Systems Architecture

1. Introduction to C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

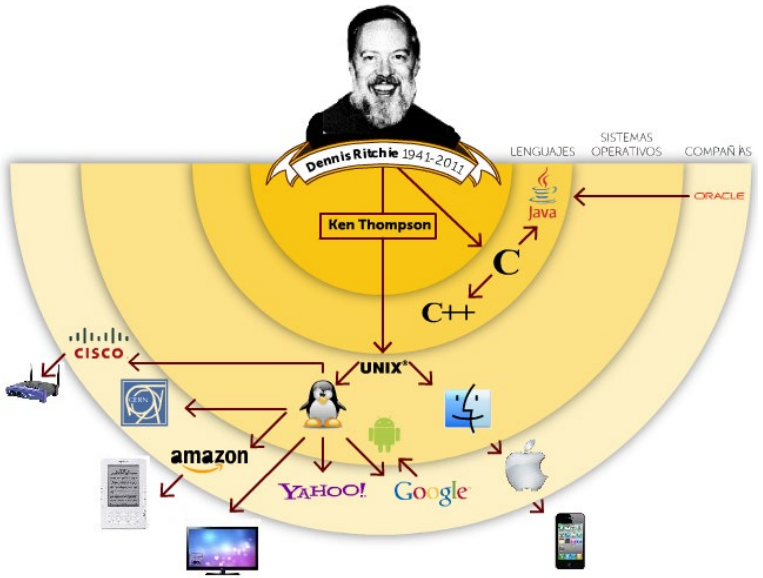
1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

Table of contents

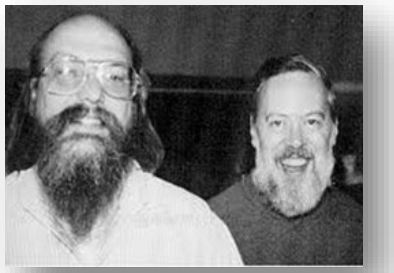
1. Introduction
   - Main features of C
   - General-purpose vs domain-specific
   - Application and system programming
   - Programming language levels
   - Compiled vs. interpreted
   - Programming paradigms
   - Type system
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

1. Introduction

- C is a general purpose programming language developed by **Dennis Ritchie** between 1969 and 1972 at Bell Laboratories
  - Originally C was oriented to the implementation of operating systems, specifically **Unix** (with Ken Thompson)
  - It was first standardized by the ANSI (American National Standards Institute) in 1989
  - It was ratified as an ISO (International Organization for Standardization) standard in 1990



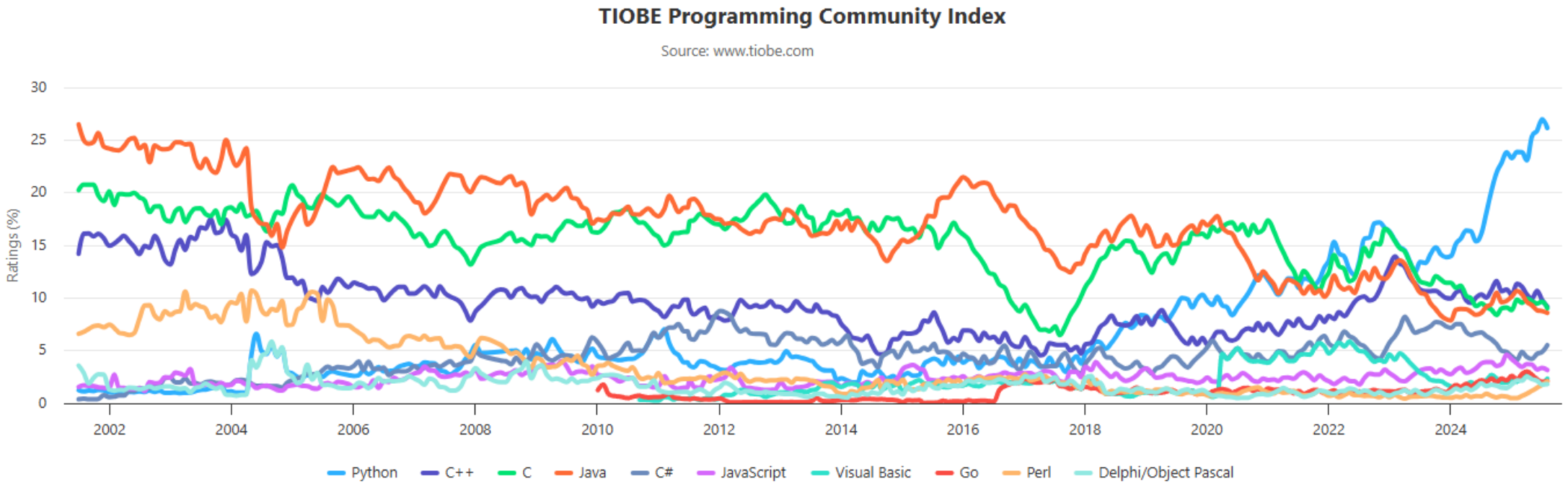
Source: Dennis Ritchie biography by Moisés Cuevas



Source: Wikipedia
https://en.wikipedia.org/wiki/Dennis_Ritchie

1. Introduction

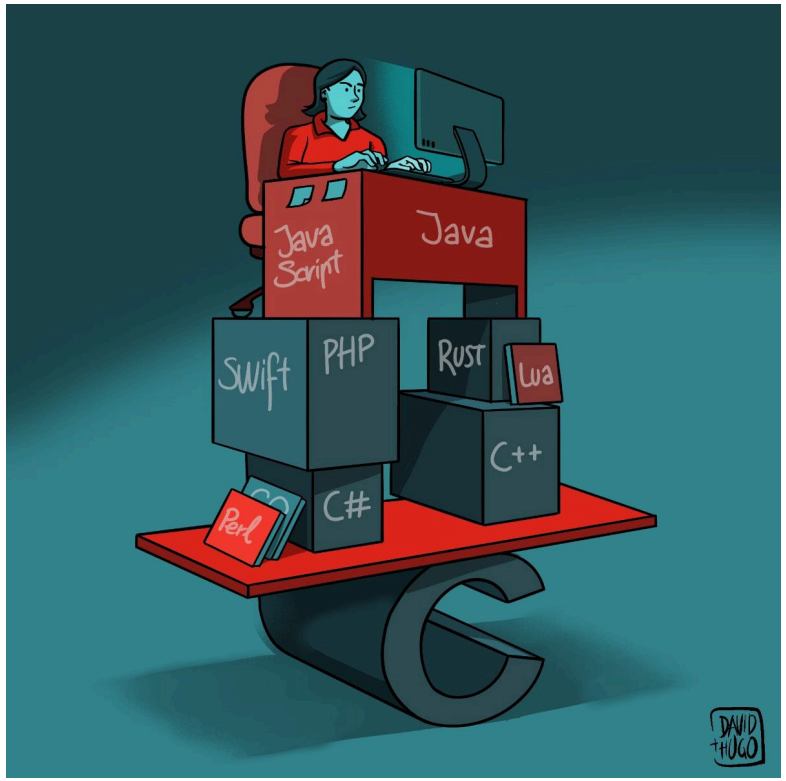
- Today C is still one of the most widely used languages



https://www.tiobe.com/tiobe-index/

1. Introduction

- C is considered the foundation of modern programming languages



© Image created by Hugo Tobio
Source: Bonilista nº 647, 3 September 2023, La Historia del lenguaje C

1. Introduction - Main features of C

- The C programming language can be classified in several ways:
  - C is a **general-purpose** programming language
    - C can be used both **application programming** and **system programming**
  - C is a **high-level** programming language
    - Although it allows certain low level handling (direct memory access)
    - For that reason, it is sometimes classified as a *medium-level* language
  - C is a **compiled** language (the C source code must be converted into machine code to be executed)
  - C is **imperative** (based on statements that are executed sequentially) and **procedural** (it relies on subroutines called *functions* to perform computations)
  - C is **statically-typed** (the type of a variable is known at compile-time) and **weakly-typed** (it allows variables of one type to be used as if they were of another)

1. Introduction - General-purpose vs domain-specific

- A **general-purpose language** (GPL) is a programming language that can create all types of programs
  - For instance: C, Java, Python, among others
  - GPLs are *Turing complete*, which means that they can theoretically solve any computational problem
- A **domain-specific language** (DSL) is a computer language specialized to a particular application domain
  - For instance: MATLAB (intended primarily for numeric computing), SQL (for relational database queries), VHDL (for hardware description)
  - DSLs can be (or not) *Turing complete*

1. Introduction - Application and system programming

- An **applications programming language** is used for implementing user applications, like desktop applications, command-line interface tools, or mobile apps
  - For instance: C, Java, Python, JavaScript, and others
  - Application software generally don't directly access hardware or low-level resources. Instead these languages use *system calls*
- A **system programming language** is used for implementing system software, i.e., software designed to provide a platform for other software, such as operating systems, devices drivers, or server-side components
  - For instance: C, C++, Go, Rust, and others
  - Unlike application software, most system software are not directly used by end users

1. Introduction - Programming language levels

High level program

```java
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

```c
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

It provides abstractions (close to the natural language) independent of a particular type of computer. Programmer friendly, portable, easier to create, debug, and maintain

Low-level program (assembly)

```
org 0x100      ; .com files always start 256 bytes into the segment

mov dx, msg    ; the address of or message in dx
mov ah, 9      ; ah=9 - "print string" sub-function
int 0x21       ; call dos services

mov ah, 0x4c   ; "terminate program" sub-function
int 0x21       ; call dos services

msg db 'Hello, World!', 0x0d, 0x0a, '$'  ; $-terminated message
```

x86

Mnemonics that represent basic instructions that can be directly translated to machine code. Highly memory efficient but non-portable and much more difficult to create, debug, and maintain

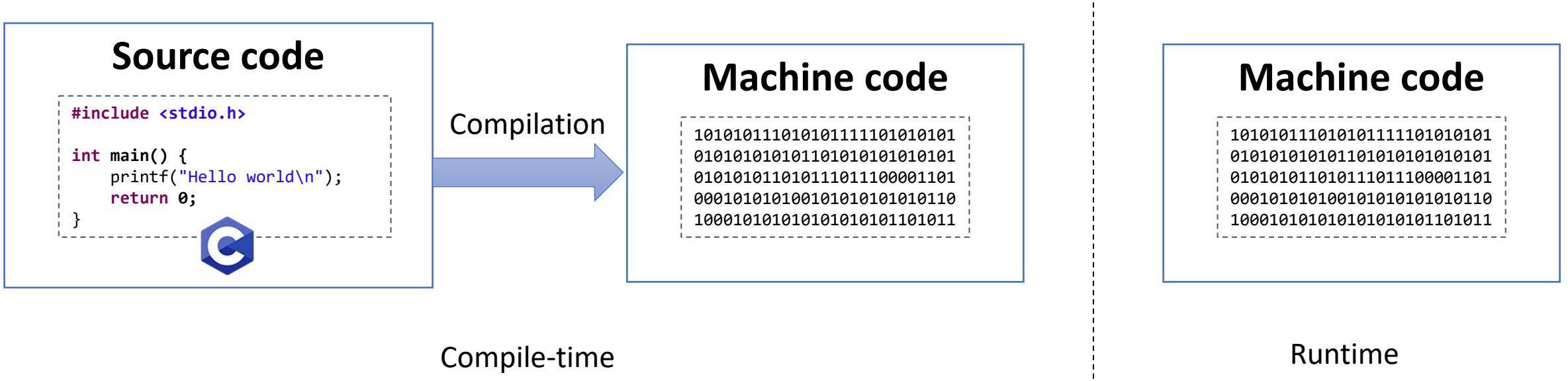
Machine code

```
10101011101010111110101010101010101011010101010101010101010110101
11011100001101000101010100101010101010110100010101010101010110101
```

Executable code (1's and 0's). Each instruction causes the CPU to perform a specific task (e.g. load value, arithmetic operation, etc.)

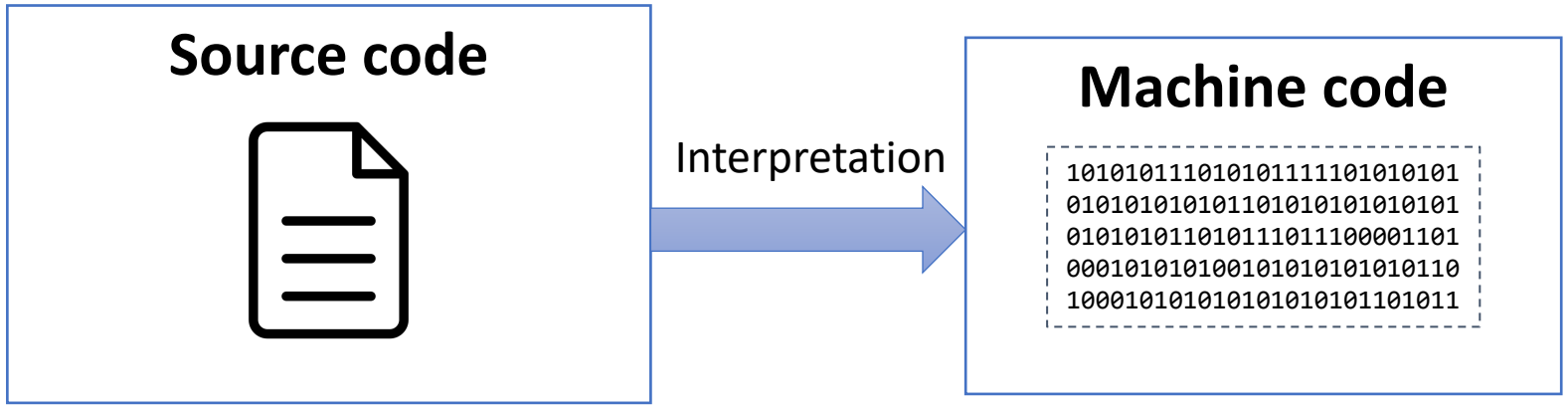
1. Introduction - Compiled vs. interpreted

- **Compiled** languages need a *build* step, i.e., they need to be manually compiled to be executed on a computer
  - This process is sometimes called *ahead-of-time* (AOT) compilation
  - Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go



1. Introduction - Compiled vs. interpreted

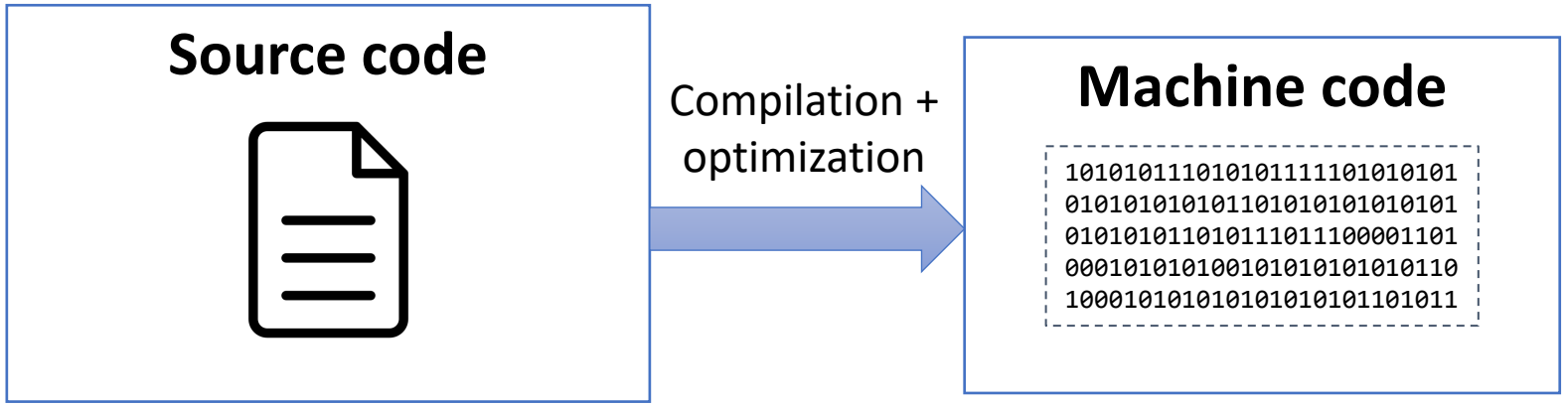
- **Interpreted** languages are executed directly into machine-language instructions without a previous compilation



  - Nevertheless, there are few implementations of pure interpreted languages nowadays, because interpreting source code directly would be quite slow

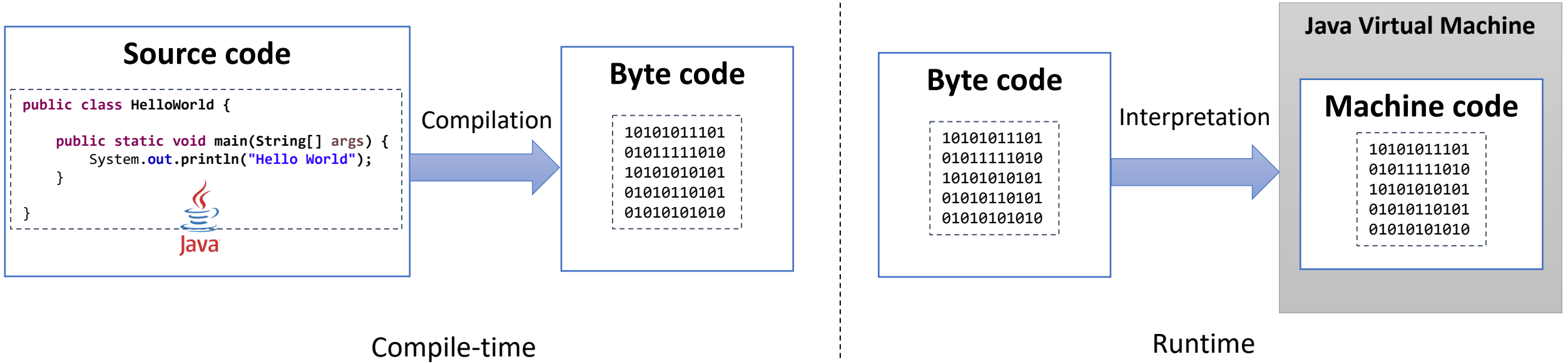
1. Introduction - Compiled vs. interpreted

- Instead, modern interpreted languages use an alternative approach called *just-in-time* (JIT)
  - JIT involves the compilation during execution of a program (at *runtime*) rather than before execution
  - Different optimizations are done in runtime to provide a better performance
  - Examples of interpreted languages using JIT are Python, JavaScript, or Ruby



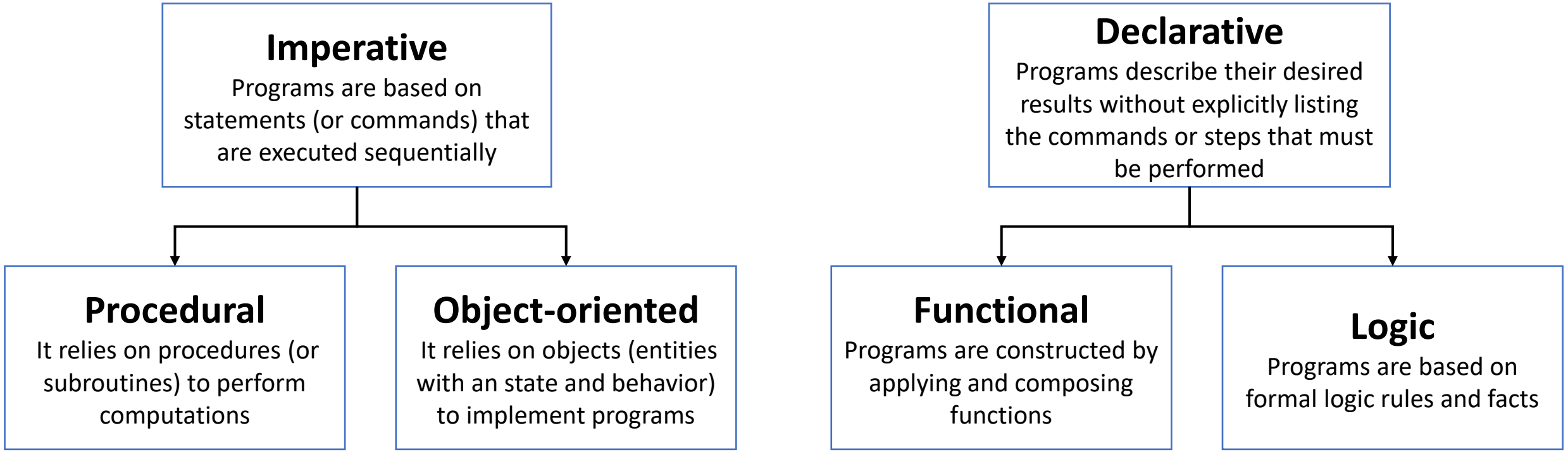
1. Introduction - Compiled vs. interpreted

- Other languages, like Java, use a mixed strategy:
  1. Compile-time: Compilation of the source code into some intermediate format called *bytecode*
  2. Runtime: bytecodes are interpreted (also using JIT) in the Java Virtual Machine (JVM)



1. Introduction - Programming paradigms

- Programming paradigms are a way to classify programming languages based on their features
- The two major programming paradigms nowadays are:



1. Introduction - Type system

- In programming, a **type** is a set of value (e.g. integers, characters, etc.)
- The rules that applies for types in a given programming languages is called *type system*, and it is usually classified as *static* or *dynamic*, and as *strongly-typed* or *weakly-typed*

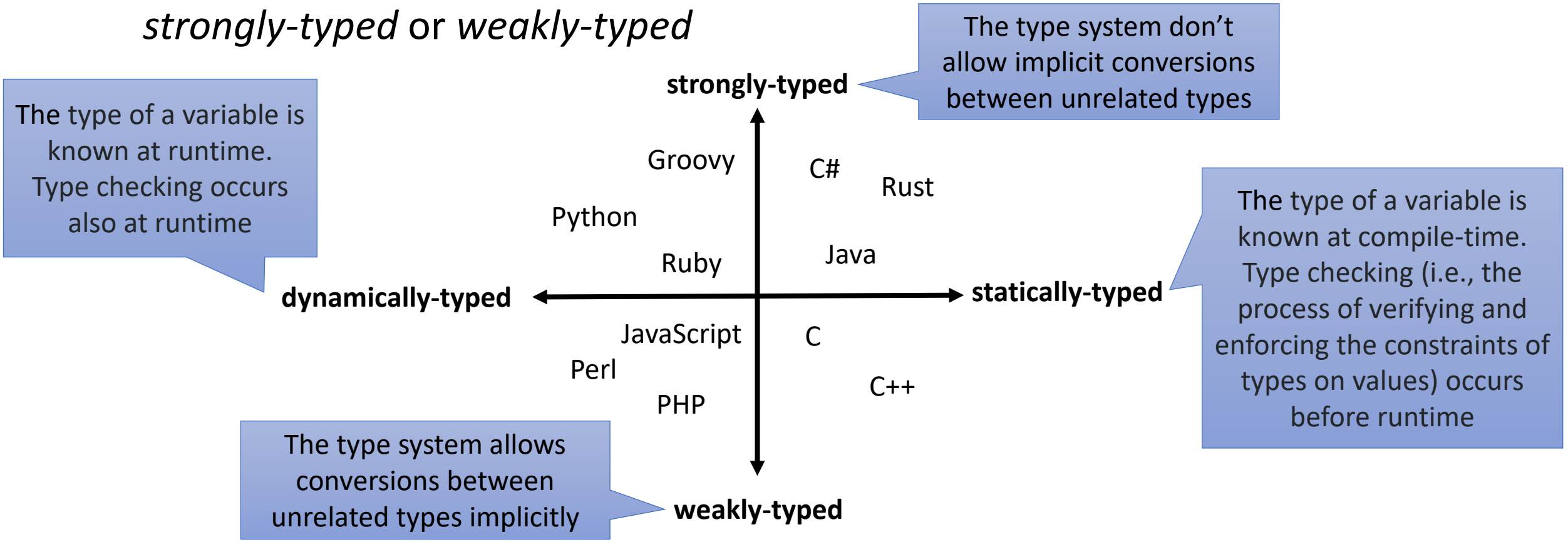


Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

2. “Hello world” in C

Fork me on GitHub

```c
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

hello_world.c

- The **`main()`** function is used by convention as the entry point of the program

https://github.com/bonigarcia/c-programming

Repository with code examples

Fork me on GitHub

2. “Hello world” in C

```c
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

- Statements end with semicolons **;**
- We define blocks of statements in curly brackets (braces) **{ }**
- The `printf()` function displays a text string in the standard output (`stdout`)
- To use that function, we need to include its code from the standard library **`stdio.h`** using the directive **`#include`**
- With **`return`** we terminate the function, returning a value to the calling process
  - The exit code is a numerical value returned by a program to the operating system upon its completion
  - The exit code **0** means *success*. Different than **0** means some error (e.g., **1** means *general error*)

Fork me on GitHub

2. “Hello world” in C

```c
#include <stdio.h>

int main() {
    printf("Hello world\n");
    return 0;
}
```

First, we use the compiler `gcc` (GNU Compiler Collection) from the shell to compile a c program (.c file)

```
gcc hello.c
```

Then, we invoke the binary name from the shell to execute it

```
./a.out
```

By convention, the generated executable is called `a.out` by default. We can explicitly set a different binary name using the flag `-o <name>`

```
gcc hello.c -o hello
```

```
./hello
```

Compile-time

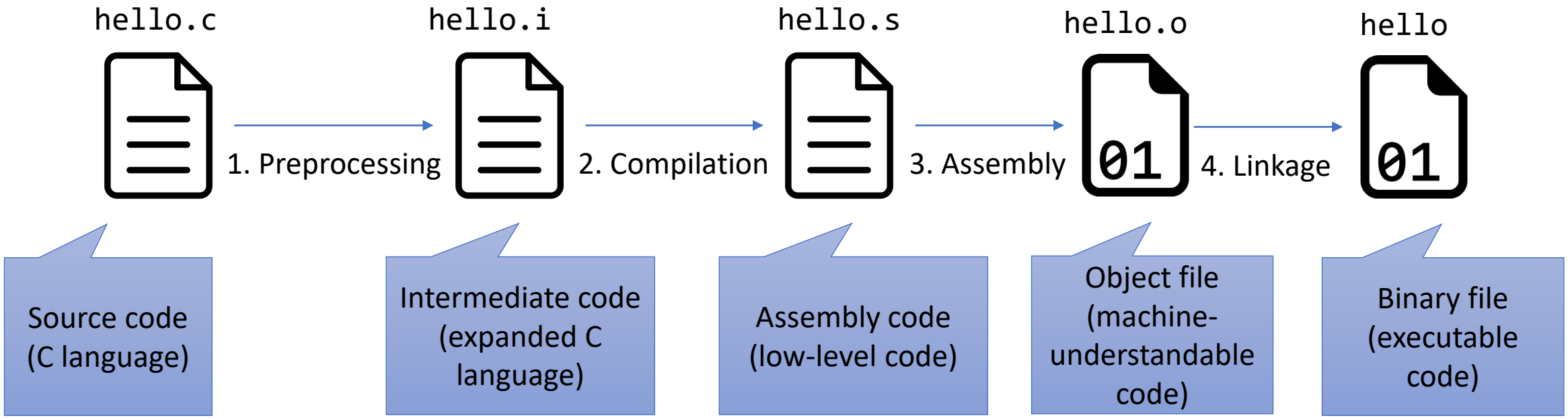
Runtime

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

3. The build process

- The build process in C has actually 4 stages:



3. The build process

1. Preprocessing: The preprocessor converts the source code (.c) into some intermediate file (.i) doing the following:
   - Removing comments
   - Expanding **macros** (expressions defined using the `#define` directive)
   - Expanding included files (include content of files defined using the `#include` directive)
2. Compilation: The compiler converts the intermediate file into an assembly file (.s) which has low-level instructions
3. Assembly: The assembler will convert the assembly code into object code (.o)
   - Object code holds the translated machine code from the original source code. Object code is not yet executable because:
     - It may contain references to functions or variables that are defined in other object files
     - It does not have an entry point required by the operating system to start the program execution
4. Linkage: The linker merges all the object(s) code into a single one (executable)

3. The build process

- Others GCC commands:

```
gcc hello.c -o hello -save-temps
```

Generates binary file but does not delete temporal files (.i, .s, and .o)

```
gcc hello.c -E -o hello.i
```

Only generates intermediate code (.i)

```
gcc hello.c -S -o hello.s
```

Only generates assembly code (.s)

```
gcc hello.c -c -o hello.o
```

Only generates object file (.o)

```
gcc hello.c -o hello -Wall
```

Generates binary file and check all warnings

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
   - Basic types
   - Enumerated types
   - Type definitions
   - Type conversions
5. Variables
6. Constants
7. Code style
8. Takeaways

4. Data types

- A data **type** defines the set of values for a variable
- There are four groups of data types in C:

Basic

Primary data types (integers, characters, etc.)

Enumerated

Integer values associated to labels

Void

No value (for functions without return or generic pointers)

Derived

Arrays, unions, structures, and pointers

4. Data types - Basic types

- The basic types in C are the following:

Data type	Description	Typical memory size (in bytes)	Range	Format specifier
`char`	Characters with sign	1	-128 to 127	`%c`
`unsigned char`	Characters without sign	1	0 to 255	`%c`
`short`	Short integers with sign	2	−32,768 to 32,767	`%hi` or `%hd`
`unsigned short`	Short integers without sign	2	0 to 65,535	`%hu`
`int`	Integers with sign	4	-2,147,483,648 to 2,147,483,647	`%i` or `%d`
`unsigned int`	Integers without sign	4	0 to 4,294,967,295	`%u`
`long`	Long integer with sign	8	-9.2e18 to 9.2e18	`%li` or `%ld`
`unsigned long`	Long integer	8	0 to 1.8E19	`%lu`
`float`	Decimal	4	1.1e-38 to 3.4e38	`%f`
`double`	Decimal with double precision	8	2.2e-308 0 to 1.7e308	`%lf`

4. Data types - Basic types

- A character variable holds **ASCII** value, i.e., an integer number between 0 and 127

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Source: Wikipedia
https://en.wikipedia.org/wiki/ASCII

Fork me on GitHub

4. Data types - Basic types

- The first argument in function `printf` (i.e., a string) can include *format specifiers* (i.e., subsequences beginning with %)
  - In this case, the additional arguments are formatted and inserted in the resulting string replacing their respective specifiers

```c
#include <stdio.h>

int main() {
    char character = 'c';
    int integer = 255;
    float float_num = 1.2;
    double double_num = 3.1e33;

    printf("This is a character: %c\n", character);
    printf("This is an integer: %d\n", integer);
    printf("This is a float: %f\n", float_num);
    printf("This is a double: %g\n", double_num);
    printf("This is an integer in hexadecimal: %X\n", integer);

    return 0;
}
```

```
This is a character: c
This is an integer: 255
This is a float: 1.200000
This is a double: 3.1e+33
This is an integer in hexadecimal: FF
```

Fork me on GitHub

4. Data types - Basic types

- The operator **`sizeof`** generates the storage size (in bytes) of an expression or a data type

```c
#include <stdio.h>

int main() {
    printf("The size of a CHAR is %ld bytes\n", sizeof(char));
    printf("The size of a SHORT is %ld bytes\n", sizeof(short));
    printf("The size of a INT is %ld bytes\n", sizeof(int));
    printf("The size of a LONG is %ld bytes\n", sizeof(long));
    printf("The size of a FLOAT is %ld bytes\n", sizeof(float));
    printf("The size of a DOUBLE is %ld bytes\n", sizeof(double));

    return 0;
}
```

```
The size of a CHAR is 1 bytes
The size of a SHORT is 2 bytes
The size of a INT is 4 bytes
The size of a LONG is 4 bytes
The size of a FLOAT is 4 bytes
The size of a DOUBLE is 8 bytes
```

Fork me on GitHub

4. Data types - Enumerated types

- Enumeration are a user defined data types used to assign names to integral constants
  - These names make a program easy to read and maintain
  - The keyword **enum** is used to declare new enumeration types in C

```c
#include <stdio.h>

int main() {
    enum days {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    };

    enum days today = SATURDAY;
    enum days tomorrow = SUNDAY;

    printf("Today is %d\n", today);
    printf("Tomorrow is %d\n", tomorrow);

    if (today == SATURDAY || today == SUNDAY) {
        printf("It's the weekend!\n");
    }

    return 0;
}
```

```
Today is 5
Tomorrow is 6
It's the weekend!
```

Fork me on GitHub

4. Data types - Type definitions

- The keyword **typedef** in C allows to create an additional name (alias) for another data type

```c
#include <stdio.h>

int main() {
    typedef unsigned char byte;
    byte character = 'd';
    printf("My character is: %c\n", character);

    return 0;
}
```

```
My character is: d
```

Fork me on GitHub

4. Data types - Type conversions

- Converting one type explicitly into another is known as **type casting** (or type-conversion)
- We convert the values from one type to another explicitly using the cast operator as follows:

```
(type) expression
```

```c
#include <stdio.h>

int main() {
    int sum = 17, count = 5;
    double mean;

    mean = (double) sum / count;
    printf("Mean: %f\n", mean);

    return 0;
}
```

```
Mean: 3.400000
```

Fork me on GitHub

4. Data types - Type conversions

- Implicit type conversion is known as **type promotion** (or coercion)

```c
#include <stdio.h>

int main() {
    int i = 1;
    char c = 'A'; // The ASCII value of 'A' is 65
    int sum;

    sum = i + c;
    printf("Sum: %d\n", sum);

    return 0;
}
```

```
Sum: 66
```

Fork me on GitHub

4. Data types - Type conversions

- Implicit type conversion is known as **type promotion** (or coercion)

```c
#include <stdio.h>

int main() {
    int i = 65;
    char c = i; // 65 is the ASCII value of 'A'

    printf("%d as a character is %c\n", i, c);

    return 0;
}
```

```
65 as a character is A
```

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
   - Scope
   - Shadowing
6. Constants
7. Code style
8. Takeaways

Fork me on GitHub

5. Variables

- **Variables** are containers for storing data values
- We distinguish three ways of handling variables:
  1. **Declaration**: statement to specify the variable name and its data type
  2. **Assignment**: set a value to the variable using the operator =
  3. **Initialization**: initial assignment during declaration

```c
#include <stdio.h>

int main() {
    int a; // declaration
    a = 10; // assignment

    int b = 10; // initialization

    return 0;
}
```

Fork me on GitHub

5. Variables - Scope

- The **scope** is the range within a program for which an item (e.g., a variable) is valid (beyond that, it cannot be accessed)
- There are two types of variables depending on its scope:
  - Global variables: defined outside a function, usually on top of the program
  - Local variables: defined inside a function or block. They can be used only by statements that are inside that function or block of code

```c
#include <stdio.h>

/*
 These are global variables
 (multi-line comment)
 */
int a = 1;
int b;

int main() {
    int c, d = 2; // local variables
    char e;

    e = 'z'; // assignments
    b = 7;
    c = 5;

    printf("a=%d b=%d c=%d d=%d e=%c\n", a, b, c, d, e);

    return 0;
}
```

```
a=1 b=7 c=5 d=2 e=z
```

Fork me on GitHub

5. Variables - Scope

- We need to be careful with the variable scope. For example:

```c
#include <stdio.h>

int main() {
    int a = 1;

    if (a > 0) {
        int b = 2;

        printf("a=%d and b=%d\n", a, b);
    }

    printf("a=%d and b=%d\n", a, b);

    return 0;
}
```

What happens here?

Fork me on GitHub

5. Variables - Shadowing

- *Shadowing* appears when a variable is defined in a scope with the same name of another one valid in a higher level scope

```c
#include <stdio.h>

int b = 0;

int main() {
    int a = 1;

    if (a > 0) {
        int b = 2;

        printf("a=%d and b=%d\n", a, b);
    }

    printf("a=%d and b=%d\n", a, b);

    return 0;
}
```

What happens here?

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

Fork me on GitHub

6. Constants

- Constants refer to fixed values that the program may not alter during its execution. There are different ways to define constants in C:
  - Using a macro defined with the preprocessor directive **#define**
  - Using the keyword **const**
  - Using the enumerated types

```c
#include <stdio.h>

#define MAX 64

int main() {
    const int num = 15;
    enum parity {
        ODD = 1, EVEN = 2
    };

    printf("MAX=%d\n", MAX);
    printf("num=%d\n", num);
    printf("EVEN=%d ODD=%d\n", EVEN, ODD);

    return 0;
}
```

```
MAX=64
num=15
EVEN=2 ODD=1
```

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

7. Code style

- Some usual guidelines in C:
  - Use meaningful names for variables, constants and functions
  - Use *snake-case* (underscores for multi-word names, e.g. `file_name`) and not *camel-case* (use of capital letter except the initial word, e.g. `fileName`)
  - Use the same indentation level (3 or 4 spaces, or 1 tab)
  - Use the same style for opening braces (in the same line or just above)
  - Define constants macros in uppercase
- Best practices:
  - Avoid code duplication (DRY - Don't Repeat Yourself)
  - Group related functions in a separate module
  - Use an automated code formatter, for example:

Windows and Linux: *Ctrl + Shift + F*
macOS: *Command + Shift + F*

Windows: *Shift + Alt + F*
Linux: *Ctrl + Shift + I*
macOS: *Shift + Option + F*

Table of contents

1. Introduction
2. “Hello world” in C
3. The build process
4. Data types
5. Variables
6. Constants
7. Code style
8. Takeaways

8. Takeaways

- C is a programming language: general-purpose, high-level, compiled, imperative and procedural, statically and weakly-typed
- C programs are organized using procedures called **functions**. At least, the function **`main()`** should be defined (i.e., the program entry point)
- The build process in C is done with GCC and has 4 steps: preprocessing, compilation, assembly, and linkage
- There are four groups of data types in C: basic (**`int`**, **`char`**, etc.), enumerated (**`enum`**), void (**`void`**), and derived (arrays, unions, structures, and pointers)
- Depending its scope, there are two types of variable in C: global (defined outside a function) and local (defined inside a function or block)