

# **PROGRAMMAZIONE PROCEDURALE**

**A.A. 2023/2024**



# PROGRAMMING LANGUAGES

- ④ A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output.
- ④ Programming languages generally consist of instructions for a computer.
- ④ Programming languages can be used to create programs that implement specific algorithms.
- ④ High level programming languages are considered high-level because they are closer to human languages and further from machine languages.

# C IS IMPERATIVE

- ④ Programming paradigms are a way to classify programming languages based on their features
- ④ C is imperative
  - ✓ The programmer instructs the machine how to change its state

```
a = a + 1;
```

# ASSEMBLY LANGUAGE

- ② As people began to write larger programs, it quickly became apparent that a less error-prone notation was required.
- ② Assembly languages were invented to allow operations to be expressed with mnemonic abbreviations. Our GCD program looks like this in x86 assembly language:

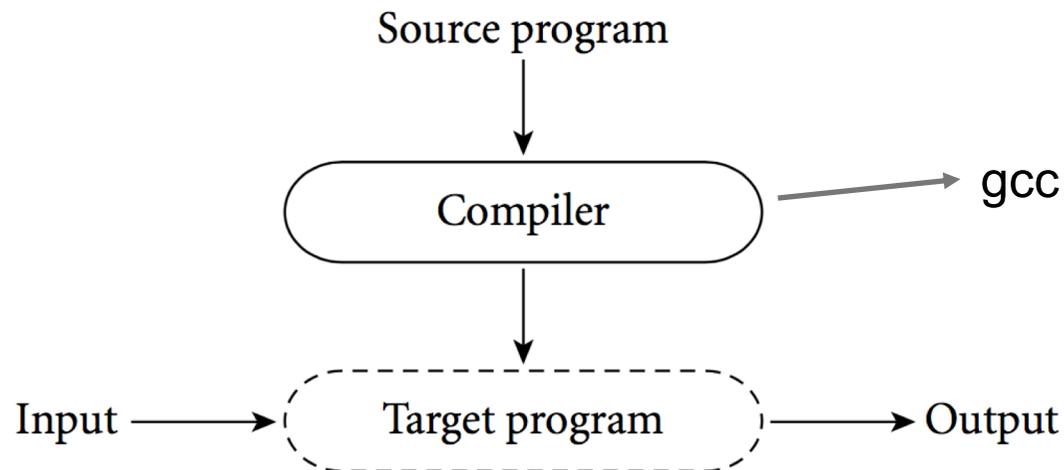
```
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    subl     $4, %esp
    andl     $-16, %esp
    call     getint
    movl     %eax, %ebx
    call     getint
    cmpl     %eax, %ebx
    je       C
A:    cmpl     %eax, %ebx
        jle     D
        subl     %eax, %ebx
        cmpl     %eax, %ebx
        jne     A
C:    movl     %ebx, (%esp)
    call     putint
    movl     -4(%ebp), %ebx
    leave
    ret
D:    subl     %ebx, %eax
    jmp     B
```

# TO HIGH-LEVEL LANGUAGES

- ④ As computers evolved, and as competing designs developed, it became increasingly frustrating to have to rewrite programs for every new machine.
- ④ It also became increasingly difficult for human beings to keep track of the wealth of detail in large assembly language programs.
- ④ People began to wish for a machine-independent language.
- ④ These wishes led in the mid-1950s to the development of the original dialect of **Fortran**, the first arguably high-level programming language.
- ④ Then **Lisp** and **Algol**.

# COMPILER

- ② Translating from a high-level language to assembly or machine language is the job of a systems program known as a *compiler*.
- ② The compiler *translates* the high-level source program into an equivalent target program (an assembly/machine language)
- ② At some arbitrary later time, the user tells the operating system to run the target program.



# A BIT OF C HISTORY



# LET'S START WITH C

- ④ C is a high-level general-purpose, procedural programming language. Dennis Ritchie first devised C in the 1970s at AT&T Bell Laboratories in Murray Hill, New Jersey, for the purpose of implementing the Unix operating system and utilities with the greatest possible degree of independence from specific hardware platforms. The key characteristics of the C language are the qualities that made it suitable for that purpose:
  - ✓ Source code portability
  - ✓ The ability to operate “close to the machine”
  - ✓ Efficiency
- ④ As a result, the developers of Unix were able to write most of the operating system in C, leaving only a minimum of system-specific hardware manipulation to be coded in assembler.

# C

- ④ C's ancestors are the typeless programming languages BCPL (the Basic Combined Programming Language), developed by Martin Richards; and B, a descendant of BCPL, developed by Ken Thompson.
- ④ A new feature of C was its variety of data types: characters, numeric types, arrays, structures, and so on.
- ④ Brian Kernighan and Dennis Ritchie published an official description of the C programming language in 1978.
- ④ Few hardware-dependent elements. For example, the C language proper has no file access or dynamic memory management statements. No input/output.
- ④ Instead, the extensive standard C library provides the functions for all of these purposes.

# VIRTUES OF C

- ④ Fast (it's a compiled language and so is close to the machine hardware)
- ④ Portable (you can compile your program to run on just about any hardware platform out there)
- ④ The language is small (unlike C++ for example)
- ④ Mature (a long history and lots of resources and experience available)
- ④ There are many tools for making programming easier (e.g. IDEs like Xcode)
- ④ You have direct access to memory
- ④ You have access to low-level system features if needed

# CHALLENGES OF USING C

- ④ The language is small
- ④ It's easy to get into trouble, e.g. with direct memory access & pointers
- ④ You must manage memory yourself
- ④ Sometimes code is more verbose than in high-level scripting languages like Python, etc

# STANDARDS

## Ⓢ K & R C (Brian Kernighan and Dennis Ritchie)

- ✓ 1972 First created by Dennis Ritchie
- ✓ 1978 *The C Programming Language* described

## Ⓢ ANSI C

- ✓ 1989 ANSI X.159-1989 aka C89 - First standardized version

## Ⓢ ISO C

## Ⓢ 1990 ISO/IEC 9899:1990 aka C90 - Equivalent to C89

## Ⓢ 1995 Amendment 1 aka C95

## Ⓢ 1999 ISO/IEC 9899:1999 aka C99

## Ⓢ 2011 ISO/IEC 9899:2011 aka C11

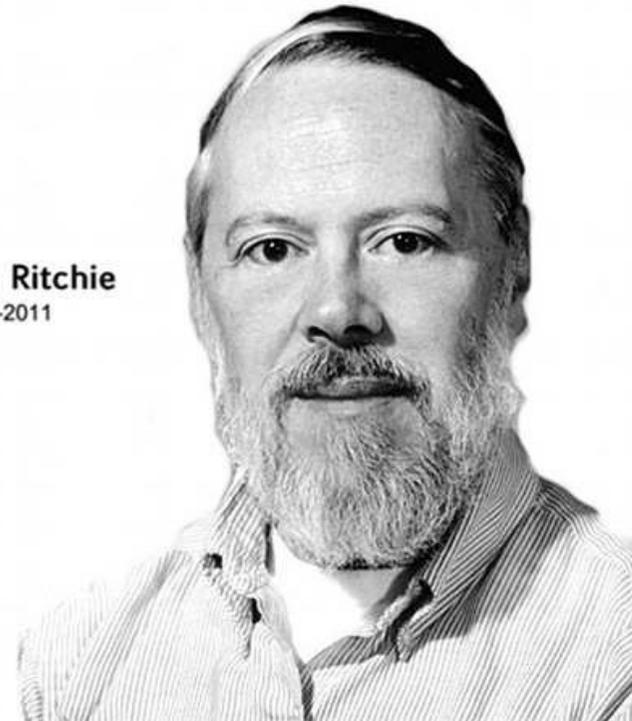
## Ⓢ 2018 ISO/IEC 9899:2018 aka C18



gcc file.c -std=c11

# DENNIS RITCHIE

Dennis Ritchie  
1941-2011



@grimmelm  
James Grimmelm

Dennis Ritchie (1941-2011). His pointer has been cast to void \*; his process has terminated with exit code 0.

# HISTORY OF C++

- ② In the early 1970s, Dennis Ritchie introduced “C” at Bell Labs.
  - ✓ <http://cm.bell-labs.co/who/dmr/chist.html>
- ② As a Bell Labs employee, **Bjarne Stroustrup** was exposed to and appreciated the strengths of C, but also appreciated the power and convenience of higher-level languages like Simula, which had language support for *object-oriented programming* (OOP).
  - ✓ Originally called *C With Classes*, in 1983 it becomes C++
- ② In 1985, the first edition of The C++ Programming Language was released
- ② Standard in 1998 (ISO/IEC 14882:1998)

# HISTORY

- ④ Adding support for OOP turned out to be the right feature at the right time for the '90s. At a time when GUI programming was all the rage, OOP was the right paradigm, and C++ was the right implementation.
- ④ At over 700 pages, the C++ standard demonstrated something about C++ that some critics had said about it for a while: C++ is a complicated beast.

# STROUSTRUP



# STRUCTURE OF C PROGRAMS



# THE STRUCTURE OF C PROGRAMS

- ④ The procedural building blocks of a C program are *functions*, which can invoke one another.
- ④ Every function in a well-designed program serves a specific purpose. Functions cannot be nested one into another.
- ④ The functions contain **statements** for the program to execute **sequentially**, and statements can also be grouped to form *block statements*, or **blocks**.
- ④ You can use the ready-made functions in the standard library (printf()), or write your own.
- ④ Every C program must define at least one function of its own, with the special name **main()**: this is the first function invoked when the program starts. The main( ) function is the program's top level of control, and can call other functions as subroutines.

# PROCEDURAL PROGRAMMING

- ④ Procedural programming is a programming paradigm, derived from structured programming, based on the concept of the procedure call.
- ④ Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out.
- ④ Other paradigms? Object-oriented (Java), Functional, etc...
- ④ C is imperative and procedural
  - ✓ Instructions are grouped into functions

# EXAMPLE

```
mainfile.c
#include <stdio.h>
void myPrintHello();

int main() {
    myPrintHello();
    return(0);
}

void myPrintHello(void) {
    printf("Hello!\n");
    return;
}
```

gcc -o mainfile mainfile.c

gcc mainfile.c  
Target program **a.out**

# MODULAR PROGRAMMING

- ④ Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- ④ For example, separate code in to different files: `mainfile.c` and `hello.c`

# EXAMPLE

mainfile.c

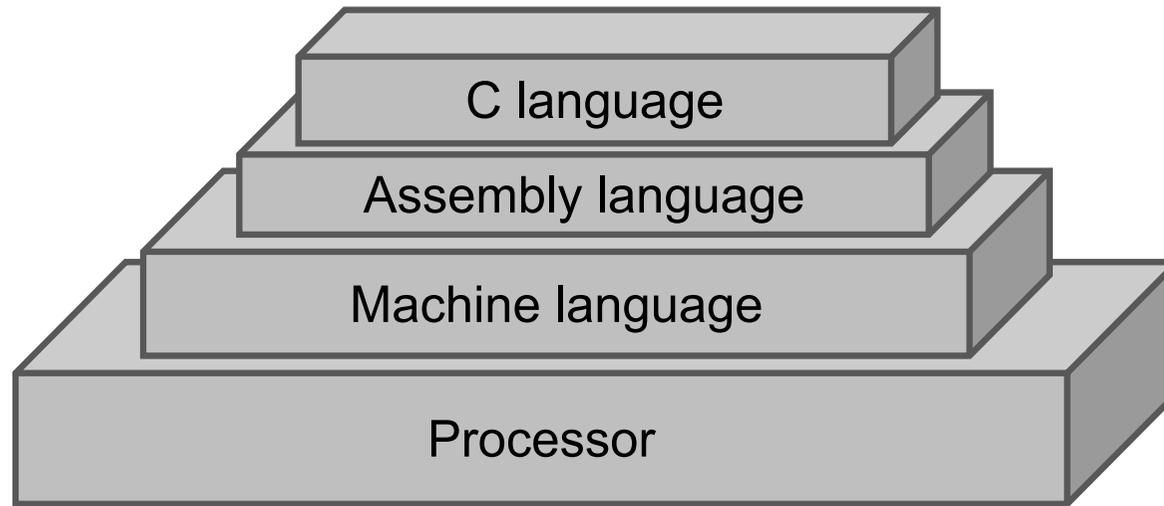
```
int main() {  
    myPrintHello();  
    return(0);  
}
```

hello.c

```
#include <stdio.h>  
  
void myPrintHello(void) {  
    printf("Hello!\n");  
    return;  
}
```

```
gcc -o executable mainfile.c hello.c
```

# STACK OF INTERPRETATION



# FROM THE BEGINNING TO THE END

```
#include <stdio.h>
int main(){
    int a, b;
    printf("Enter first positive integer: \n");
    scanf("%d", &a);
    printf("Enter second positive integer: \n");
    scanf("%d", &b);
    while(b != 0) {
        if(a > b)
            a = a - b;
        else
            b = b - a;
    }
    printf("GCD = %d\n", a);
    return 0;
}
```

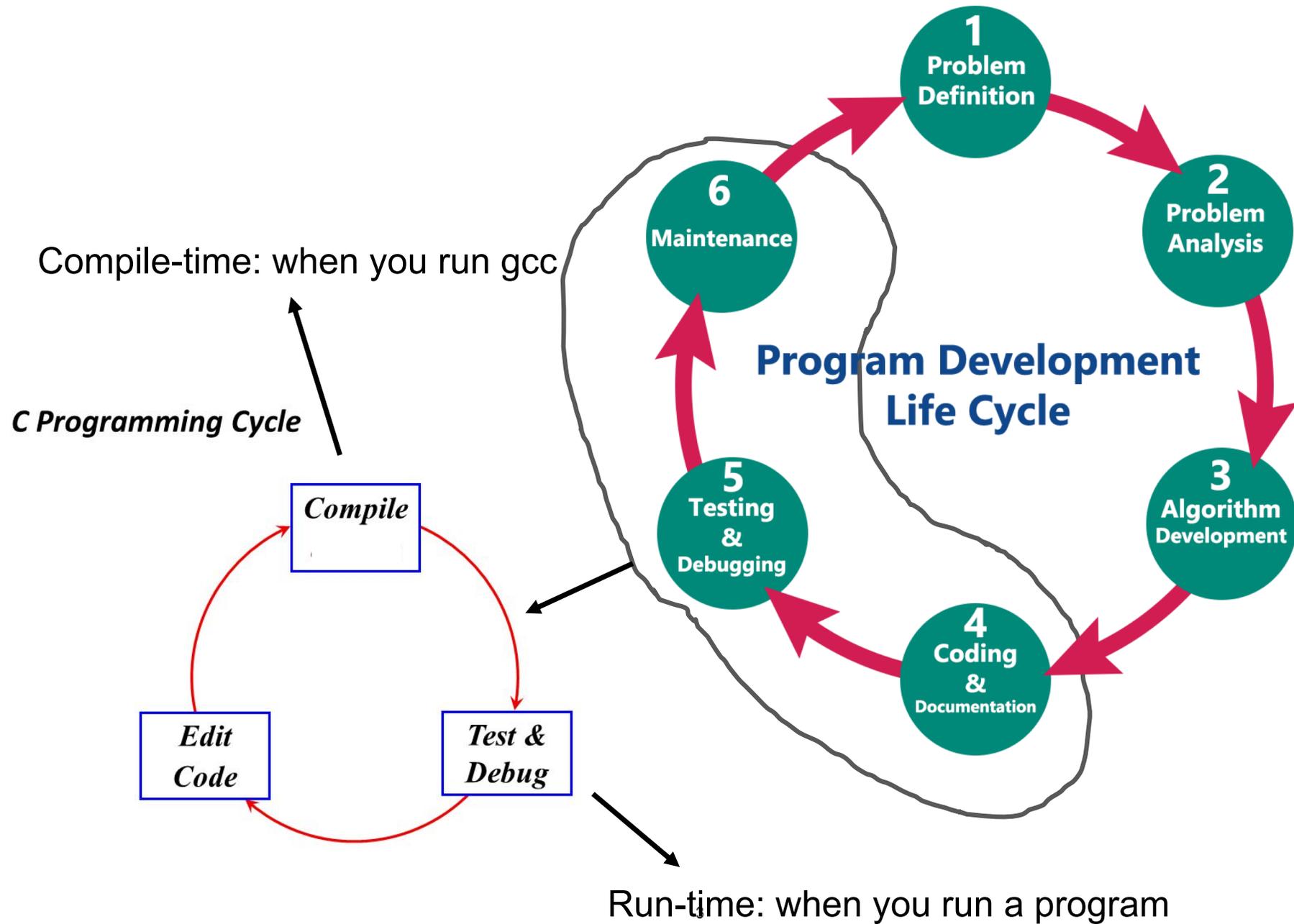
```
    pushl   %ebp
    movl   %esp, %ebp
    pushl   %ebx
    subl   $4, %esp
    andl   $-16, %esp
    call   getint
    movl   %eax, %ebx
    call   getint
    cmpl   %eax, %ebx
    je     C
A:  cmpl   %eax, %ebx
    jle   D
    subl   %eax, %ebx
    cmpl   %eax, %ebx
    jne   A
C:  movl   %ebx, (%esp)
    call   putint
    movl   -4(%ebp), %ebx
    leave
    ret
D:  subl   %ebx, %eax
    jmp   B
```

```
55 89 e5 53 83 ec 04 83 e4 f0 e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00 00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00 8b 5d fc c9 c3 29 d8 eb eb 90
```

# SOME MORE WORDS ON PROGRAMMING



# LIFE OF A PROGRAMMER



# DIFFERENT KINDS OF ERRORS

- ④ A **compile time error** is an **error** that is detected by the **compiler**. Common causes for **compile time errors** include: **Syntax errors** such as missing semi-colon or use of a reserved keyword (such as “while”).
- ④ A **runtime error** is a program **error** that occurs while the program is running. The term is often used in contrast to other types of program **errors**, such as **syntax errors** and **compile time errors**. There are many different types of **runtime errors**.

# COMPILE-TIME

```
#include <stdio.h>

int main(){
    int a, b;
    printf("Enter first positive integer: \n");
    scanf("%d", &a);
    printf("Enter first positive integer: \n");
    scanf("%d", &b)

    while(b != 0) {
        if(a > b)
            a = a - b;
        else
            b = b - a; }
    printf("GCD = %d\n", a);
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc -o euclid
euclid.c
```

```
euclid.c:9:20: error: expected ';' after expression
```

```
    scanf("%d", &b)
                ^
```

```
    ;
```

```
1 error generated.
```

# RUN-TIME

```
#include <stdio.h>
int main()
{
    int *a= NULL;

    printf("%d", *a);

    int i= 0;
    while (i < 4) {
        printf("Hello World");
        i++;
    }

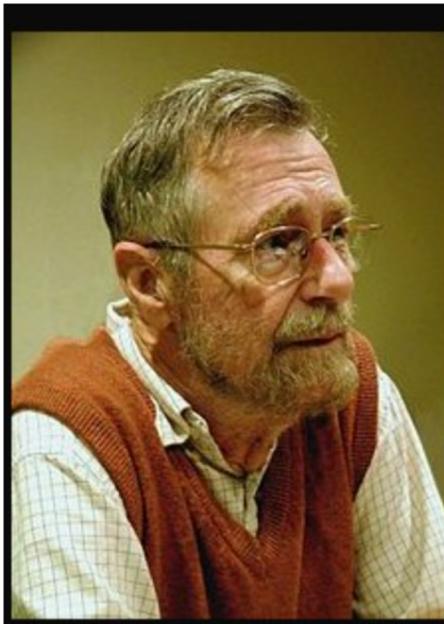
    return 0;
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc -o
runtime_error runtime_error.c
MacBook-Francesco:ProgrammI francescosantini$
./runtime_error
Segmentation fault: 11
```

# YOU NEED TO

- ② You need to understand the output of the compiler to remove all compile-time errors
- ② But it is not over: you need to extensively test the program with different inputs, and remove all (“as more as possible”) run-time errors

*“Testing shows the presence, not the absence of bugs” [Dijkstra]*



If debugging is the process of removing bugs, then programming must be the process of putting them in.

(Edsger Dijkstra)

# WARNINGS

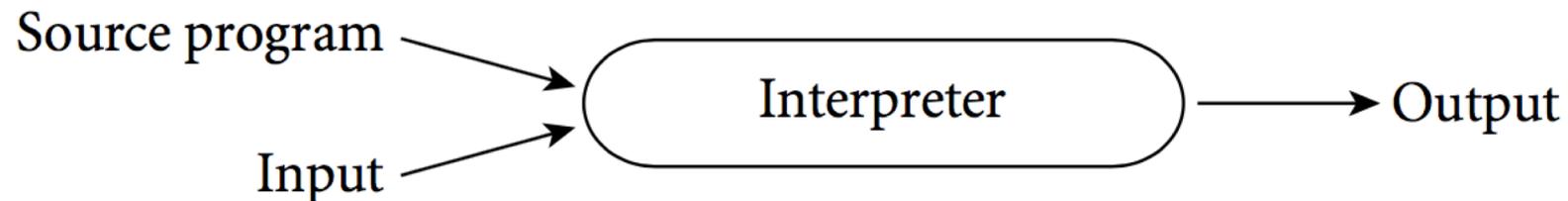


# OTHER PARADIGMS

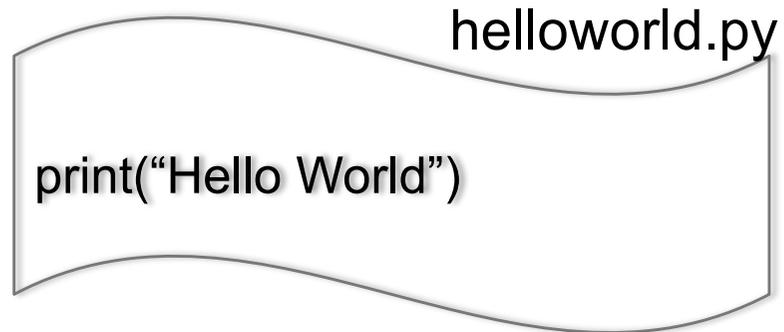


# INTERPRETED LANGUAGES

- ④ An alternative style (w.r.t. compilation) of implementation for high-level languages is known as *interpretation*.
- ④ Unlike a compiler, an interpreter stays around for the execution of the application.
- ④ The interpreter implements a virtual machine whose “machine language” is the high-level programming language.



# AN EXAMPLE WITH PYTHON



```
MacBook-Francesco:~ francescosantini$ python3 helloworld.py  
Hello World
```

# DIFFERENCES

- ⊙ We say that a language is compiled if the translator analyzes it thoroughly (rather than effecting some “mechanical” transformation), and if the intermediate program does not bear a strong resemblance to the source.
- ✓ These two characteristics—thorough analysis and nontrivial transformation—are the hallmarks of compilation

# DIFFERENCES

- ④ Certain languages (e.g., Python) are sometimes referred to as “interpreted languages” because most of their error checking must be performed at run time.
- ④ Certain other languages (e.g., C and C++) are sometimes referred to as “compiled languages” because almost all of their semantic error checking can be performed at compile time.

# SU LIBRO

- ④ Sezione 1.9. Rivedremo bene i concetti in 1.9.2-1.9.7 alla fine del corso.
- ④ Sezione 2.1-2.3.