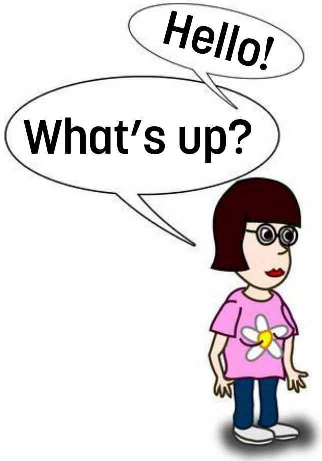
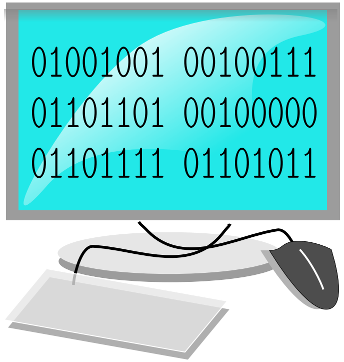
# Computer language:
# Source code, assembly, binary code

COMPACT
OSADL
ONLINE
LECTURES

# Communicating with a computer



For humans to be able to communicate with a computer (more easily), **computer languages** are used.



Humans communicate mostly via language.

A computer "understands" only binary code (01) representing physical states of hardware ("machine code").

# Computer languages

- A Computer language is a **set of operations and operators to instruct a computer** what to do.
- The text that is written by a programmer in a computer language is called **"source code"**.
- Source code must be translated into machine code for a computer to understand it.

# Assembly vs. high-level languages

- Different computer architectures (e.g. Intel, ARM, PowerPC or RISC-V) require instructions in different machine code.
- **Assembly languages** are computer languages for a single architecture → can be directly translated into machine code.
- Computer languages that can be used on any architecture are called **high-level languages** → must first be translated into an assembly language.

# High-level computer languages

- High-level computer languages are classified into **interpreter and compiler** languages on the one hand, and into **general and problem-oriented** languages on the other hand.

## Example computer languages

| | Interpreter languages | Compiler languages |
|---|---|---|
| **General purpose** | PHP, Javascript, Python | C/C++, C#, Rust |
| **Problem oriented** | APL, R | Fortran, Cobol |

# Interpreter languages

- An **interpreter** executes interpreter language source code line by line **when running** a program.
- Binary code is not saved but immediately passed on to the processor.
- Many interpreters are able to translate the code into an **intermediate language** ("I-code") that needs less space and can be executed more quickly (e.g. minified Javascript).

# Interpreter languages

- Interpreters normally provide a text interface for interactive programming:

## For example Python

Interactive:

```
$ python
>>> print("Hello")
Hello
>>> a = 2
>>> b = 3
>>> print(a + b)
5
```

Static:

Source code (hello.py):

```
#!/usr/bin/env python
print("Hello")
a = 2
b = 3
print(a + b)
```

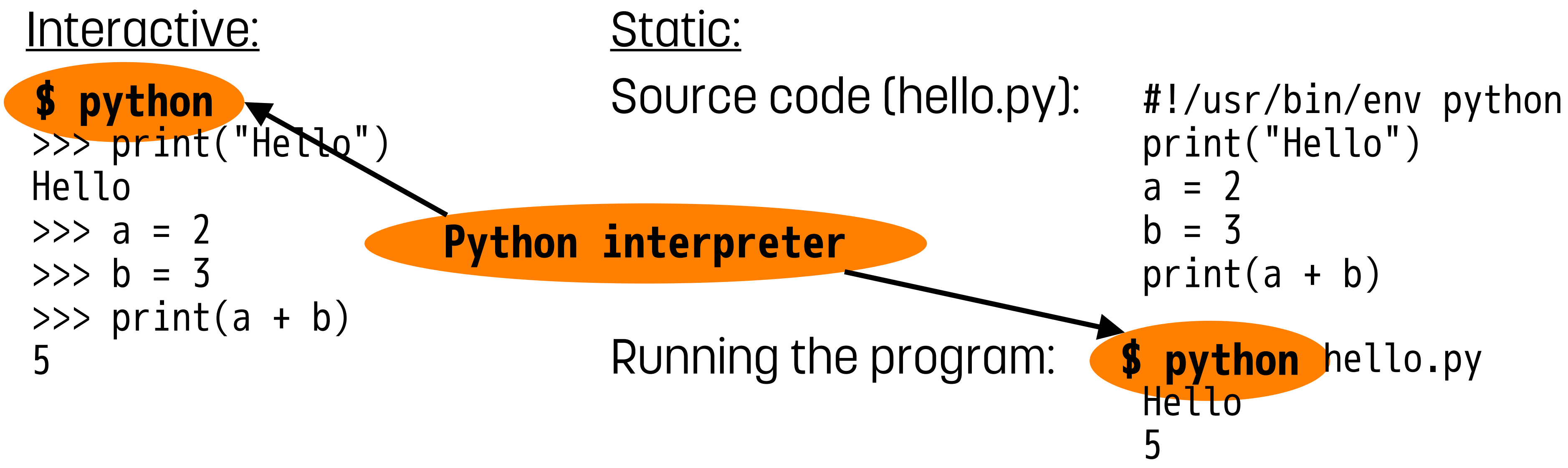
Running the program:

```
$ python hello.py
Hello
5
```

# Interpreter languages

- Interpreters normally provide a text interface for interactive programming:

## For example Python



Interactive:

```
$ python
>>> print("Hello")
Hello
>>> a = 2
>>> b = 3
>>> print(a + b)
5
```

Python interpreter

Static:

Source code (hello.py):

```
#!/usr/bin/env python
print("Hello")
a = 2
b = 3
print(a + b)
```

Running the program:

```
$ python hello.py
Hello
5
```

# Compiler languages

- A **compiler** translates ("compiles") compiler language source code into a **binary executable** that is **stored** before it can be run on a computer.
- Usually, the hardware-independent source code is first compiled into hardware-dependent code in assembly language which is, in a subsequent step, converted into binary machine code.

# Flow of operations from source to machine code

A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

# Flow of operations from source to machine code

A programmer writes code in a high-level language, i.e. C

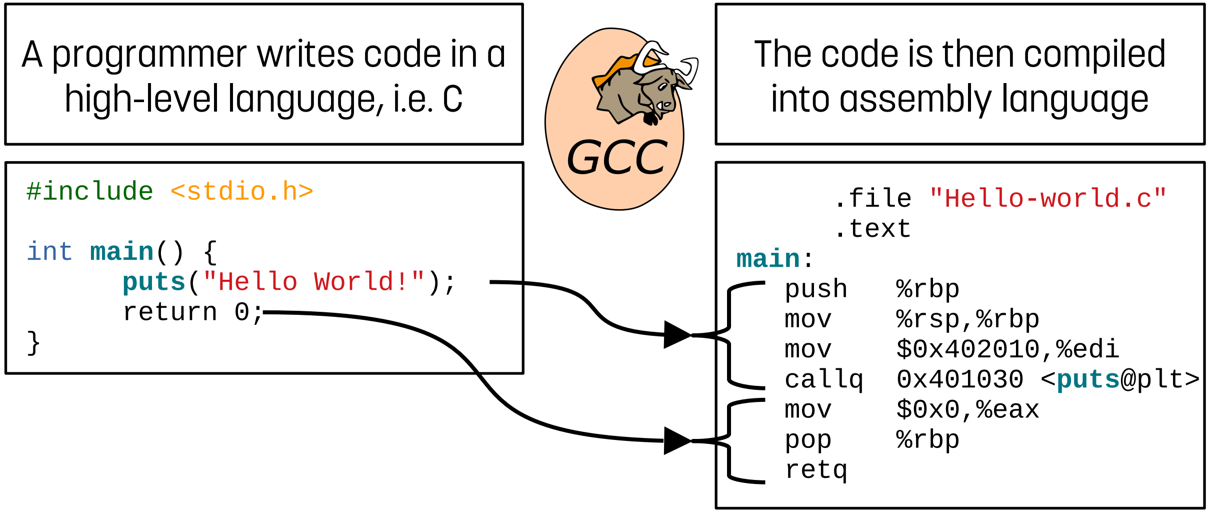
```
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

Compiler

The code is then compiled into assembly language

```
      .file "Hello-world.c"
      .text
main:
   push   %rbp
   mov    %rsp,%rbp
   mov    $0x402010,%edi
   callq  0x401030 <puts@plt>
   mov    $0x0,%eax
   pop    %rbp
   retq
```

# Flow of operations from source to machine code



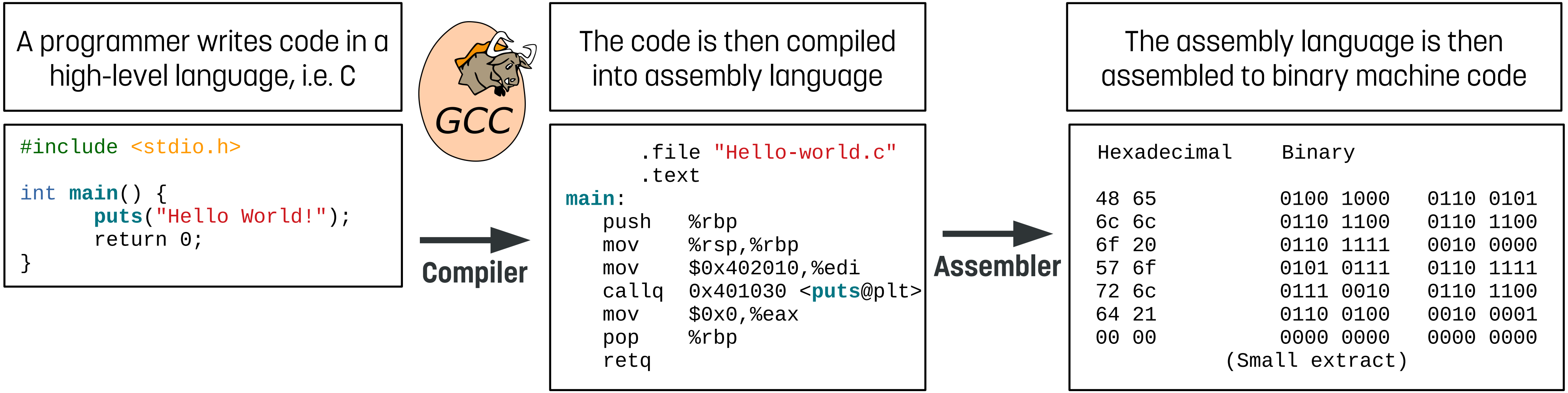
A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

The code is then compiled into assembly language

```
       .file "Hello-world.c"
       .text
main:
   push    %rbp
   mov     %rsp,%rbp
   mov     $0x402010,%edi
   callq   0x401030 <puts@plt>
   mov     $0x0,%eax
   pop     %rbp
   retq
```

# Flow of operations from source to machine code



A programmer writes code in a high-level language, i.e. C

```c
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

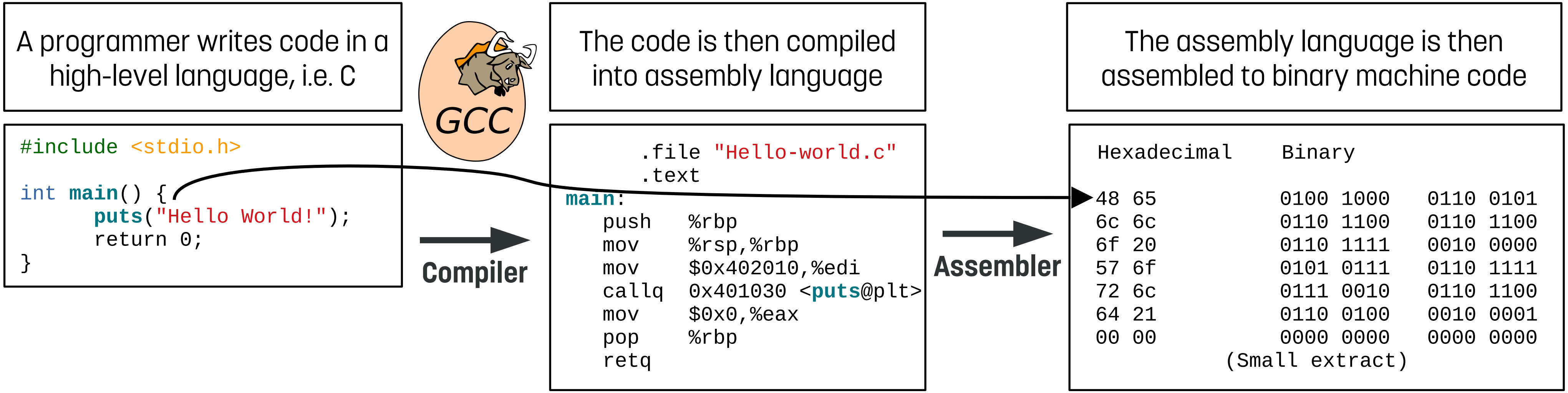
The code is then compiled into assembly language

```
      .file "Hello-world.c"
      .text
main:
   push   %rbp
   mov    %rsp,%rbp
   mov    $0x402010,%edi
   callq  0x401030 <puts@plt>
   mov    $0x0,%eax
   pop    %rbp
   retq
```

The assembly language is then assembled to binary machine code

```
 Hexadecimal    Binary

 48 65          0100 1000   0110 0101
 6c 6c          0110 1100   0110 1100
 6f 20          0110 1111   0010 0000
 57 6f          0101 0111   0110 1111
 72 6c          0111 0010   0110 1100
 64 21          0110 0100   0010 0001
 00 00          0000 0000   0000 0000
           (Small extract)
```

# Flow of operations from source to machine code



A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

The code is then compiled into assembly language

```
       .file "Hello-world.c"
       .text
main:
   push   %rbp
   mov    %rsp,%rbp
   mov    $0x402010,%edi
   callq  0x401030 <puts@plt>
   mov    $0x0,%eax
   pop    %rbp
   retq
```

The assembly language is then assembled to binary machine code

```
 Hexadecimal    Binary

48 65          0100 1000   0110 0101
6c 6c          0110 1100   0110 1100
6f 20          0110 1111   0010 0000
57 6f          0101 0111   0110 1111
72 6c          0111 0010   0110 1100
64 21          0110 0100   0010 0001
00 00          0000 0000   0000 0000
          (Small extract)
```

# Flow of operations from source to machine code

A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
      puts("Hello World!");
      return 0;
}
```

GCC

Compiler

The code is then compiled into assembly language

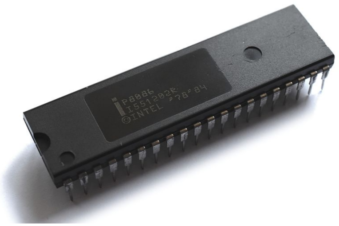
```
       .file "Hello-world.c"
       .text
main:
   push    %rbp
   mov     %rsp,%rbp
   mov     $0x402010,%edi
   callq   0x401030 <puts@plt>
   mov     $0x0,%eax
   pop     %rbp
   retq
```

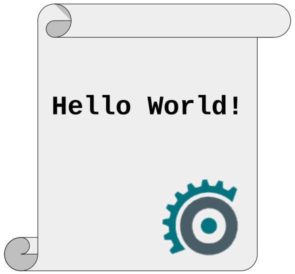
Assembler

The assembly language is then assembled to binary machine code **and executed on a processor**

```
 Hexadecimal    Binary

 48 65          0100 1000   0110 0101
 6c 6c          0110 1100   0110 1100
 6f 20          0110 1111   0010 0000
 57 6f          0101 0111   0110 1111
 72 6c          0111 0010   0110 1100
 64 21          0110 0100   0010 0001
 00 00          0000 0000   0000 0000
           (Small extract)
```

**Software development in a tool chain**





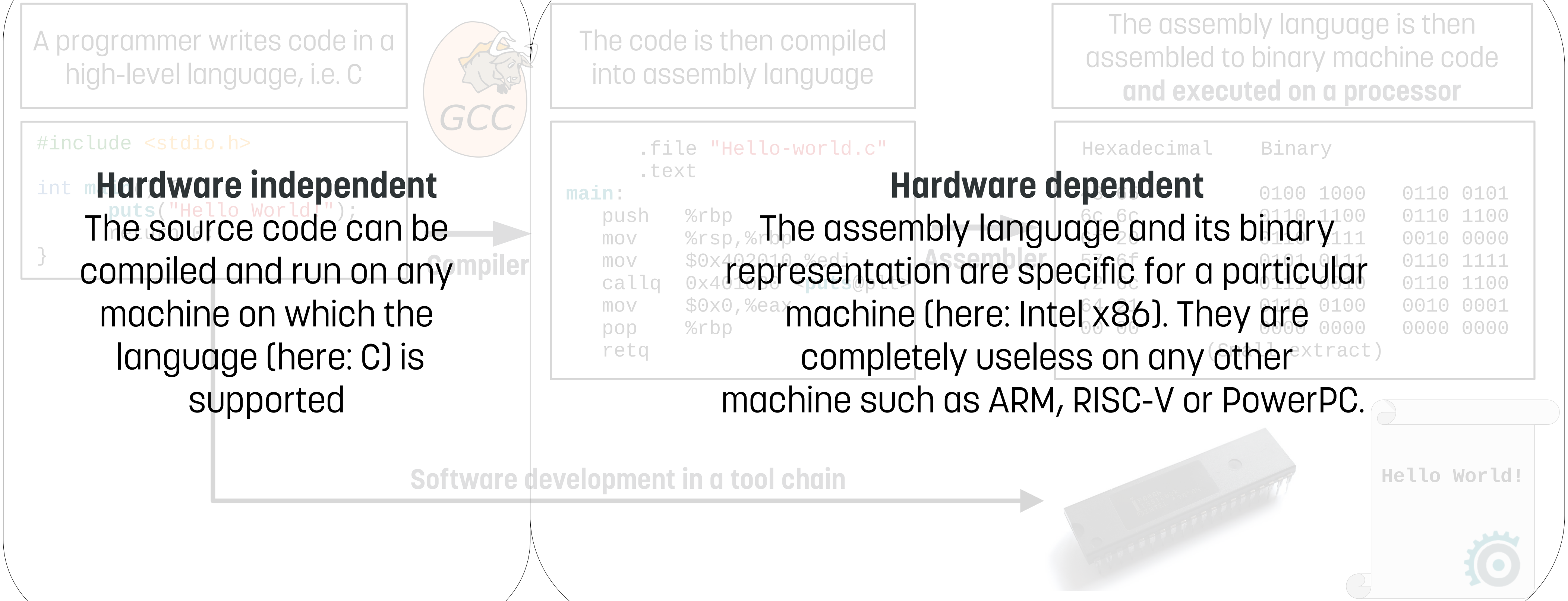
# Flow of operations from source to machine code

**Hardware independent**

The source code can be compiled and run on any machine on which the language (here: C) is supported

**Hardware dependent**

The assembly language and its binary representation are specific for a particular machine (here: Intel x86). They are completely useless on any other machine such as ARM, RISC-V or PowerPC.



A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
    puts("Hello World!");
    return 0;
}
```

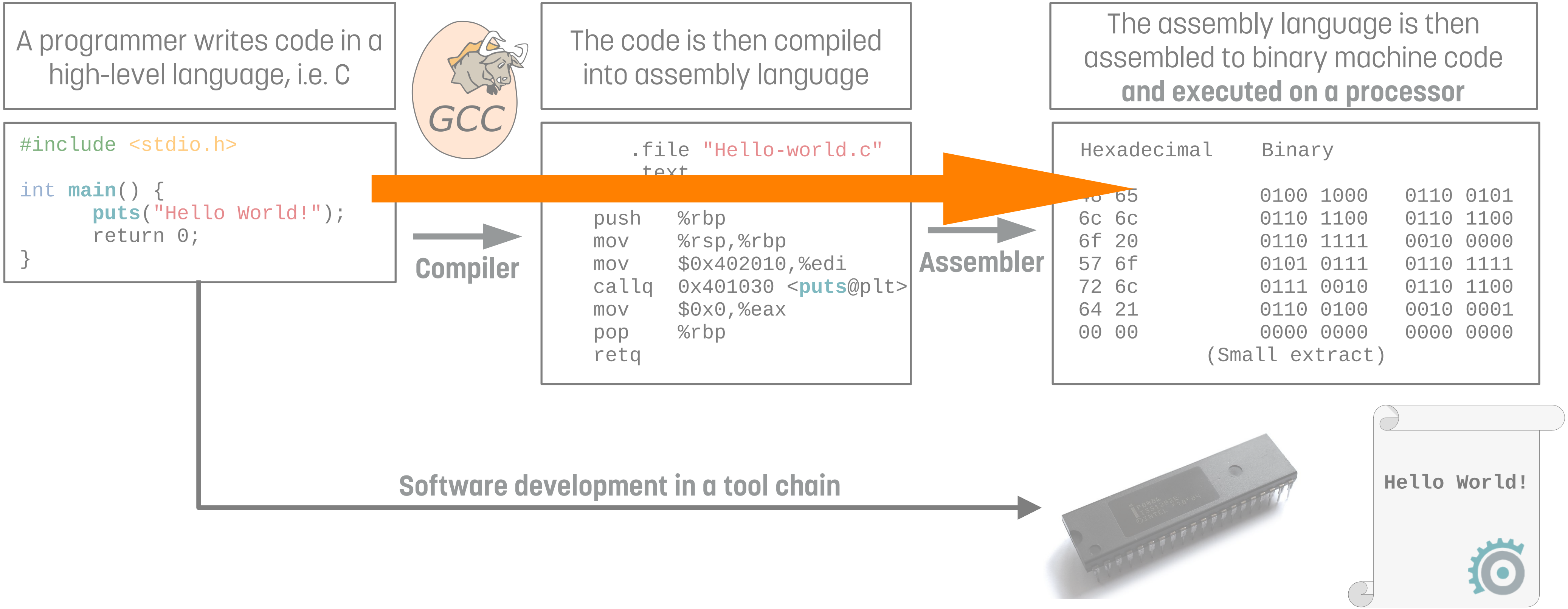
The code is then compiled into assembly language

```
        .file "Hello-world.c"
        .text
main:
    push    %rbp
    mov     %rsp,%rbp
    mov     $0x402010,%edi
    callq   0x401030 <puts@plt>
    mov     $0x0,%eax
    pop     %rbp
    retq
```

The assembly language is then assembled to binary machine code **and executed on a processor**

```
Hexadecimal     Binary
48 65           0100 1000    0110 0101
6c 6c           0110 1100    0110 1100
6f 20           0110 1111    0010 0000
57 6f           0101 0111    0110 1111
72 6c           0111 0010    0110 1100
64 21           0110 0100    0010 0001
00 00           0000 0000    0000 0000
           (Small extract)
```

# Flow of operations from source to machine code



A programmer writes code in a high-level language, i.e. C

```
#include <stdio.h>

int main() {
        puts("Hello World!");
        return 0;
}
```

The code is then compiled into assembly language

```
        .file "Hello-world.c"
         .text

    push    %rbp
    mov     %rsp,%rbp
    mov     $0x402010,%edi
    callq   0x401030 <puts@plt>
    mov     $0x0,%eax
    pop     %rbp
    retq
```

The assembly language is then assembled to binary machine code **and executed on a processor**

```
 Hexadecimal     Binary

 48 65           0100 1000   0110 0101
 6c 6c           0110 1100   0110 1100
 6f 20           0110 1111   0010 0000
 57 6f           0101 0111   0110 1111
 72 6c           0111 0010   0110 1100
 64 21           0110 0100   0010 0001
 00 00           0000 0000   0000 0000
           (Small extract)
```

# Flow of operations from machine to source code

The engineer tries his or her best to understand the code

```
#include <stdio.h>

int a001() {
        puts("Hello World!");
        return 0;
}
```

De

compiler

The source code is estimated from assembly (this is heuristic)

```
      .file "file.c"
       .text

    push    %rbp
    mov     %rsp,%rbp
    mov     $0x402010,%edi
    callq   0x401030 <puts@plt>
    mov     $0x0,%eax
    pop     %rbp
    retq
```

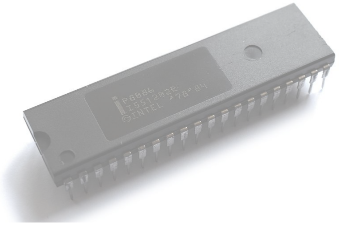
Dis

assembler

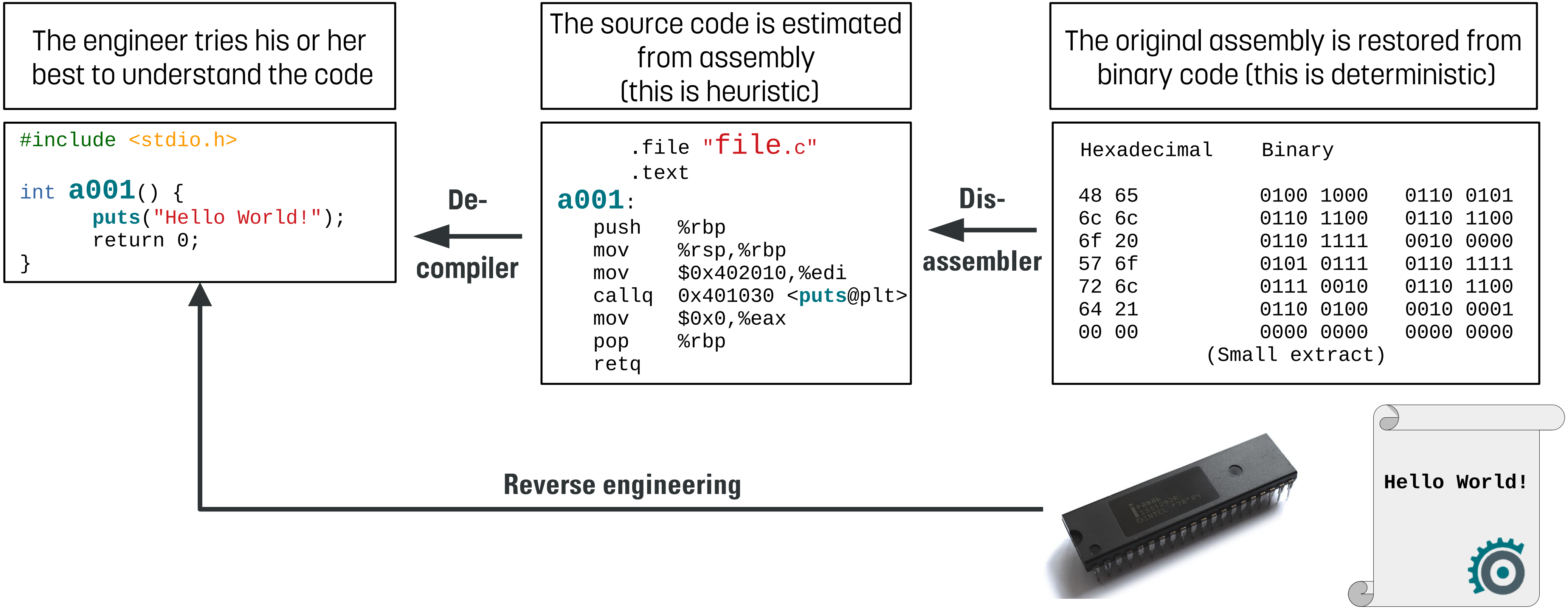
The original assembly is restored from binary code (this is deterministic)

```
 Hexadecimal     Binary

     5           0100 1000   0110 0101
6c 6c            0110 1100   0110 1100
6f 20            0110 1111   0010 0000
57 6f            0101 0111   0110 1111
72 6c            0111 0010   0110 1100
64 21            0110 0100   0010 0001
00 00            0000 0000   0000 0000
          (Small extract)
```

Reverse engineering



# Flow of operations from machine to source code



The engineer tries his or her best to understand the code

The source code is estimated from assembly (this is heuristic)

The original assembly is restored from binary code (this is deterministic)

```c
#include <stdio.h>

int a001() {
        puts("Hello World!");
        return 0;
}
```

De-compiler

```
       .file "file.c"
       .text
a001:
   push    %rbp
   mov     %rsp,%rbp
   mov     $0x402010,%edi
   callq   0x401030 <puts@plt>
   mov     $0x0,%eax
   pop     %rbp
   retq
```

Dis-assembler

```
 Hexadecimal    Binary

 48 65          0100 1000    0110 0101
 6c 6c          0110 1100    0110 1100
 6f 20          0110 1111    0010 0000
 57 6f          0101 0111    0110 1111
 72 6c          0111 0010    0110 1100
 64 21          0110 0100    0010 0001
 00 00          0000 0000    0000 0000
           (Small extract)
```

Reverse engineering

# Flow of operations from machine to source code

The engineer tries his or her best to understand the code

```
#include <stdio.h>

int a001() {
      puts("Hello World!");
      return 0;
}
```

De-compiler

The source code is estimated from assembly (this is heuristic)

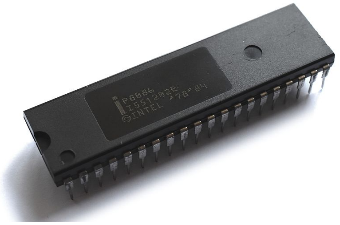
```
        .file "file.c"
        .text
a001:
    push    %rbp
    mov     %rsp,%rbp
    mov     $0x402010,%edi
    callq   0x401030 <puts@plt>
    mov     $0x0,%eax
    pop     %rbp
    retq
```

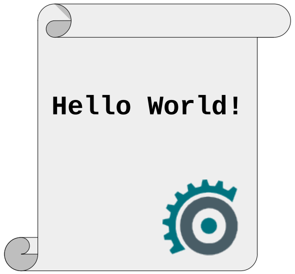
Dis-assembler

The original assembly is restored from binary code (this is deterministic)

```
 Hexadecimal    Binary

 48 65          0100 1000   0110 0101
 6c 6c          0110 1100   0110 1100
 6f 20          0110 1111   0010 0000
 57 6f          0101 0111   0110 1111
 72 6c          0111 0010   0110 1100
 64 21          0110 0100   0010 0001
 00 00          0000 0000   0000 0000
              (Small extract)
```

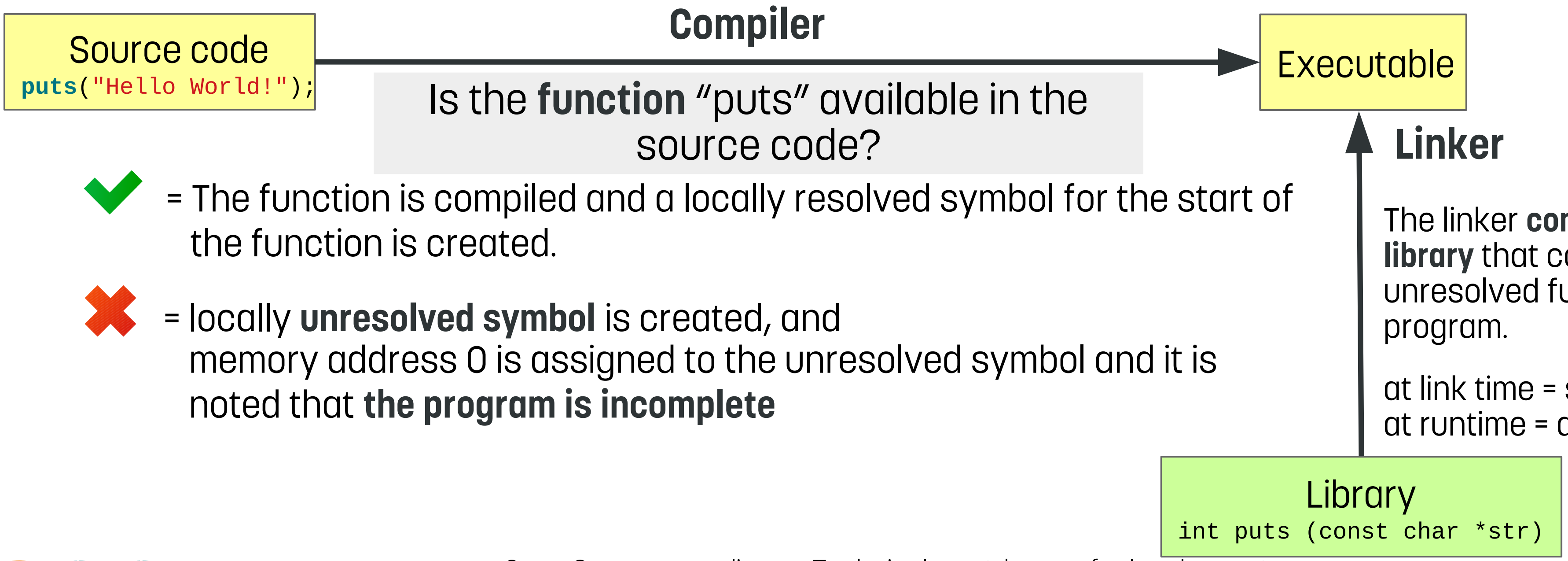
**Reverse engineering**





# Dependencies

- Most programs are not self-contained but require external **dependencies**, i.e. libraries:



Source code

`puts("Hello World!");`

**Compiler**

Executable

Is the **function** “puts” available in the source code?

✓ = The function is compiled and a locally resolved symbol for the start of the function is created.

✗ = locally **unresolved symbol** is created, and memory address 0 is assigned to the unresolved symbol and it is noted that **the program is incomplete**

**Linker**

The linker **combines the library** that contains the unresolved function with the program.

at link time = static linking
at runtime = dynamic linking

Library

`int puts (const char *str)`

# Package managers

- Package managers keep track of (often) complex **dependency constructs** (e.g. correct versions).
- Package managers also store **meta-information** on their packages, including on licensing. However, this information is usually maintained manually and can be **incomplete or outdated**.
- Various systems / programming languages use different package managers.

# Package managers: An excerpt

- Linux distributions:
  - Debian, Ubuntu: dpkg, apt
  - RedHat, Fedora: rpm, dnf
- C / C++:
  - Conan
- Java:
  - Maven
- Rust:
  - Cargo
- Javascript:
  - NPM
- Python:
  - Pip
- PHP:
  - Composer
- Partly **NOT compatible** with each other