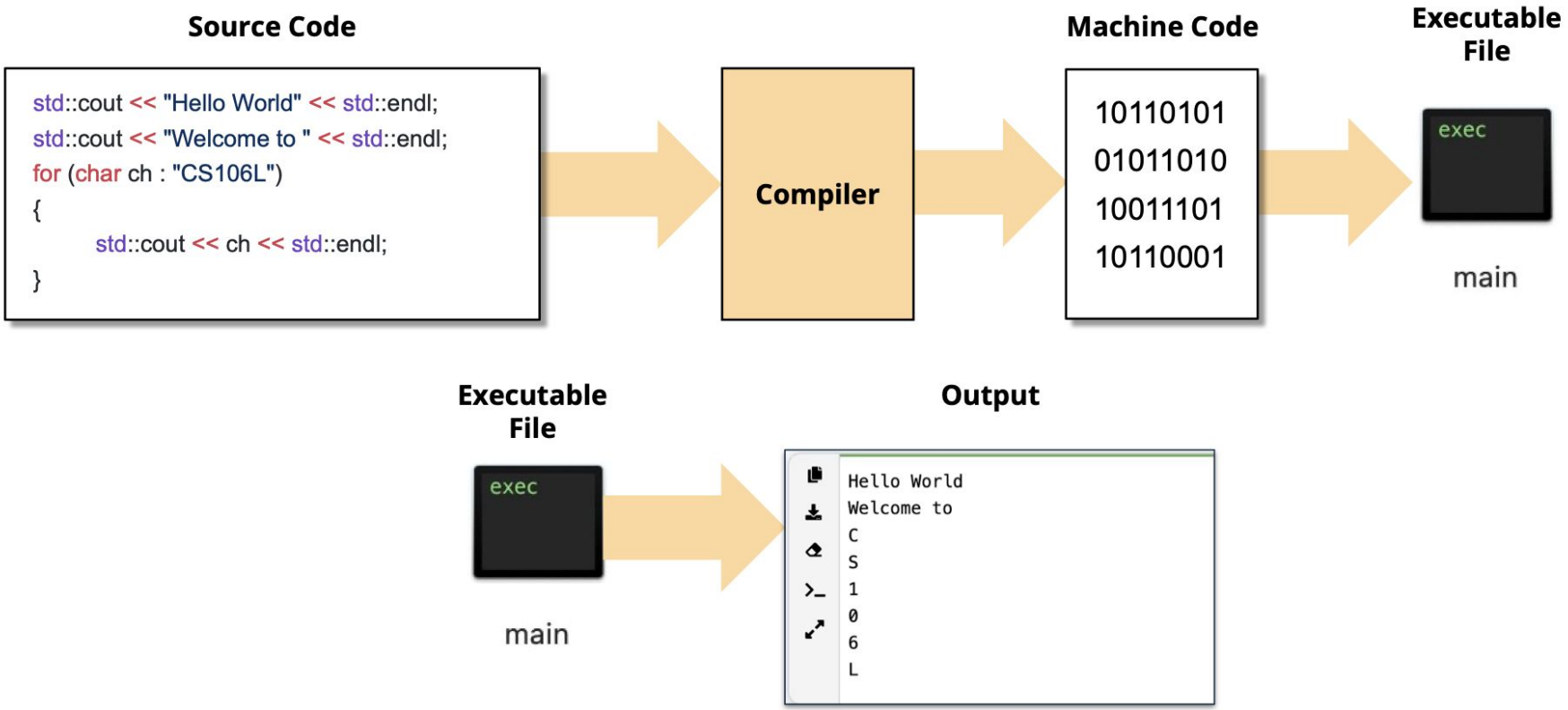
Source

Compiled Languages



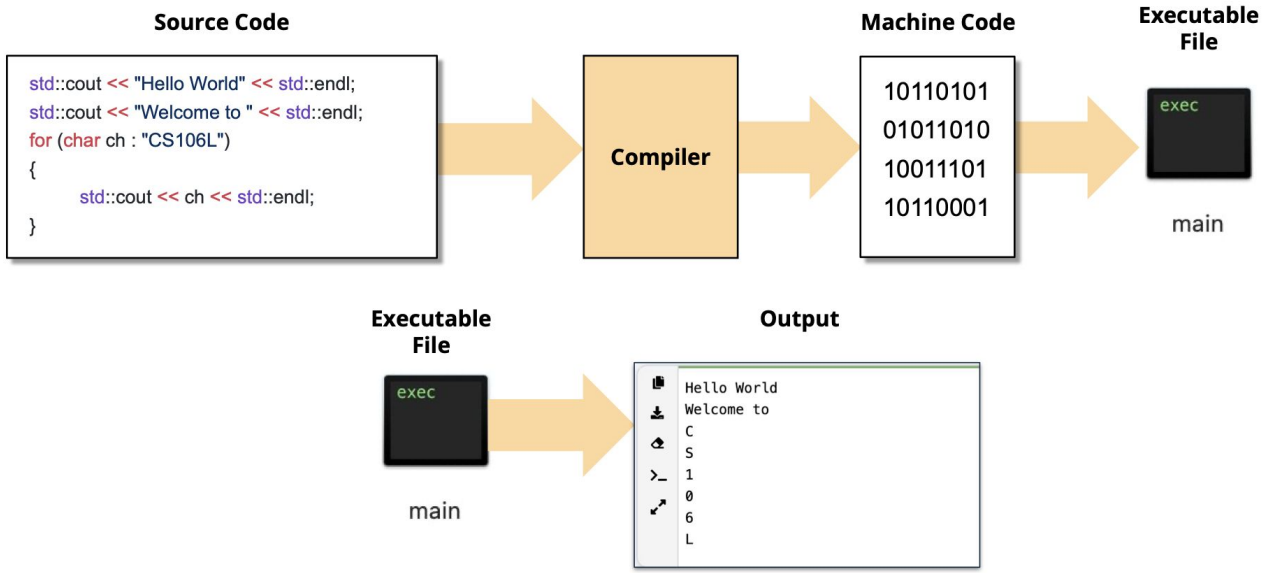
Source

Compiled Languages



Source

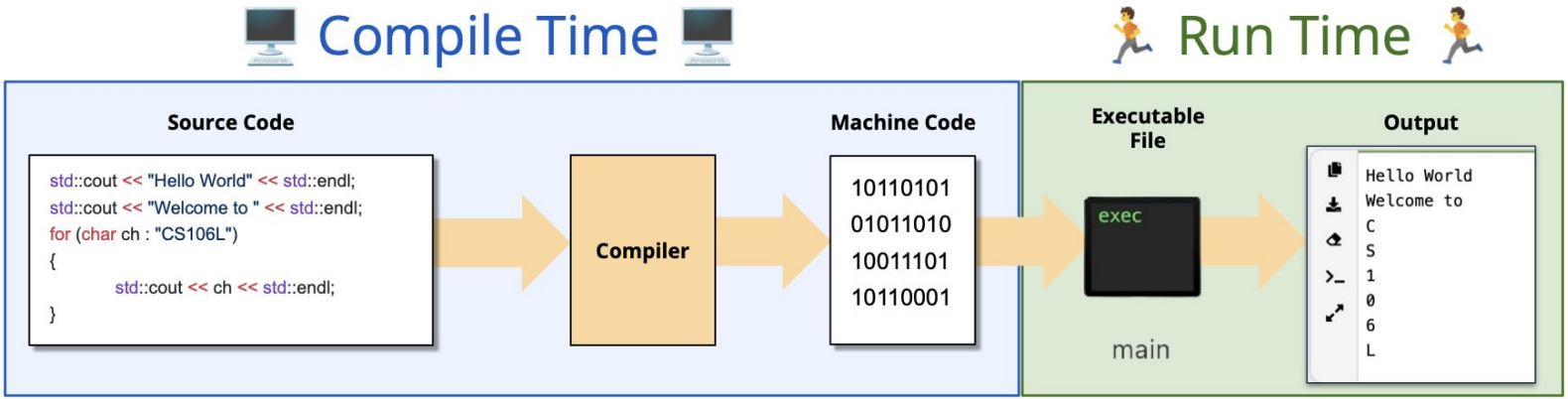
Compiled Languages



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

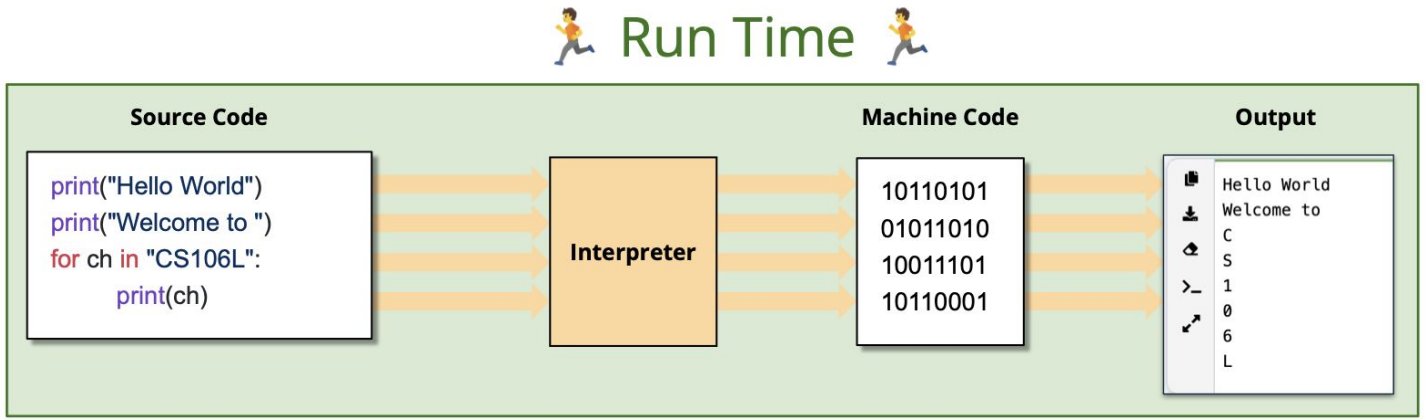
Source

Compiled Languages: Compile Time versus Run Time



Source

Interpreted Languages: Compile Time versus Run Time



Source

Compiler versus Interpreter

Feature	Compiled Language	Interpreted Language
Translation	Whole code converted to machine code before execution	Code translated line-by-line at runtime
Speed	**Faster** execution	**Slower** execution
Error Detection	Errors shown **after compilation**	Errors shown **line by line during execution**
Output	Generates **executable file** (.exe, binary, etc.)	No standalone executable produced
Memory Usage	More memory efficient	Uses more memory due to interpreter overhead
Portability	Less portable (platform dependent binary)	More portable (needs only interpreter)
Examples	C, C++, Rust, Go	Python, Ruby, JavaScript, PHP

Any Questions?

C++ is a compiled language

Compile Time Versus Run Time Errors

```
#include<iostream>
int main(){
  std::cout<<"Hello, World"<<std:endl;
  return 0;
}
```

Run time or compile time?

Compile Time Versus Run Time Errors

```cpp
#include<iostream>
int main(){
  std::cout<<"Hello, World"<<std:endl;
  return 0;
}
```

Compile Time Error!

```
~/w/T/cs1214/code/lec-1 g++ test.cpp
test.cpp:3:33: error: unexpected ':' in nested name specifier; did you mean '::'?
    3 |   std::cout<<"Hello, World"<<std:endl;
      |                                 ^
      |                                 ::
1 error generated.
```

Compile Time Versus Run Time Errors

```cpp
#include <iostream>

int main() {
    int *ptr = nullptr; // null pointer
    std::cout << *ptr << std::endl; // dereferencing null -> Segmentation fault
    return 0;
}
```

```
~/w/T/c/c/lec-1  g++ segfault.cpp
~/w/T/cs1214/code/lec-1  ./a.out
[1]    9990 segmentation fault  ./a.out
```

Compile Time Versus Run Time Errors

```cpp
#include <iostream>

int main() {
    int *ptr = nullptr; // null pointer
    std::cout << *ptr << std::endl; // dereferencing null -> Segmentation fault
    return 0;
}
```

```
~/w/T/c/c/lec-1  g++ segfault.cpp
~/w/T/cs1214/code/lec-1  ./a.out
[1]    9990 segmentation fault  ./a.out
```

Run Time Error!

C++ compilers can be noisy.. why?

C++ compilers can be noisy.. Why?
The compiler is processing all of our types

[Source]

Types are super important!

[Source]

Types

- A **type** in C++ defines a set of **values** and a set of **operations** that can be performed on those values.
- It tells the **compiler** how much memory to allocate and how to **interpret** the stored data.
- It determines **valid operations** (e.g., arithmetic, comparison) and **type safety**.

[Source]

Types

- Examples:
  - `int` → integer values, supports +, –, *, /
  - `char` → character values
  - `double` → floating-point numbers
  - `bool` → true/false logic

[Source]

Compiler checks for types before generating machine code

[Source]

Compiler checks for types before generating machine code

C++ is a compiled, statically typed language

[Source]

Dynamic Versus Static Typing

Python (Dynamic)

```
a = 3
b = “test”
def foo(c):
    d=106
    d=”Hello, World!” ✅
```

C++ (Static)

```
int a = 3
string b = “test”
void foo(string c):
    int d=106
    d=”Hello, World!” ❌
```

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

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

Why Static Typing ?

- ✅ **Catches errors early**: at compile time
- ⚡ **Faster execution**: no runtime type checks
- 🔒 **Type safety**: prevents invalid operations
- 🧩 **Easier maintenance**: IDE support, safer refactoring
- 📘 **Self-documenting code**: clear variable & function types
-

💡 *C++ is statically typed → all types are known and checked during compilation.*

Any Questions?

C++ Type System

The C++ Type System

- C++ is a **strongly typed** and **statically typed** language

> Every entity has a type and that type never changes

```
int abc;
```

- **int** is a type. Variable **abc** has type **int**.

Source

The C++ Type System

```
cout << 123 + 456
```

- Literals **123** and **456** also represent values of type int in C++, although this does not need to be explicitly stated.
- Applying binary operator **+** to two values of type **int** results in another value of type int.
- There, the expression **123 + 456** also has type **int**

Source

The C++ Type System

- Every variable, function, or expression has a **type** in order to be compiled.
- User can introduce new types with **class** or **struct**.

Source

The C++ Type System

- The **type** specifies:
  - The **amount of memory** allocated for the variable (or expression result)
  - The **kind of values** that may be stored and how the compiler interprets the bit patterns in those values
  - The **operations** that are permitted for those entities and provides semantics

Source

Type Categories

C++ organizes the language types in two main categories:

- **Fundamental types:** often called ***primitive types/builtin types***. Types provided by the language itself that don't require additional headers
- **Compound types:** Composition or references to other types

Source

Fundamental Types

C++ defines a set of primitive types

- Void type
- Boolean type
- Integer types
- Character types
- Floating point types

All other types are composed of these fundamental types in some way

Source

Void Type (1/2)

The void type has no values

- Identified by the C++ keyword void
- No objects of type void are allowed
- Mainly used as a return type for functions that do not return any value
- Pointers to void are also permitted

Source

Void Type (2/2)

```
void * pointer; // OK: pointer to void
void object; // ERROR: object of type void
void doSomething(){
// do something
}
void doSomething(void); // OK: no params
```

Source

Void Type (2/2)

```c
void * pointer; // OK: pointer to void
void object; // ERROR: object of type void
void doSomething(){
// do something
}
void doSomething(void); // OK: no params
```

In C `sizeof(void) == 1` (GCC), while in C++ `sizeof(void)` does not compile!

Source

Boolean Type

The boolean type can hold two values

- Identified by the C++ keyword `bool`
- Represents the truth values `true` or `false`

```
bool condition = true;
// ..
if(condition){
// …
}
```

Source

Integer Type (1/5)

The integer types represent integral values

- Identified by the C++ keyword `int`
- Some properties of integer types can be changed through modifiers
- `int` keyword may be omitted if at least one modifier is used

Source

Integer Type (2/5)

Singedness modifiers

- Signed integers will have signed representation (i.e. they can represent negative numbers)
- Unsigned integers will have unsigned representation(i.e they can only represent non-negative numbers

Source

Integer Type (3/5)

Size modifiers

- Short integers will be optimized for spacer (at least 16 bits wide)
- Long integers will be at least 32 bits wide
- Long long integers will be at least 64 bits wide

Source

Integer Type (4/5)

Modifiers and int keyword can be specified in any order

```
// a, b, c and d all have the same type
unsigned long long int a;
unsigned long long b;
long unsigned int long c;
long long unsigned d;
```

By default integers are signed, thus the signed keyword can be omitted

Source

Integer Type (5/5)

Overview of the integer types as specified by the C++ standard

Canonical Type Specifier	Minimum Width	Minimum Range
`short` `unsigned short`	16 bit	-2^{15} to $2^{15}-1$ 0 to $2^{16}-1$
`int` `unsigned`	16 bit	-2^{15} to $2^{15}-1$ 0 to $2^{16}-1$
`long` `unsigned long`	32 bit	-2^{31} to $2^{31}-1$ 0 to $2^{32}-1$
`long long` `unsigned long long`	64 bit	-2^{63} to $2^{63}-1$ 0 to $2^{64}-1$

The exact width of integer types is not specified by the standard!

Source

Fixed-Width Integer Types

Sometimes we need integer types with a guaranteed width

- Use fixed-width integer types defined in `<cstdint> header`
- `int8_t, int16_t, int32_t` and `u` for signed integers of with 8, 16, 32, or 64 bit respectively
- `int8_t, uint16_t, uint32_t` and `uint64_t` for unsigned integers of width 8, 16, 32 or 64 bits respectively

Only defined if the C++ implementation directly supports the type

Source

Fixed-Width Integer Types

```
#include<cstdint>
long a; // may be 32 or 64 bits wide
int32_t b; // Guaranteed to be 32 bits wide
int64_t c; // Guaranteed to be 64 bits wide
```

Source

Integer Type Guidelines (1/3)

- Use basic (i.e. non-fixed-width) integer types by default
  - They guarantee a minimum range that can be supported
  - Most of the time we do not need to know an exact maximum value
  - Usually (unsigned) int or long are a reasonable choice

Source

Integer Type Guidelines (2/3)

- Use basic (i.e. non-fixed-width) integer types by default
  - They guarantee a minimum range that can be supported
  - Most of the time we do not need to know an exact maximum value
  - Usually (unsigned) int or long are a reasonable choice

- Only use fixed-width integer types where absolutely required
  - E.g. in data structures that need to have deterministic fixed size

Source

Integer Type Guidelines (3/3)

- Use basic (i.e. non-fixed-width) integer types by default
  - They guarantee a minimum range that can be supported
  - Most of the time we do not need to know an exact maximum value
  - Usually (unsigned) int or long are a reasonable choice

- Only use fixed-width integer types where absolutely required
  - E.g. in data structures that need to have deterministic fixed size

- Do not prematurely optimize for space consumption
  - Registers on modern CPUs are likely to be 64 bit wide anyway
  - Most of the time a program only becomes susceptible to overflow bugs if narrow integer types are used without good reason

Source

Character Types (1/2)

- Character types represent character codes and (to some extent) integral values
- The C++ type char may either be equivalent to signed char or unsigned char, depending on the implementation
- **Minimum width: 8 bits**
- `char` may be **signed or unsigned** (implementation-dependent)

In C++, `char` is always a distinct type; even when it shares representation with `signed char` or `unsigned char`, the type system treats it separately for overload resolution, type checking, and semantics.

Source

Character Types (2/2)

```
char c = 'A';               // character
signed char x = -10;        // small integer
unsigned char y = 200;      // small integer

std::cout << c << std::endl;          // A
std::cout << int(c) << std::endl;     // 65
// Explicit cast tells the compiler to treat char as an
integer

int i = 66;
std::cout << i << std::endl;          // 66
std::cout << char(i) << std::endl;    // B
// Integer interpreted as a character code
```

Floating Point Types (1/2)

Floating point types of varying precision

- float usually represents IEEE-754 32 bit floating point numbers
- double usually represents IEEE-754 64 bit floating point numbers
- long double is a floating point type with extended precision (varying width depending on platform and OS, usually between 64 bit and 128 bit)

Source

Floating Point Types (2/2)

Floating point types of varying precision

- float usually represents IEEE-754 32 bit floating point numbers
- double usually represents IEEE-754 64 bit floating point numbers
- long double is a floating point type with extended precision (varying width depending on platform and OS, usually between 64 bit and 128 bit)

Floating point types may support special values

- Infinity
- Negative zero
- Not-a-number

Source

Implicit Conversions (1/5)

What are Type Conversions?

- Type conversions may happen **automatically**
- Occur when an object of **type A** is used where **type B** is expected
- Exact rules are **complex** (defined by the C++ standard)

Integral to bool

```
int x = 0;
int y = 42;

bool b1 = x;   // false
bool b2 = y;   // true
```

✔️ `0` → `false`
✔️ non-zero → `true`

Implicit Conversions (2/5)

bool to Integral

```
bool t = true;
bool f = false;

int a = t;   // 1
int b = f;   // 0
```

Implicit Conversions (3/5)

Floating-Point to Integral (Truncation)

```
double d = 3.9;
int i = d;   // 3
```

Fractional part is **discarded** (not rounded)

Implicit Conversions (4/5)

If one assigns an out-of-range value to an unsigned integral type of width w, the result is the original value modulo 2^w

```
unsigned char u = 400;
```

An unsigned char in C/C++ typically holds values from 0 to 255

$400 \mod 2^8 = 144$

unsigned char width = **8 bits**

// Stored value
u == 144

Implicit Conversions (5/5)

If one assigns an out-of-range value to an unsigned integral type of width w, the result is the original value modulo 2^w

```
unsigned char u = 400;
```

An unsigned char in C/C++ typically holds values from 0 to 255

$400 \mod 2^8 = 144$

unsigned char width = **8 bits**

// Stored value
u == 144

Quiz

```
uint16_t i = 257;
uint8_t j = i;
if (j) {
/* executed if j is not zero */
}
```

What is the value of j?

https://db.in.tum.de/teaching/ss21/c++praktikum/slides/lecture-02.1.pdf?lang=en