C Programming (W3)

Welcome!!

Please check attendance individually.

(Mobile App)

Things to do today

01 How does a program work?

02 Explain main.c & Debugging

03 Standard IO

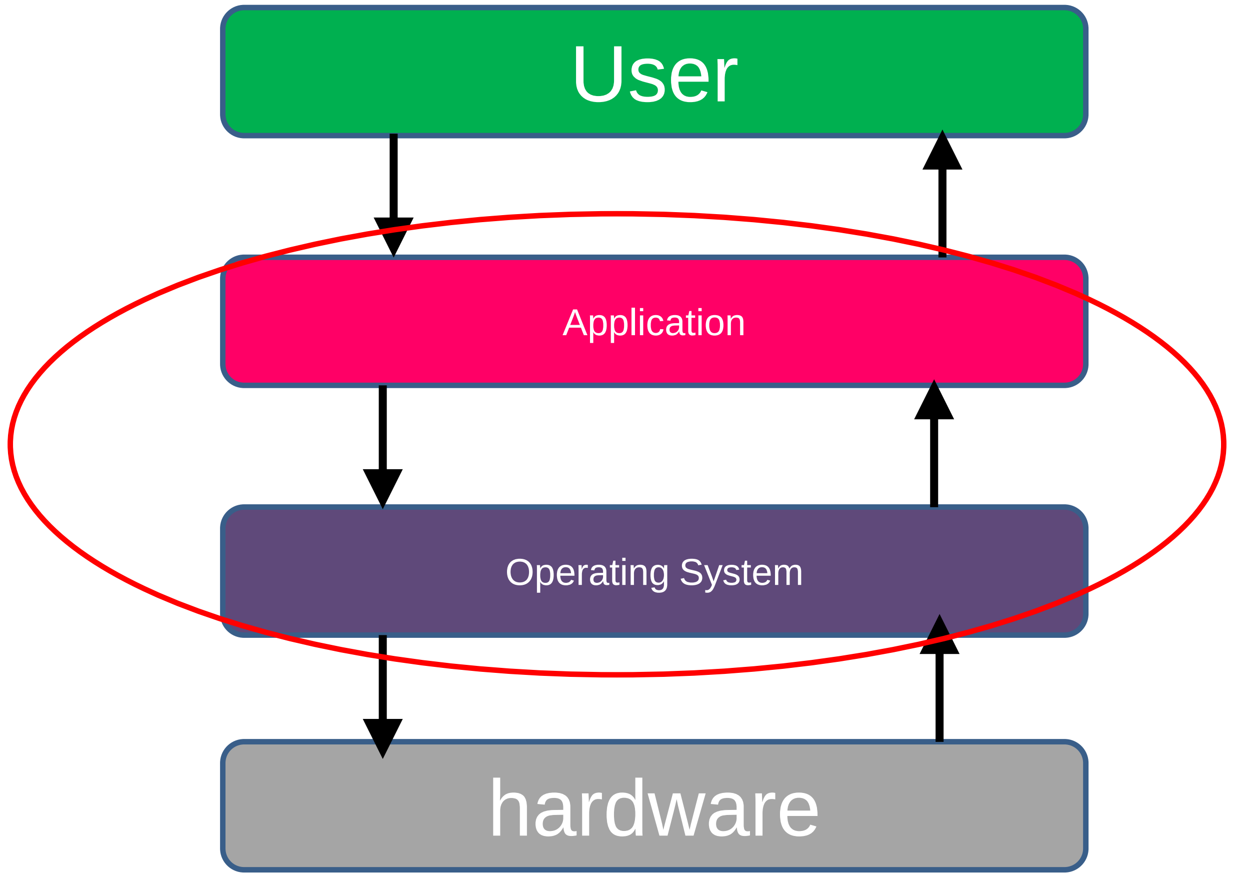
04 Codyssey (Requirement of C1-P1)

1. Install VSC (MinGW64)
2. Create github repositories
3. Register Codyssey
4. Build main.c

5. Start codyssey project
   - Join team
   - Apply to join (3 hours, a week)

What if there is no program on the computer ?

Without computer hardware and no programs, a computer is just a useless machine that generates some heat and noise .
“ Windows” and additionally installing various applications .



Software vs. Program vs. Application

- All applications are software, but not all software is applications.
  . Software is the broadest term, encompassing all digital programs and data.
  . Program refers to a specific set of instructions that tell a computer what to do.
  . Application is a type of software designed for end users to perform specific tasks

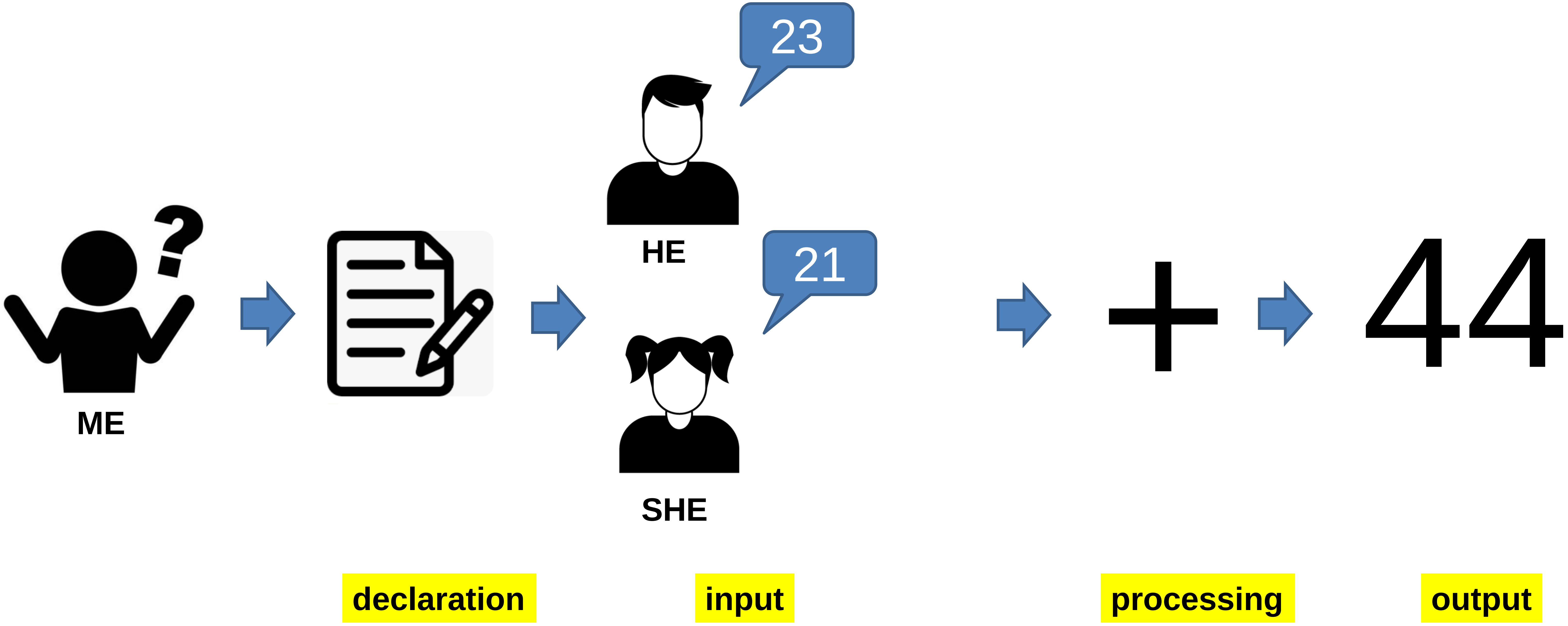
Category	Software	Program	Application
Definition	A broad term that includes all programs and data running on a computer	A set of instructions written to perform a specific task	A type of software designed for end users to perform specific tasks
Scope	Includes operating systems, drivers, utilities, and applications	A subset of software that consists of executable code	A subset of software that provides user-oriented functionality
Purpose	Manages hardware, provides system functionality, and supports applications	Performs a specific operation or task within a system	Enables users to complete tasks like document editing, communication, or browsing
User Interaction	Can run in the background (e.g., OS, drivers) or be user-facing	May or may not be user-facing (e.g., a script or background process)	Always designed for direct user interaction
Examples	Windows, macOS, Linux, firmware, database software	A simple Python script, a sorting algorithm, a file copy script	Microsoft Word, Google Chrome, Instagram

Think over

I am trying to write a C program that takes two people's ages and calculates their sum, following the input instructions.

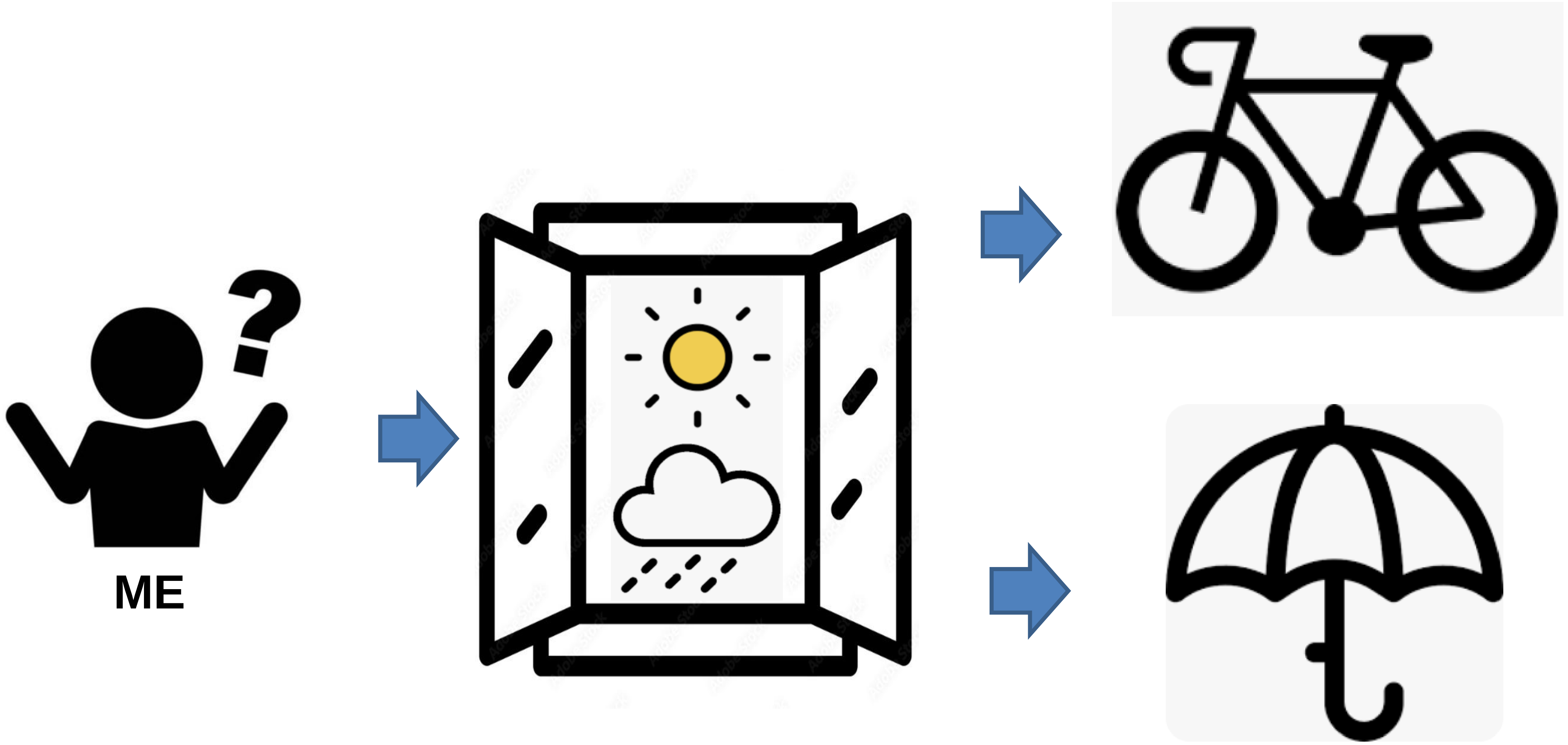
What is Programing?

- Program operation process



What is Programing?

- Program operation process

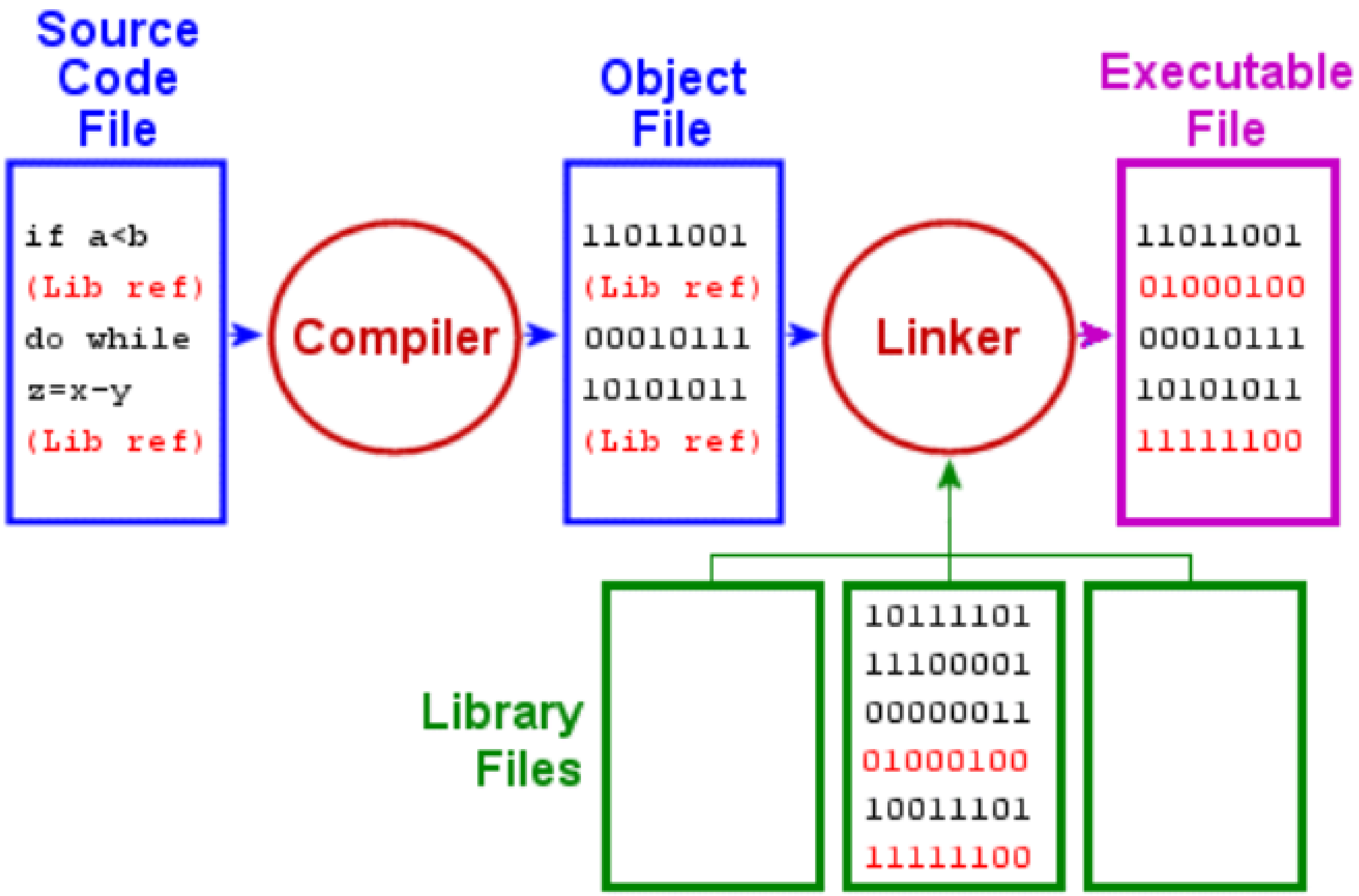


```
int Bicycle, Umbrella;

If(weather == “rainy”)
{
    printf(“Umbrella”);
}
else
{
    printf(“Bicycle”);
}
```

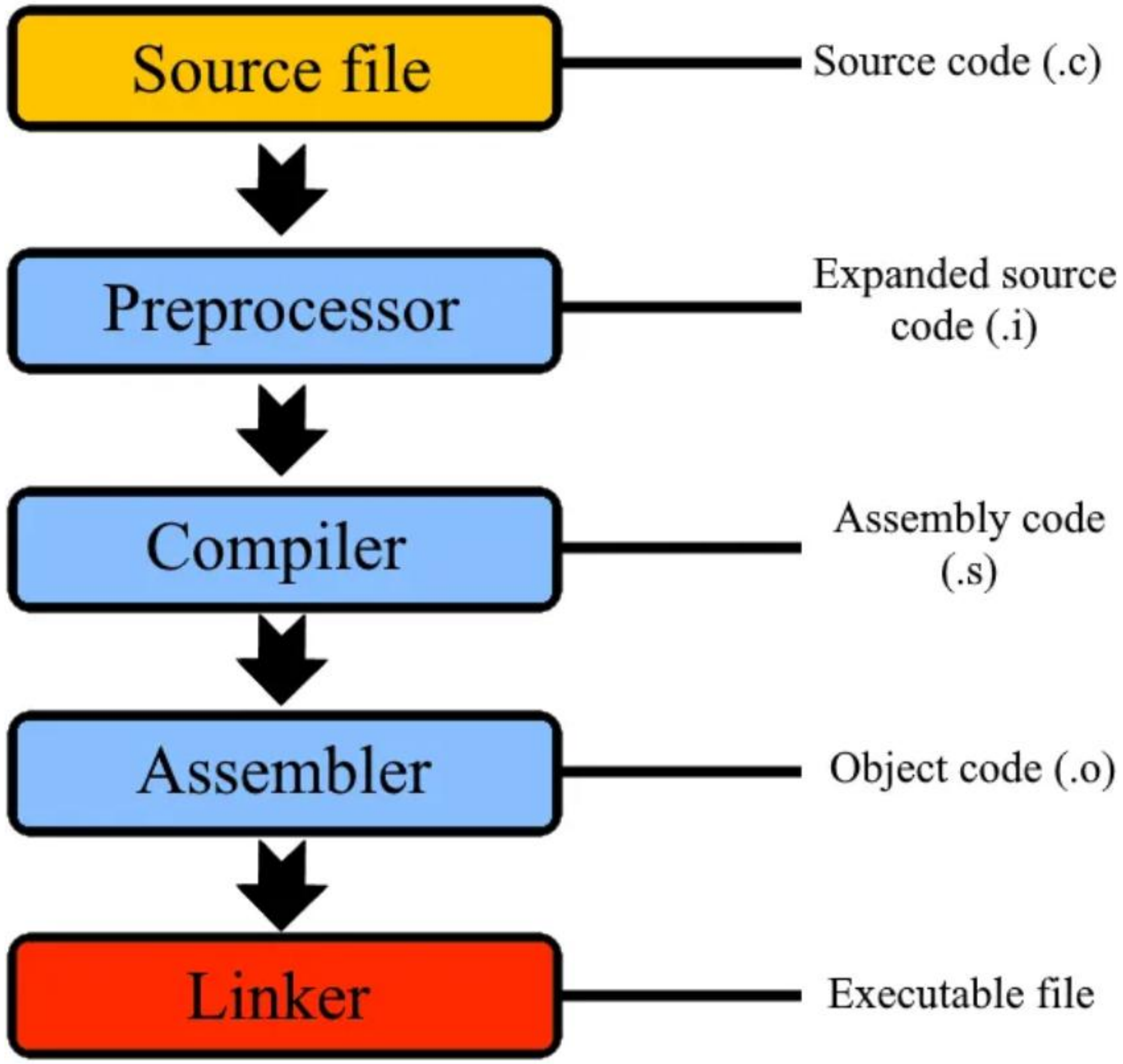
What is compile?

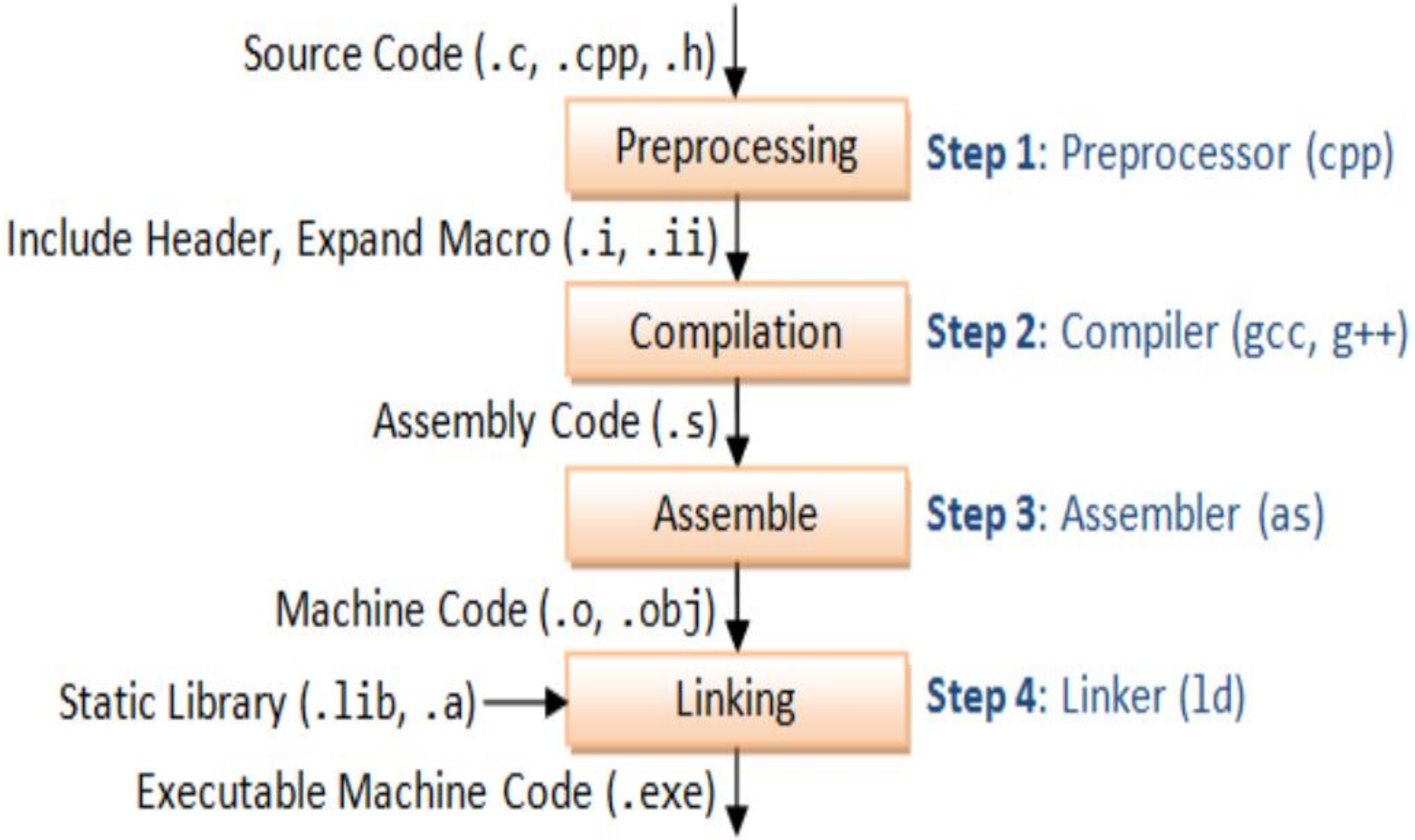
- Translating a high-level language created by humans into a low-level language that a computer can interpret.



Compile Process

- Source code : https://github.com/lattera/glibc/tree/master/stdio-common




Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

Compile Process

1. main.c : Source code

□ Preprocessor : Processes headers (#include) and macros (#define)

2. main.i : Intermediate file containing expanded source code

□ Compiler : Edits preprocessed source code into assembly code for a specific processor

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

3. main.s : Assembly file

□ Assembler : Converts assembly code into machine code

```
gcc -S main.i -o main.s
```

4. main.o : Object file

□ Linker : Creates an executable file using the object file and library

```
gcc -c main.s -o main.o  -> objdump –d main.o / nm main.o
```

5. main.exe : Executable file

```
Gcc main.c –o main -> hexdump -C main
```

Compiler vs. Interpreter

Compiler is a program that translates the entire source code of a programming language into machine code (binary) **before execution**. The resulting executable file can be run independently without requiring the original source code.

Examples of compiled languages:
C, C++, Rust, Go

Characteristics:
- Translates the entire program at once. (Modify-> Compile all again)
- Generates a separate executable file.
- Faster execution since the program is already compiled.
- Errors are detected before execution.

Compiler vs. Interpreter

Interpreter is a program that translates and executes code **line by line** at runtime. It does not generate a separate executable file; instead, it processes the source code dynamically.

Examples of interpreted languages:
Python, JavaScript, Ruby, PHP

Characteristics:

- Translates and runs the code **line by line**.
- No separate executable file; the interpreter is needed each time.
- Slower execution compared to compiled programs.
- Errors are detected **during execution** (runtime).

Compiler vs. Interpreter

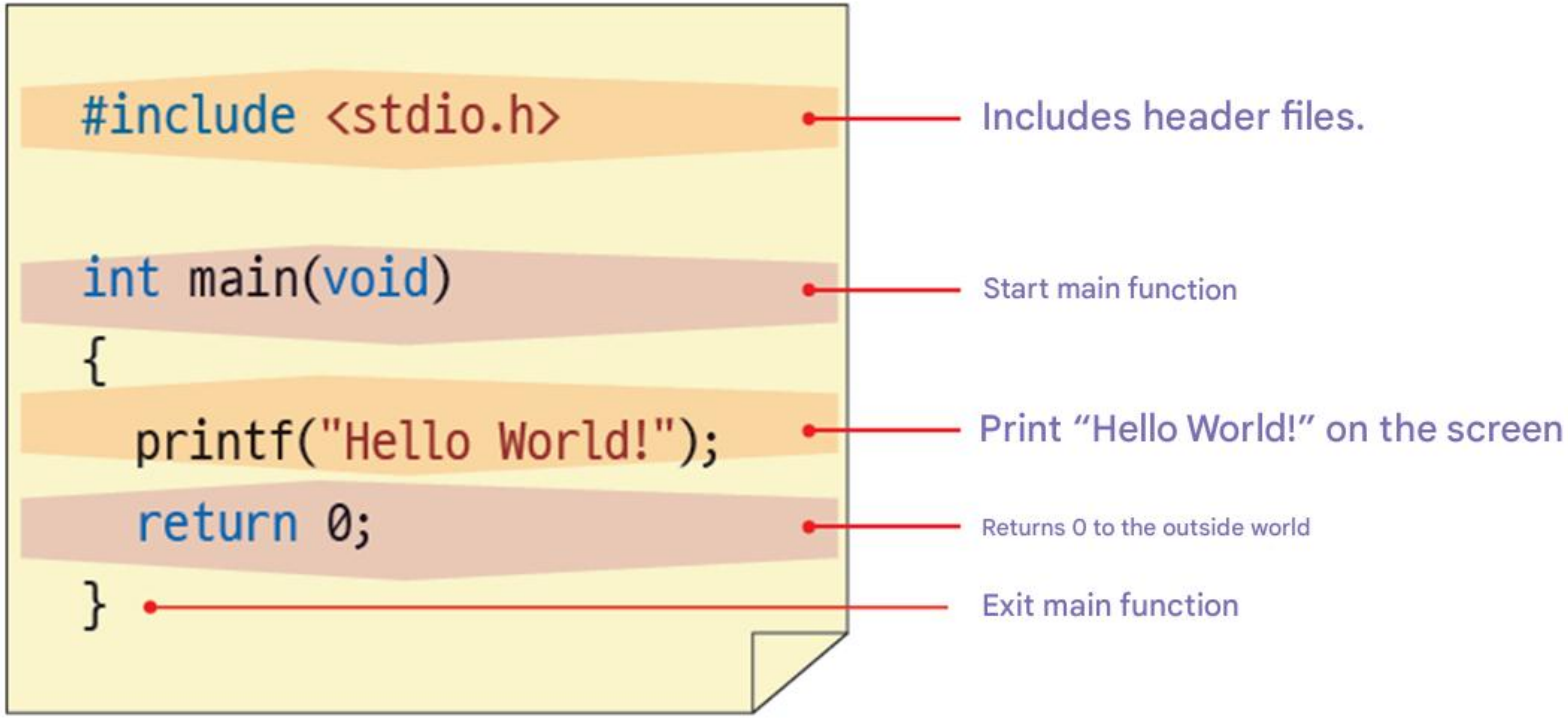
Key Differences

Feature	Compiler	Interpreter
Execution Speed	Fast (precompiled)	Slow (line-by-line)
Error Detection	Before execution	During execution
Output	Executable file	No separate executable
Usage	C, C++, Rust	Python, JavaScript

Some languages, like **Java**, use both:

- Java source code is compiled into bytecode (.class file).
- The JVM (Java Virtual Machine) interprets the bytecode at runtime.

Brief source description



```c
#include <stdio.h>

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

Includes header files.

Start main function

Print “Hello World!” on the screen

Returns 0 to the outside world

Exit main function

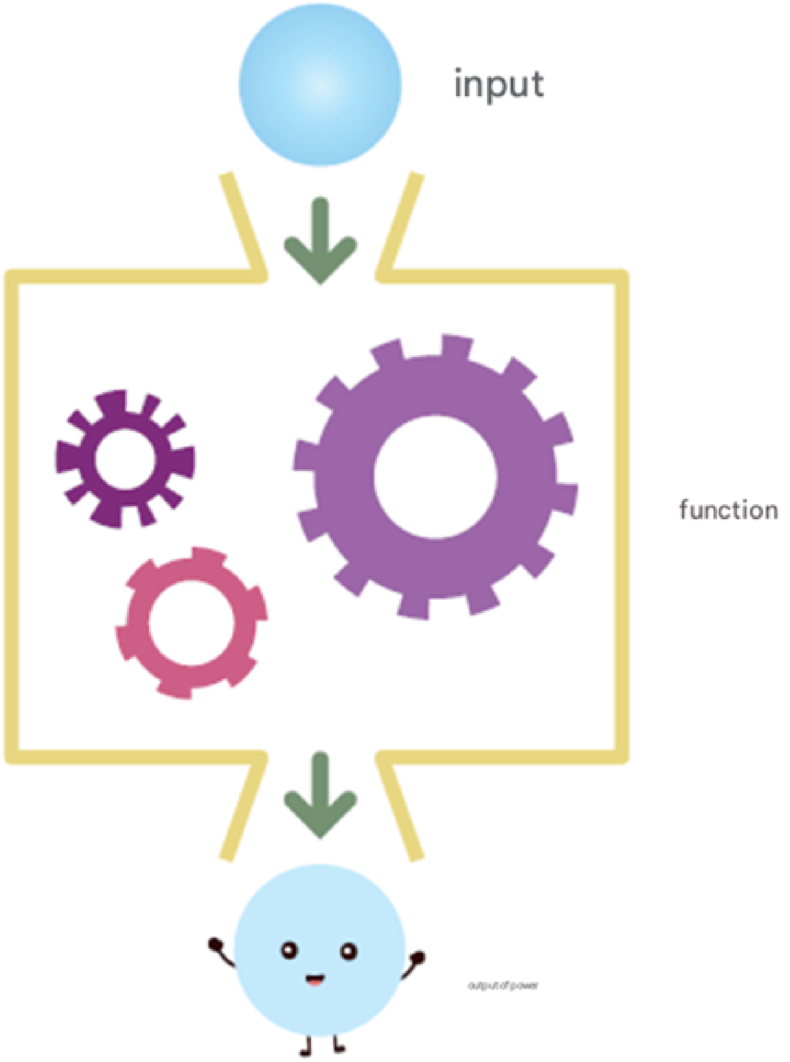
program

Function

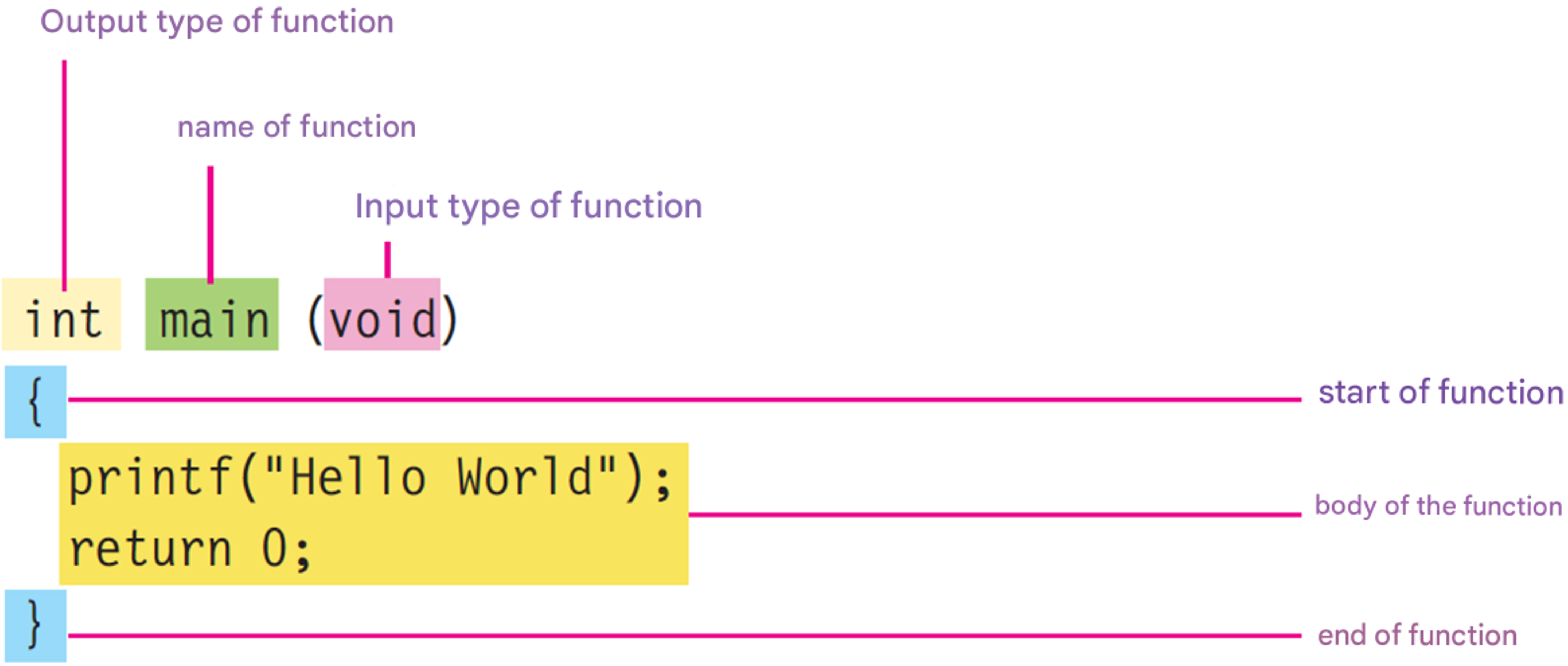
- Function : A standalone piece of code written to perform a specific task.
- (Reference) Mathematical function

$$y = x^2 + 1$$

- program = set of functions



Brief description of the function



Output type of function

name of function

Input type of function

```
int main (void)
{
  printf("Hello World");
  return 0;
}
```

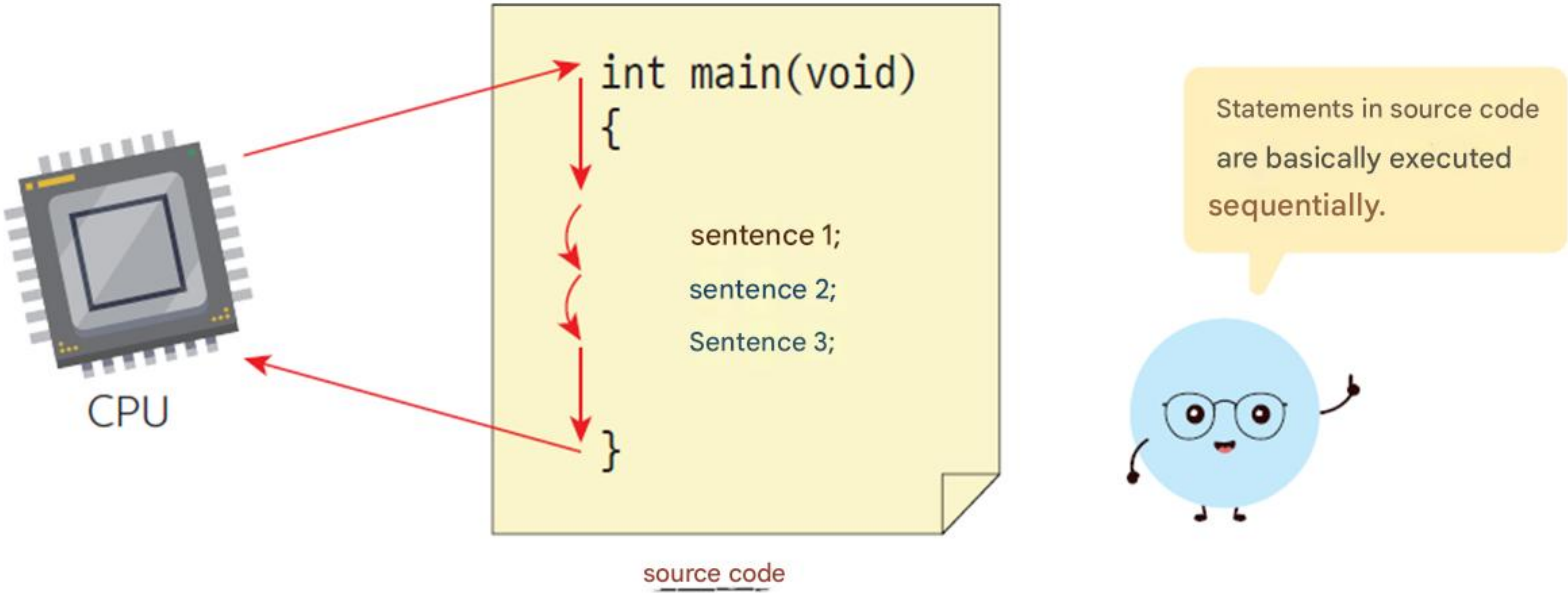
start of function

body of the function

end of function

Sentence (Imperative)

- A function consists of several statements .
- Sentences are executed sequentially .
- There must be a ; at the end of a sentence .



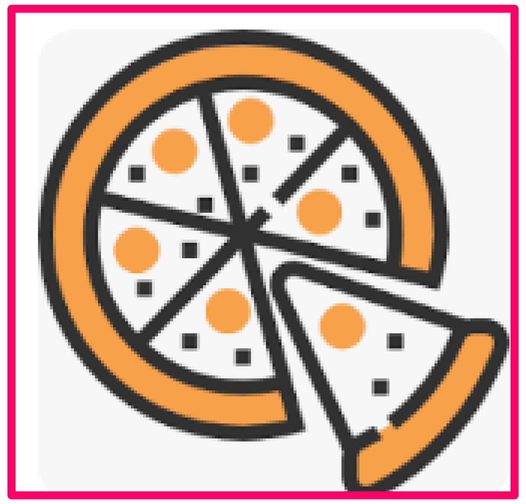
```
int main(void)
{
        sentence 1;
        sentence 2;
        Sentence 3;
}
```

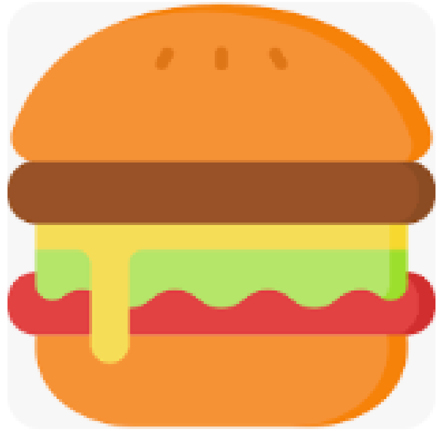
Statements in source code are basically executed sequentially.

Think over

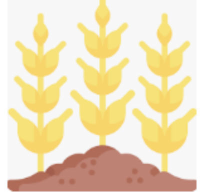
You are on an island with nothing. You want to cook pizza or spaghetti. Think about what you need to prepare. Imagine a list of things you need.

Let’s cook





Making pizza base

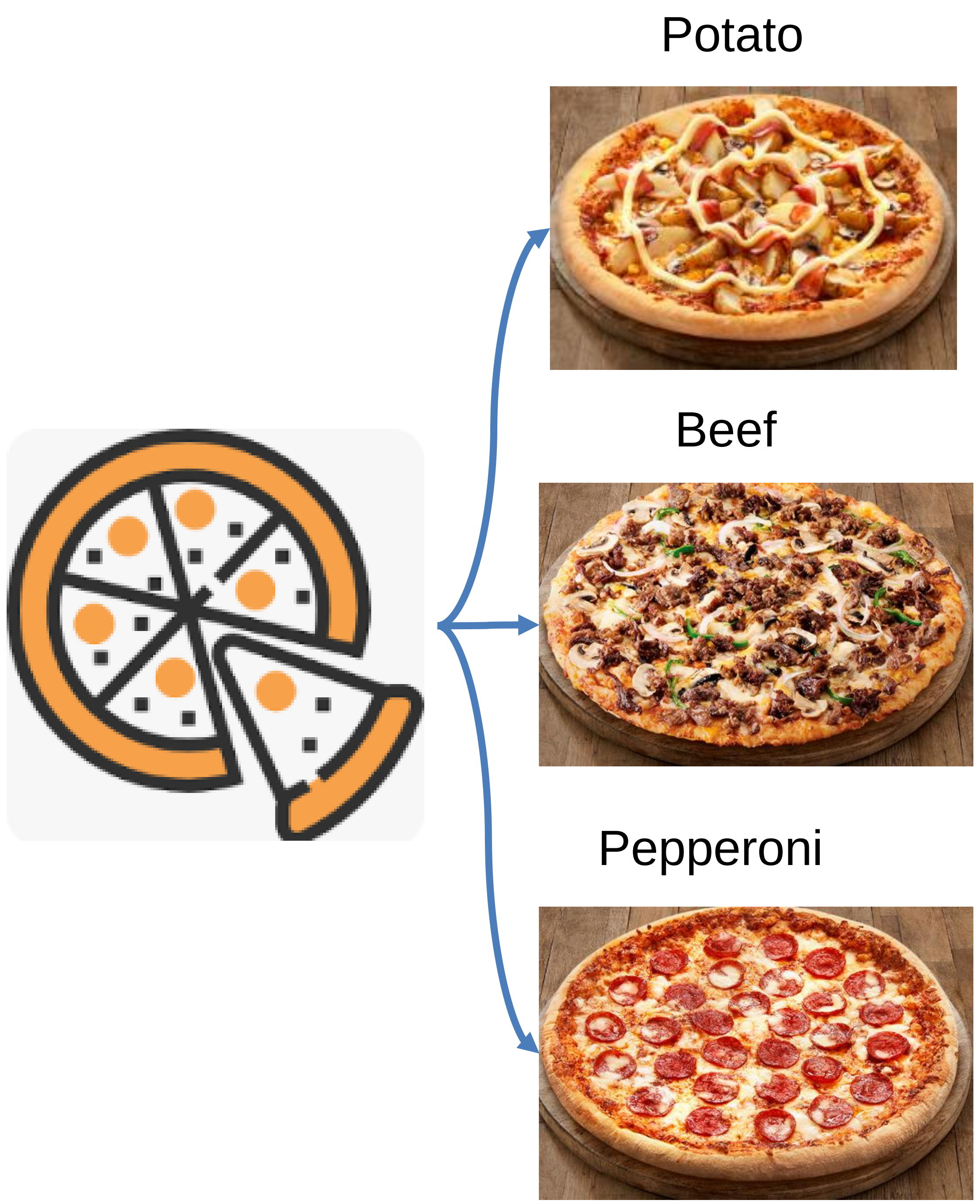


Cooking pizza

The real appearance of printf function

- Source code : https://github.com/lattera/glibc/tree/master/stdio-common

Let’s make pizza



Potato

Beef

Pepperoni

What should I consider?

1. Select dough(가루 반죽)

* Crust / Base

Thin / Thick
What kinds of wheat flour(밀가루)

2. Select sauce / cheese / topping

Tomato, Olive oil, Garlic, Herb, BBQ Sause
Mozzarella, Gorgonzola, …
Potato, Pepperoni, Beef, ….

3. Baking condition

Oven – 15 mins
Pan - 30 mins

Programming Process

- The programming process is similar to the cooking process.

1. Problem Analysis & Requirements Definition (Preparation)

2. Selecting Data & Tools (Gathering Ingredients)

3. Coding, Implementation (Cooking)

4. Debugging & Testing (Tasting & Adjusting)

5. Deployment & Release (Serving the Dish)

6. Maintenance (Feedback & Improvement)

Programming Process

1. Problem Analysis & Requirements Definition

(Programming) Understanding the problem and defining requirements

- Decide what program to create
- Define the necessary features and functionalities

(Cooking) Deciding what dish to make before cooking

- Choose a menu and find a recipe
- Identify the ingredients needed

Example:

Program: "Create a user login system”

Dish: "Make pasta"

Programming Process

2. Selecting Data & Tools (Gathering Ingredients)

(Programming) Choosing the programming language, tools, and data

- Decide on the programming language (Python, Java, C++, etc.)
- Select necessary libraries and frameworks, IDE

(Cooking) Preparing ingredients before cooking

- Gather pasta, olive oil, garlic, tomato sauce, and noodles
- Prepare kitchen tools like a frying pan and pot

Example:

Program: C + Standard Lib + VSCode + Github

Dish: Using spaghetti with tomato sauce

Programming Process

3. Coding, Implementation (Cooking)

(Programming) Writing the actual code to build the program

- Define variables, write functions, and implement algorithms
- Be mindful of errors and bugs

1. Design (Define requirement)
2. Write source code
3. Compile & Link
4. Execute a program
5. Debugging
6. Store & Maintaining

(Cooking) Preparing and cooking the dish

- Chop the ingredients (garlic), boil the pasta
- Control the heat while cooking the sauce and mixing everything

Example:

Program: Writing a void main(): function and implementing “hello world!”

Dish: Fry the garlic, boil the sauce and cook with the noodles

Programming Process

4. Debugging & Testing (Tasting & Adjusting)

(Programming) Checking if the program works correctly and fixing errors

- Find & fix bugs
- Test with different inputs to ensure reliability

(Cooking) Tasting the dish and adjusting flavors

- If it’s too bland, add salt; if it’s too salty, add water
- Adjust seasoning for the best taste

Example:

Program: Fixing a bug where login credentials are not verified correctly

Dish: Adjusting the seasoning by adding salt or pepper

Programming Process

5. Deployment & Release (Serving the Dish)

(Programming) Deploying the program for users

- Storing at github or publishing an app
- Sharing it with users

(Cooking) Serving the finished dish

- Plating the food in an appealing way
- Serving it to family or customers

Example:

Program: Deploying the website on AWS or Github

Dish: Serving the pasta to guests

Programming Process

6. Maintenance (Feedback & Improvement)

(Programming) Updating and improving the program based on user feedback

- Adding new features and security updates
- Continuous maintenance and bug fixes

(Cooking) Improving the dish based on feedback

- If guests say the dish is too salty, adjust it next time
- Experiment with new recipes to enhance flavors

Example:

Program: Optimizing login speed if it's slow

Dish: "The pasta is overcooked" → Reduce boiling time

Algorithm

Q) Can anyone cook if they just learn how to use an oven and have the ingredients ?

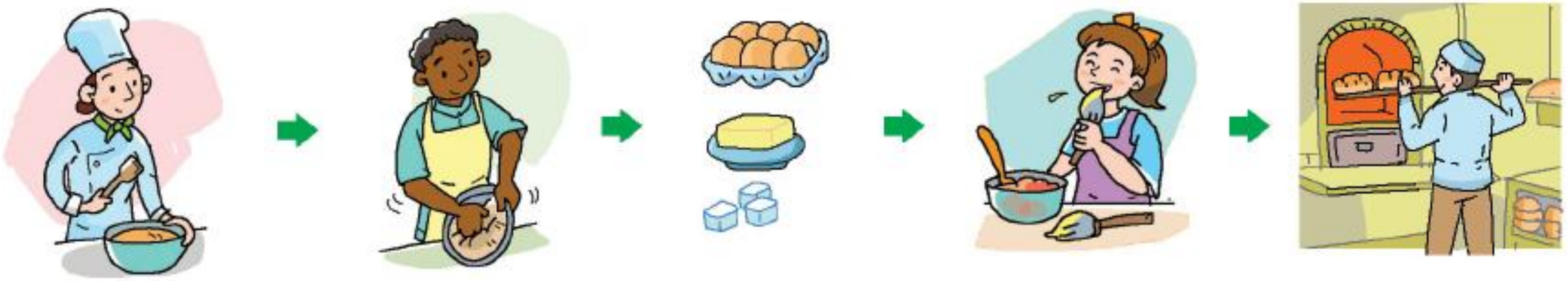
A) You need to know how to cook .

Algorithm

computer

Algorithm for making bread

① Prepare an empty bowl .
② Add yeast to flour and milk and stir .
③ Add butter , sugar , and eggs and mix .
④ Leave in a warm place to ferment.
⑤ Bake in an oven at



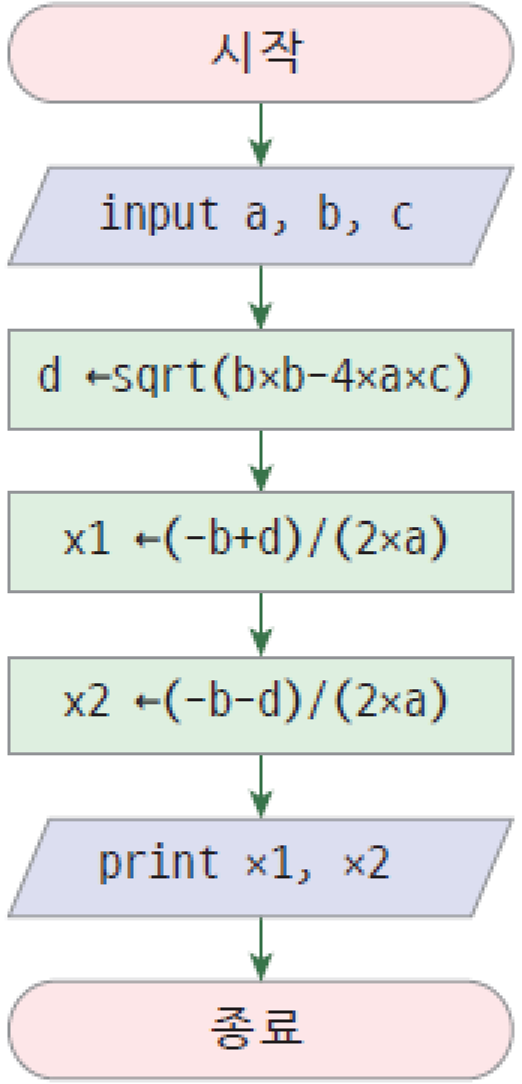
The Art of Algorithms

- Natural language (natural language)
- Flowchart
- pseudo-code

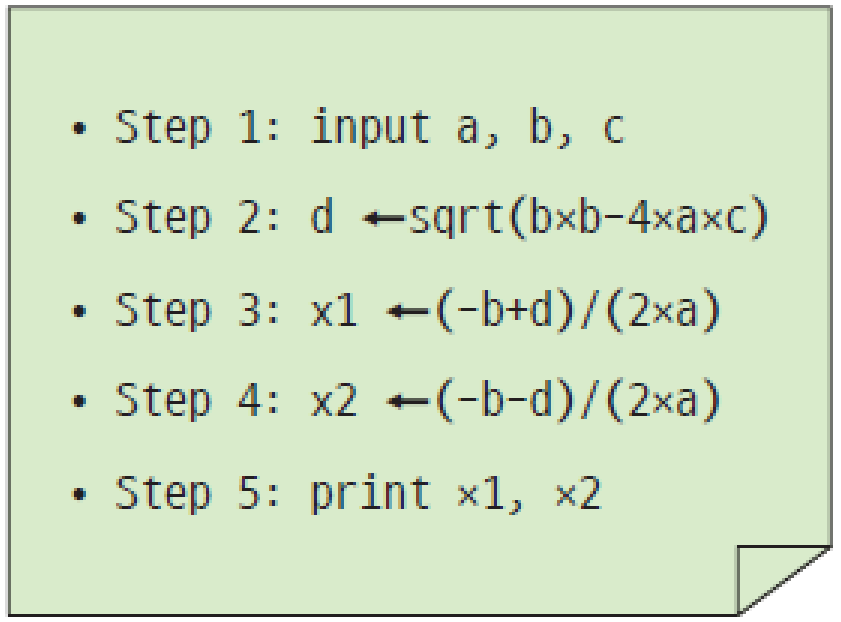
You need to design algorithms without sitting right in front of your computer.

Example of an algorithm

- Algorithm for finding roots of quadratic equations



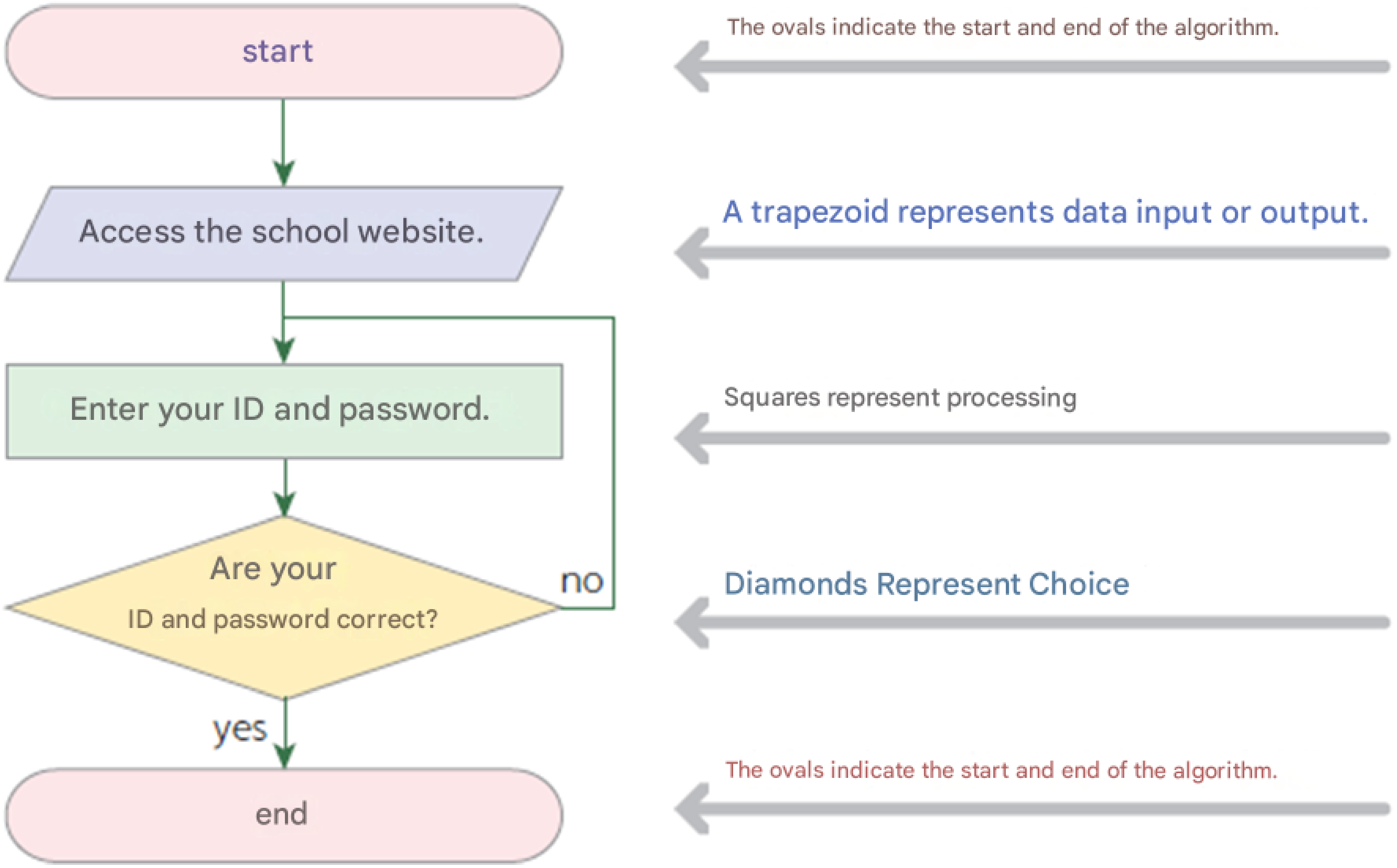
```
시작
input a, b, c
d ←sqrt(b×b-4×a×c)
x1 ←(-b+d)/(2×a)
x2 ←(-b-d)/(2×a)
print ×1, ×2
종료
```



- Step 1: input a, b, c
- Step 2: d ←sqrt(b×b-4×a×c)
- Step 3: x1 ←(-b+d)/(2×a)
- Step 4: x2 ←(-b-d)/(2×a)
- Step 5: print ×1, ×2

Example of an algorithm

- Let's show the algorithm for logging into the school homepage in a flowchart .



start

Access the school website.

Enter your ID and password.

Are your

ID and password correct?

no

yes

end

The ovals indicate the start and end of the algorithm.

A trapezoid represents data input or output.

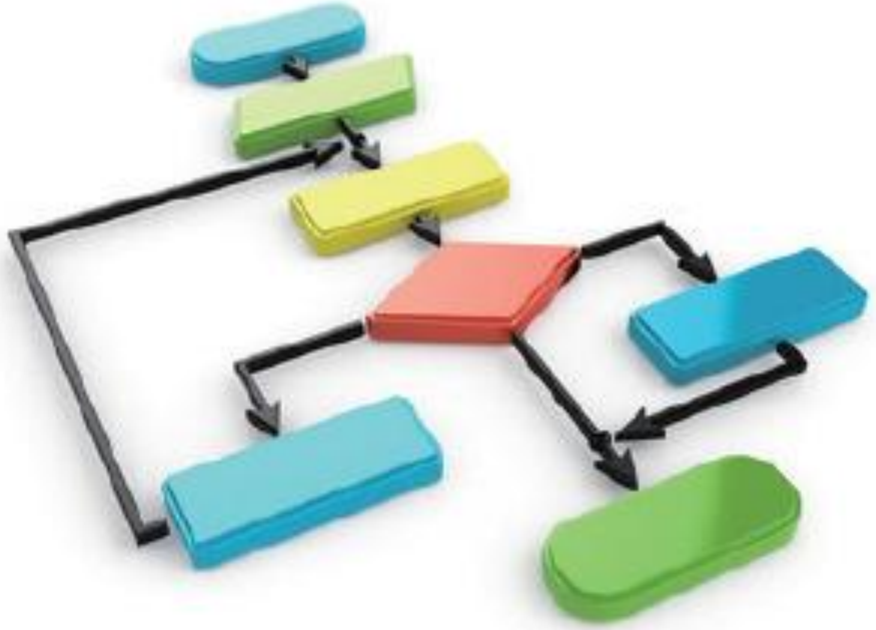
Squares represent processing

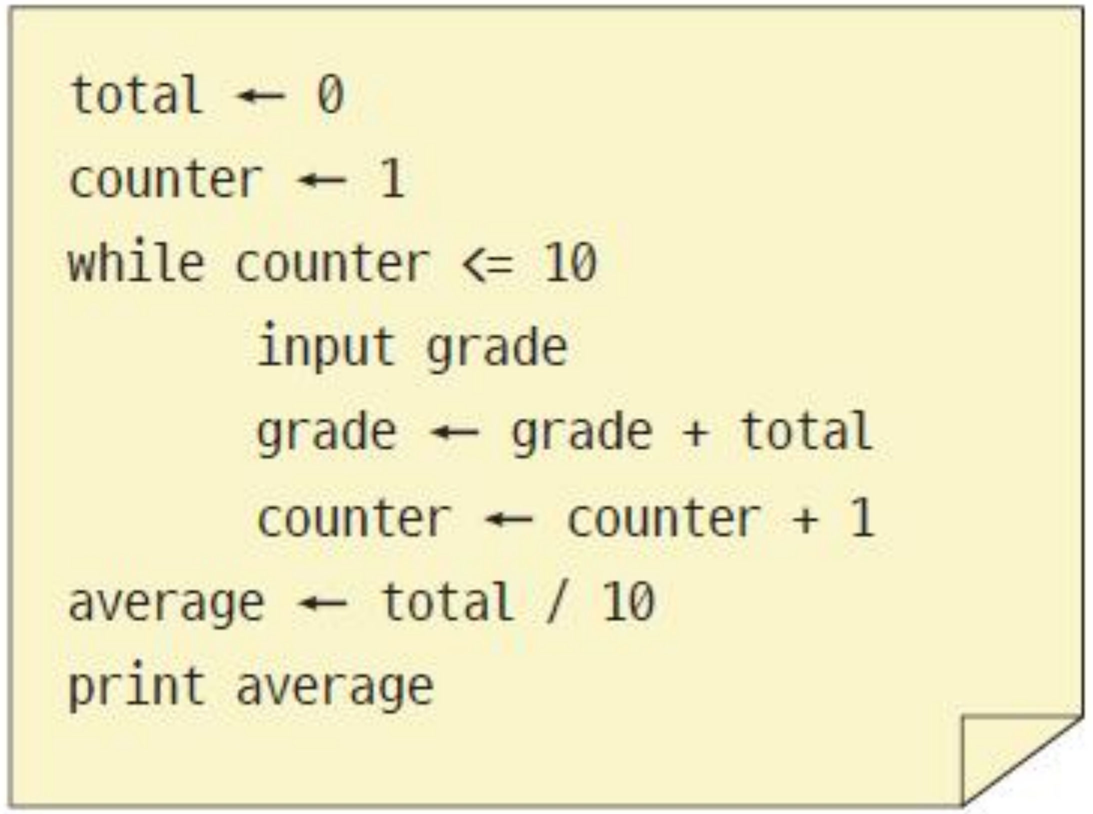
Diamonds Represent Choice

The ovals indicate the start and end of the algorithm.

Pseudocode

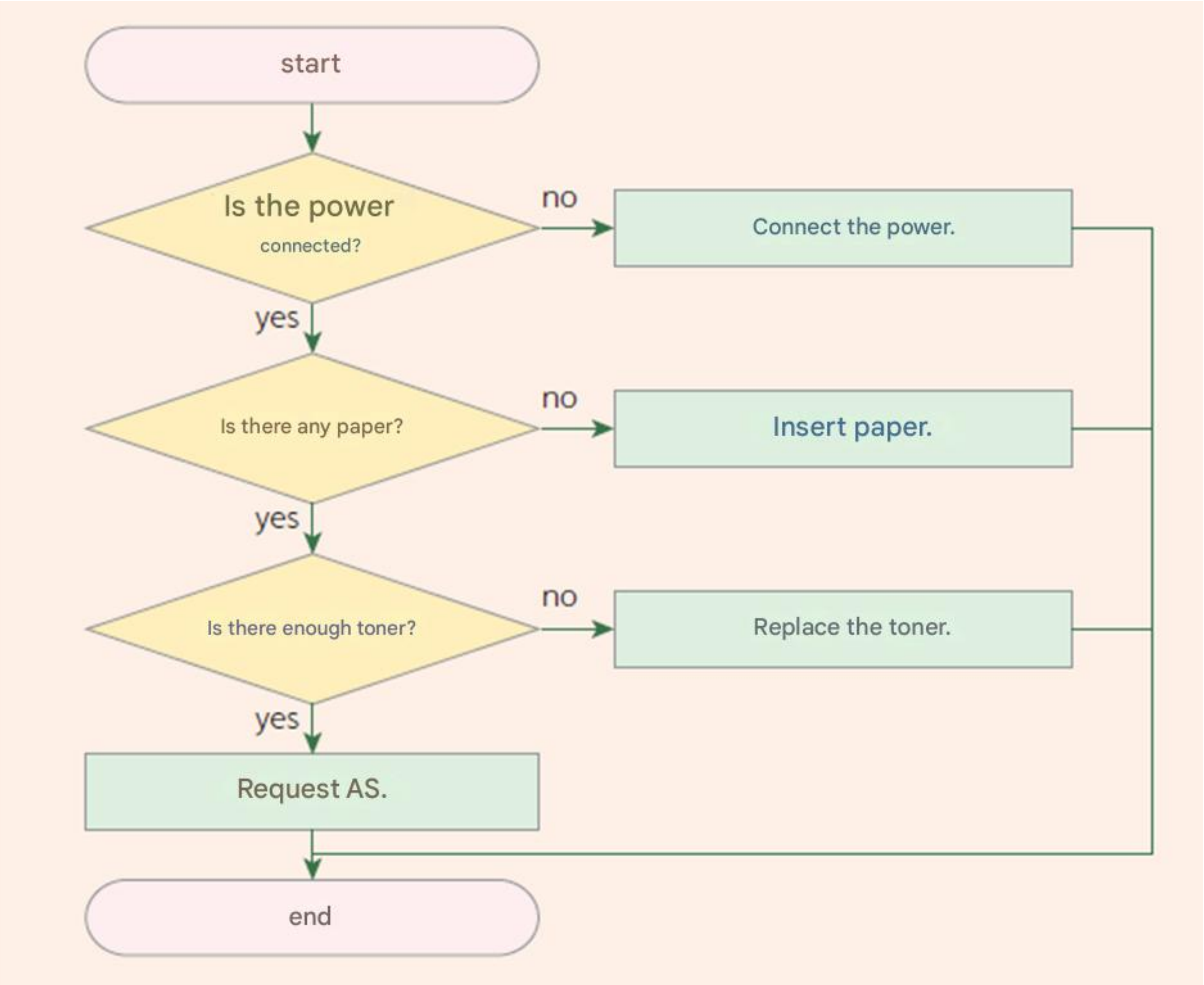
- Pseudo code is a code that is more systematic than natural language and is mainly used to express algorithms.
- For example, the algorithm that takes the grades of 10 students and calculates the average can be expressed in pseudocode as follows :



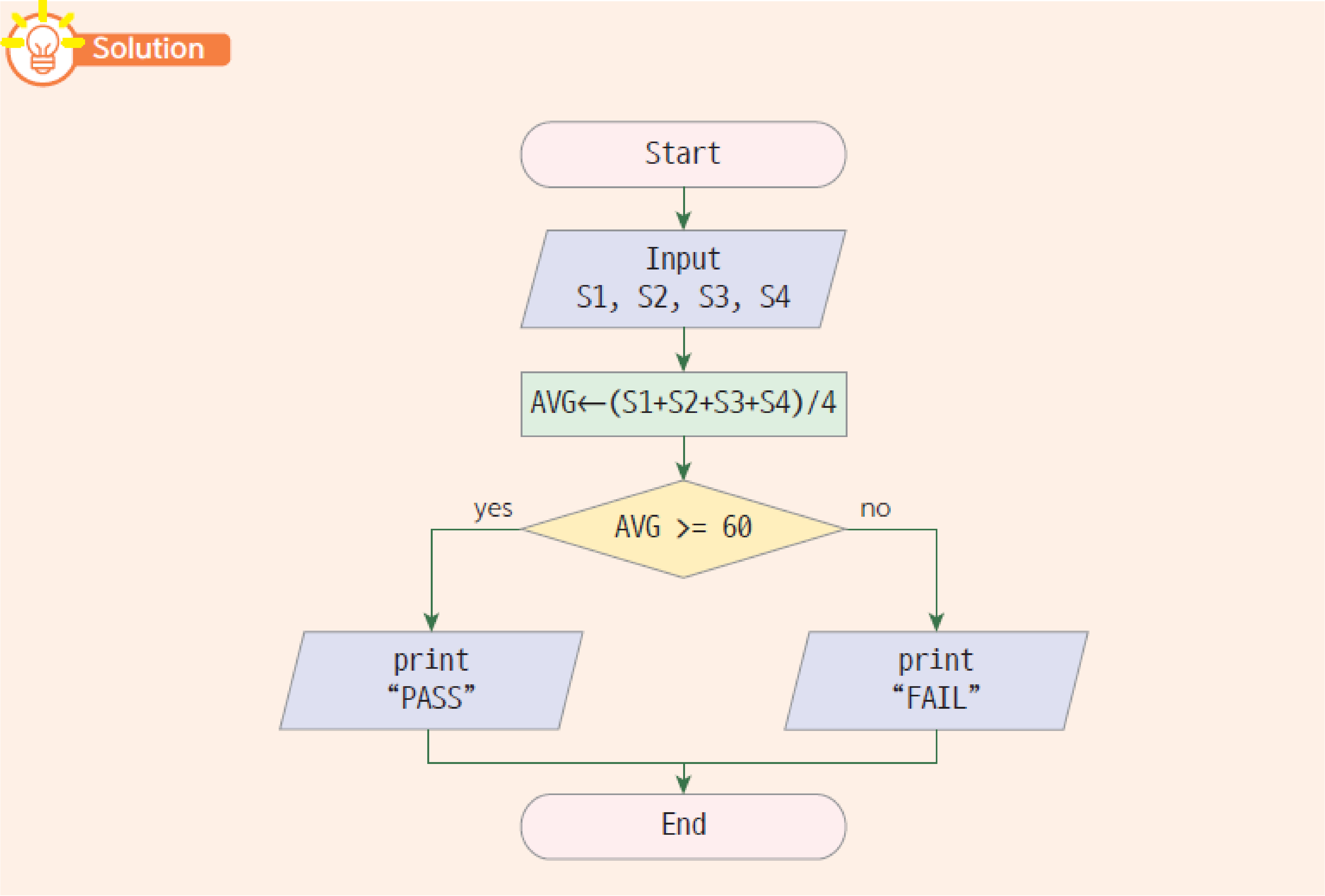


```
total ← 0
counter ← 1
while counter <= 10
        input grade
        grade ← grade + total
        counter ← counter + 1
average ← total / 10
print average
```

Algorithm for handling printer failures

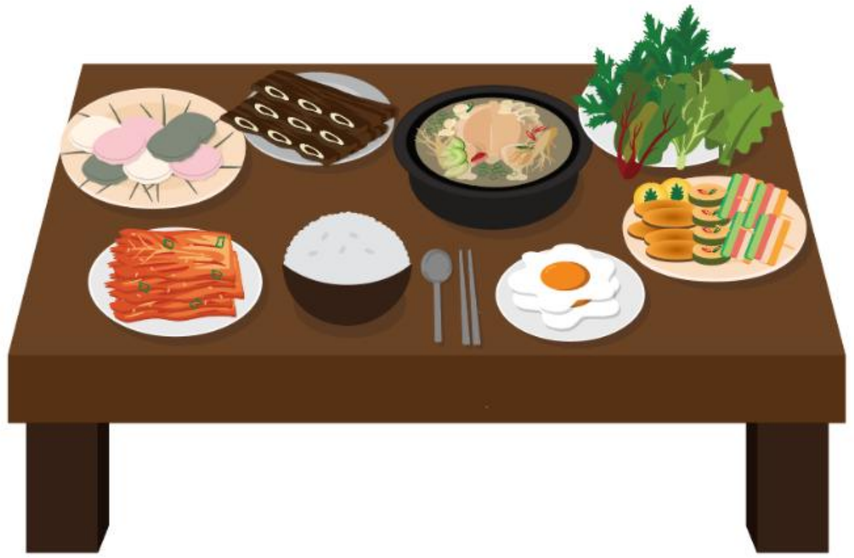


Algorithm for determining pass or fail



Variables & Data types

- Variable: A memory space where **data** can be stored. (bowl)
  - The bowl can hold rice, side dishes, and water.



- Variable creation and rules and features
  - Reserved words (keywords) cannot be used (for, if, else,...)
  - Spaces cannot be included
  - Only English letters and underscores (_) can be used as the first letter (number x)
  - Special characters other than underscores (_) cannot be used
  - Case sensitive

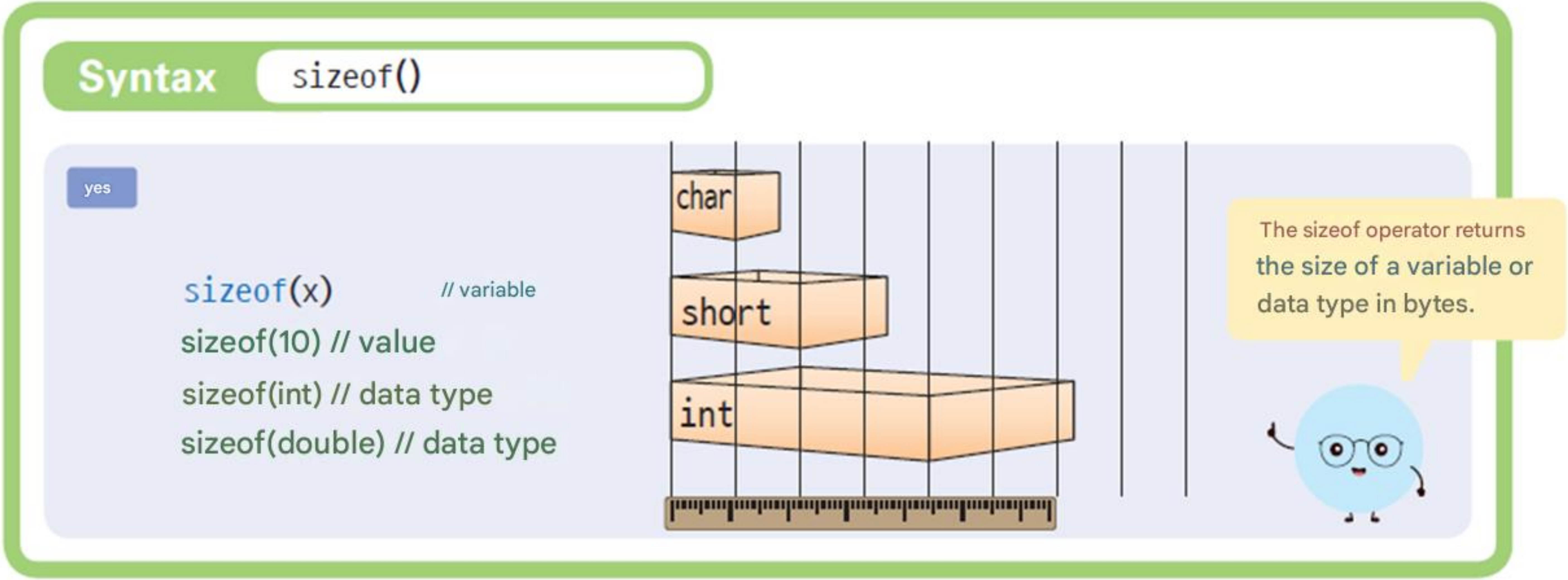
Variables & Data types

- Data types: To use memory space efficiently, data types of appropriate shape and size must be used.

Data Type	Description	Size (bytes)	Range	Example
int	Integer data type for whole numbers.	4	-2,147,483,648 to 2,147,483,647	int num = 10;
short	Short integer data type. Smaller range than int.	2	-32,768 to 32,767	short num = 100;
long	Long integer data type, typically used for larger numbers.	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long num = 1000000L;
long long	Extended long integer data type, used for even larger numbers.	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long long num = 1234567890123LL;
unsigned int	Unsigned integer, stores only positive values.	4	0 to 4,294,967,295	unsigned int num = 10U;
unsigned short	Unsigned short integer, stores only positive values.	2	0 to 65,535	unsigned short num = 100U;
unsigned long	Unsigned long integer, stores only positive values.	8	0 to 18,446,744,073,709,551,615	unsigned long num = 1000000UL;
unsigned long long	Unsigned long long integer, stores only positive values.	8	0 to 18,446,744,073,709,551,615	unsigned long long num = 1234567890123ULL;
char	Character data type, used to store single characters.	1	-128 to 127 (signed) or 0 to 255 (unsigned)	char letter = 'A';
unsigned char	Unsigned character, stores only positive character values (0 to 255).	1	0 to 255	unsigned char letter = 65U;
float	Single-precision floating point number.	4	±1.5 × 10^−45 to ±3.4 × 10^38	float num = 3.14f;
double	Double-precision floating point number, provides higher precision than float.	8	±5.0 × 10^−324 to ±1.7 × 10^308	double num = 3.141592;
long double	Extended precision floating point number (depends on the system).	10 or 16	Varies by system, typically ±3.4 × 10^−4932 to ±1.1 × 10^4932	long double num = 3.141592653589793;
_Bool	Boolean type (from C99 standard), stores true or false.	1	0 (false), 1 (true)	_Bool isTrue = 1;
void	Void type, used to indicate the absence of data or return type for functions.	N/A	N/A	void function() {}

Variable declaration

- Declaring variables: This is the task of creating space for variables. It can be initialized.



Syntax sizeof()

yes

```
sizeof(x)        // variable
sizeof(10) // value
sizeof(int) // data type
sizeof(double) // data type
```

The sizeof operator returns the size of a variable or data type in bytes.

Variable type

- local variables
- global variables
- static variables
- dynamic variables

```
#include <stdio.h>
void main() {
    int num = 1;
    if(1) {
       int num = 2;
       printf("%d\n", num);
    }
    printf("%d\n", num);
 }
```

Variable type

- local variables
- global variables
- static variables
- dynamic variables

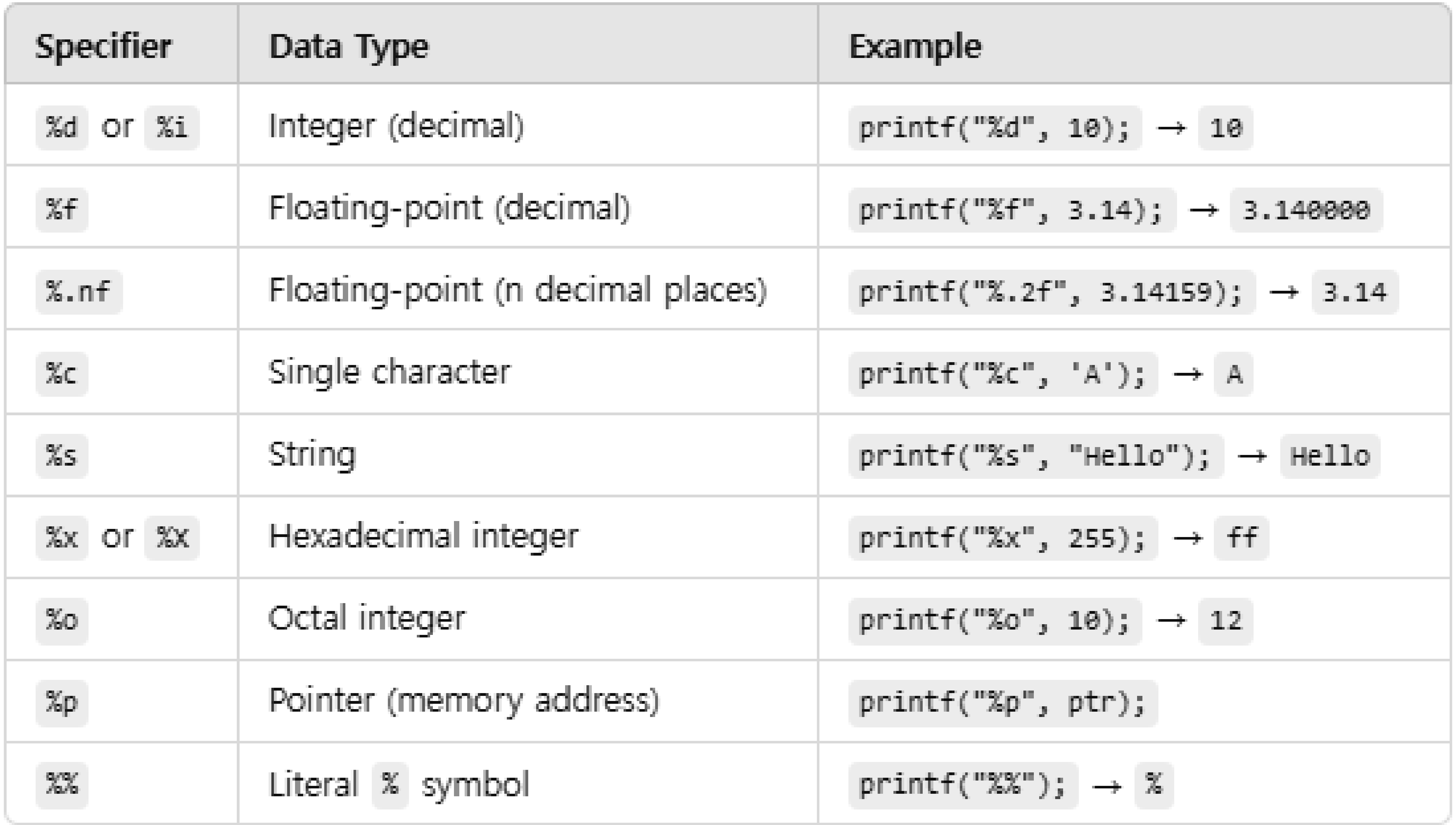
```
#include <stdio.h>

int num = 1;
void main() {
   printf("%d\n", num);
   num = 2;
   printf("%d\n", num);
   if(1) {
      num = 3;
      printf("%d\n", num);
   }
}
```

Standard Output

- Standard output sends data to an output device, usually the screen.
- It utilizes functions like printf(), puts(), and putchar()

Standard Output (printf)



Specifier	Data Type	Example
`%d` or `%i`	Integer (decimal)	`printf("%d", 10);` → `10`
`%f`	Floating-point (decimal)	`printf("%f", 3.14);` → `3.140000`
`%.nf`	Floating-point (n decimal places)	`printf("%.2f", 3.14159);` → `3.14`
`%c`	Single character	`printf("%c", 'A');` → `A`
`%s`	String	`printf("%s", "Hello");` → `Hello`
`%x` or `%X`	Hexadecimal integer	`printf("%x", 255);` → `ff`
`%o`	Octal integer	`printf("%o", 10);` → `12`
`%p`	Pointer (memory address)	`printf("%p", ptr);`
`%%`	Literal `%` symbol	`printf("%%");` → `%`

o.h>
: char *format, ...);

e format specifier

Standard Output (printf)

3. Format of printf()

- Width and Alignment

  You can specify a **minimum width** for the output using numbers.

  Default is right-aligned; to left-align, use -

```
printf("%10d\n", 123);  // Right-aligned, width 10
printf("%-10d\n", 123); // Left-aligned, width 10
```

Practice

- Precision for Floating-Point Numbers

```
printf("%.2f\n", 3.14159);  // Prints with 2 decimal places
```

Practice

- Padding with Zeros

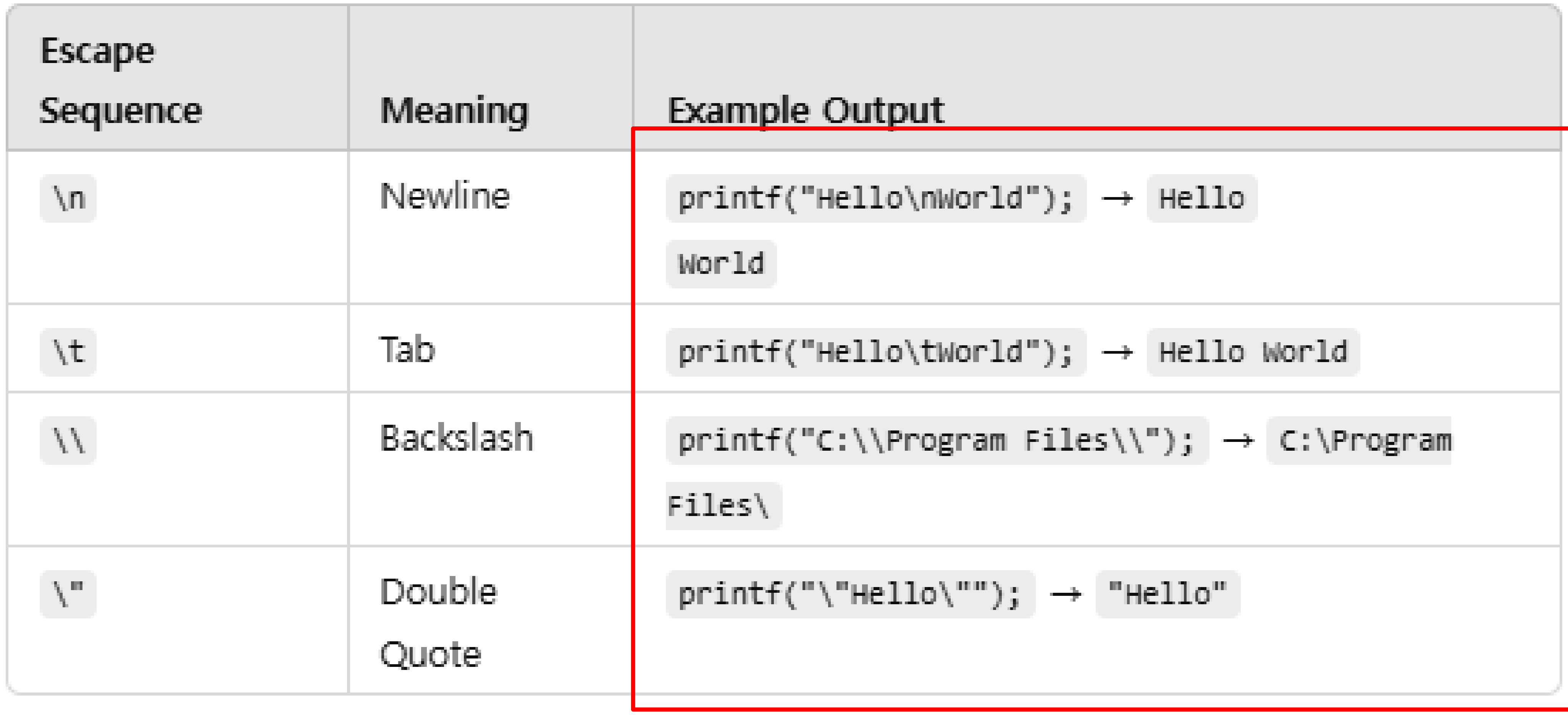
```
printf("%05d\n", 42);  // Pads with zeros up to 5 digits
```

Practice

Standard Output (printf)

4. Using Escape Sequences of printf()

- printf supports escape sequences to control output formatting



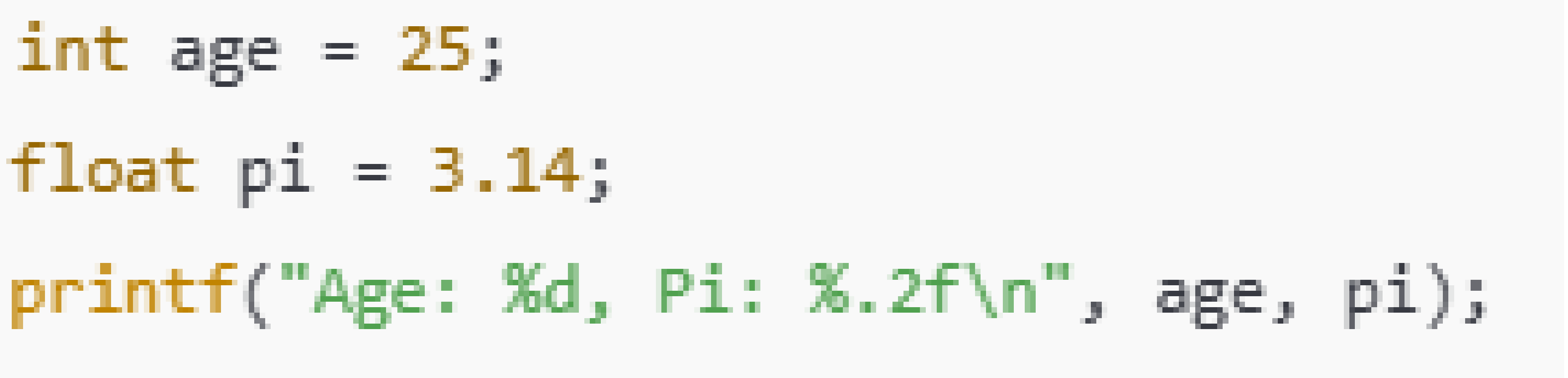
Escape Sequence	Meaning	Example Output
\n	Newline	printf("Hello\nWorld"); → Hello World
\t	Tab	printf("Hello\tWorld"); → Hello World
\\	Backslash	printf("C:\\Program Files\\"); → C:\Program Files\
\"	Double Quote	printf("\"Hello\""); → "Hello"

Practice

Standard Output (printf)

5. Printing Multiple Values

- You can print multiple values in a single printf call by passing multiple

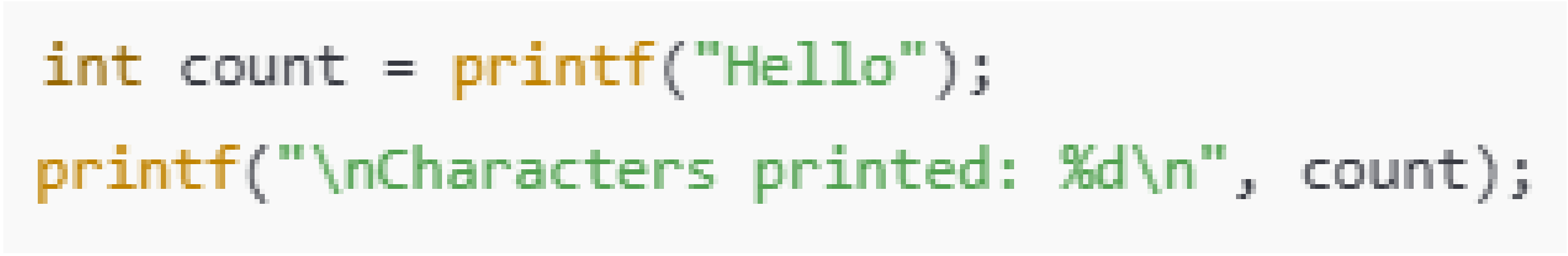


```
int age = 25;
float pi = 3.14;
printf("Age: %d, Pi: %.2f\n", age, pi);
```

Practice

5. Return Value of printf

-printf returns the number of characters printed (excluding \0)



```
int count = printf("Hello");
printf("\nCharacters printed: %d\n", count);
```

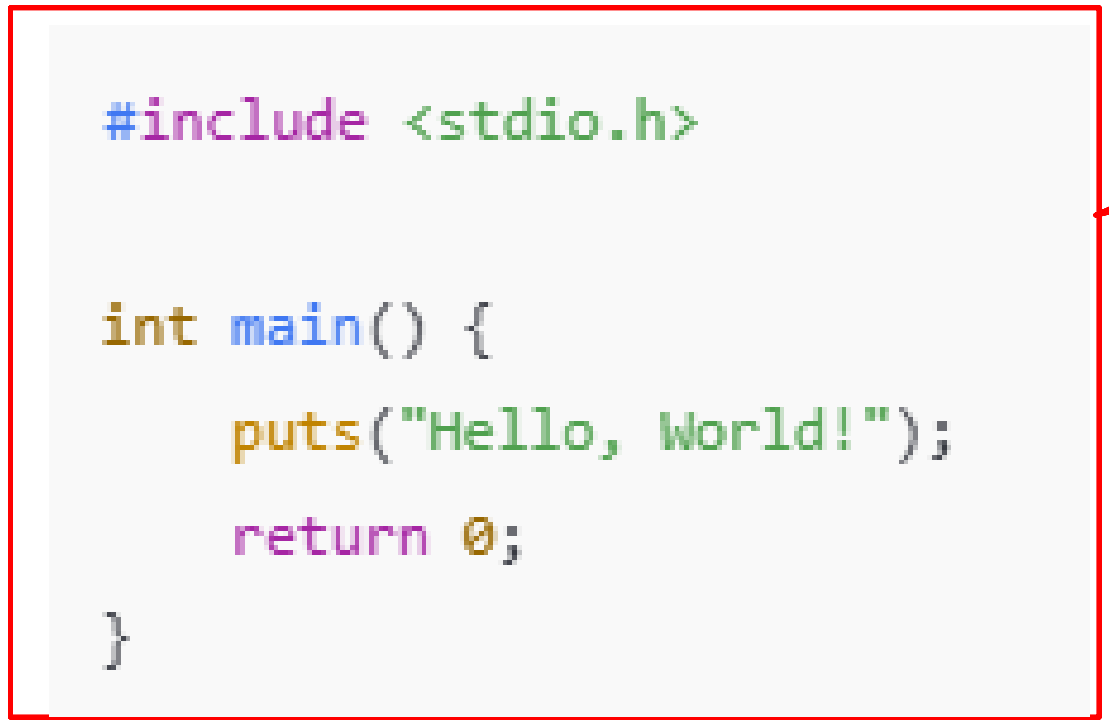
Practice

Standard Output (puts)

1. Syntax of **puts**()

```
#include <stdio.h>
int puts(const char *str);
```

- puts prints a string (str) to the console and automatically appends a newline (\n) at the end.
- It is simpler and safer than printf("%s\n", str); because it does not require format specifiers.
- It returns a non-negative integer on success and EOF (-1) on failure.



```c
#include <stdio.h>

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

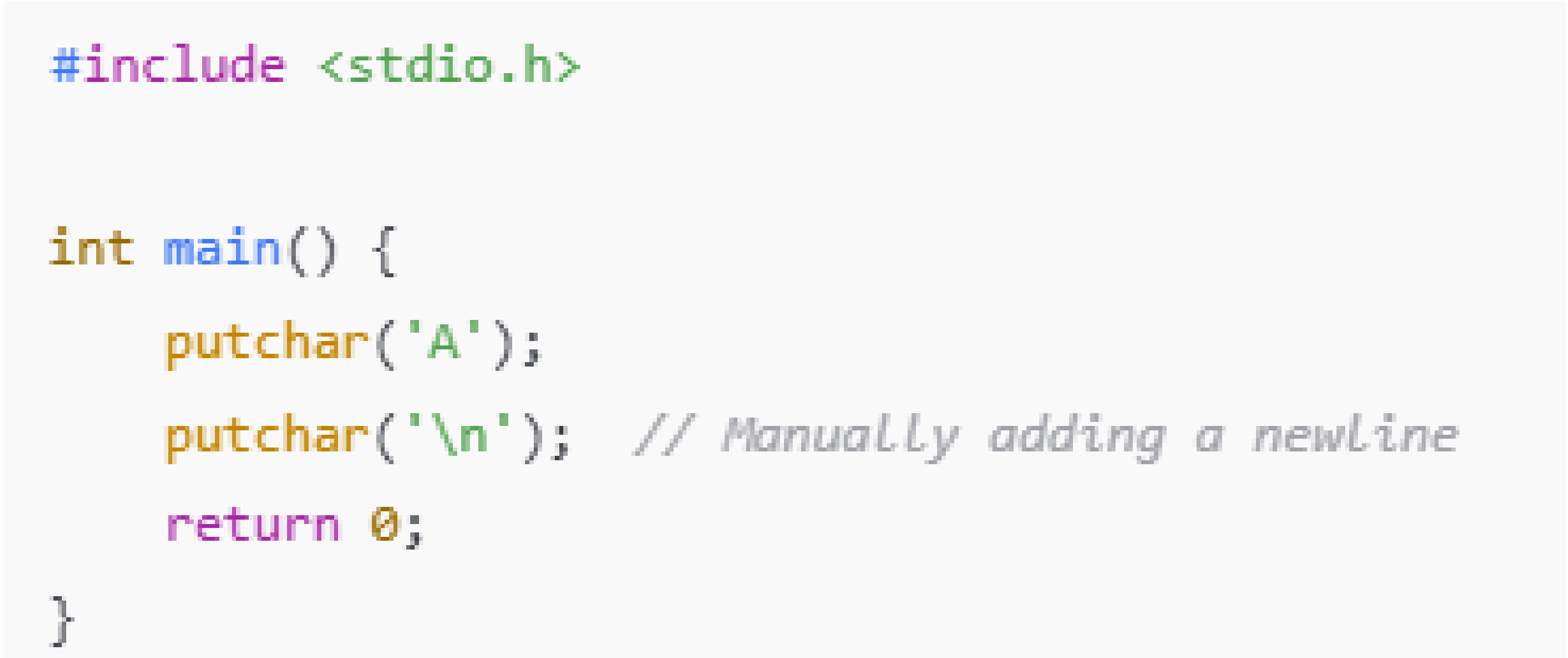
Practice

Standard Output (putchar)

1. Syntax of **putchar**()

```
#include <stdio.h>
int putchar(int ch);
```

- putchar prints a single character (ch) to the console.
- It is simpler and safer than printf("%s\n", str); because it does not require format specifiers.
- It returns a non-negative integer on success and EOF (-1) on failure.



```c
#include <stdio.h>

int main() {
    putchar('A');
    putchar('\n');  // Manually adding a newline
    return 0;
}
```

Practice

Standard Input

Standard input reads data from an input device, typically the keyboard.

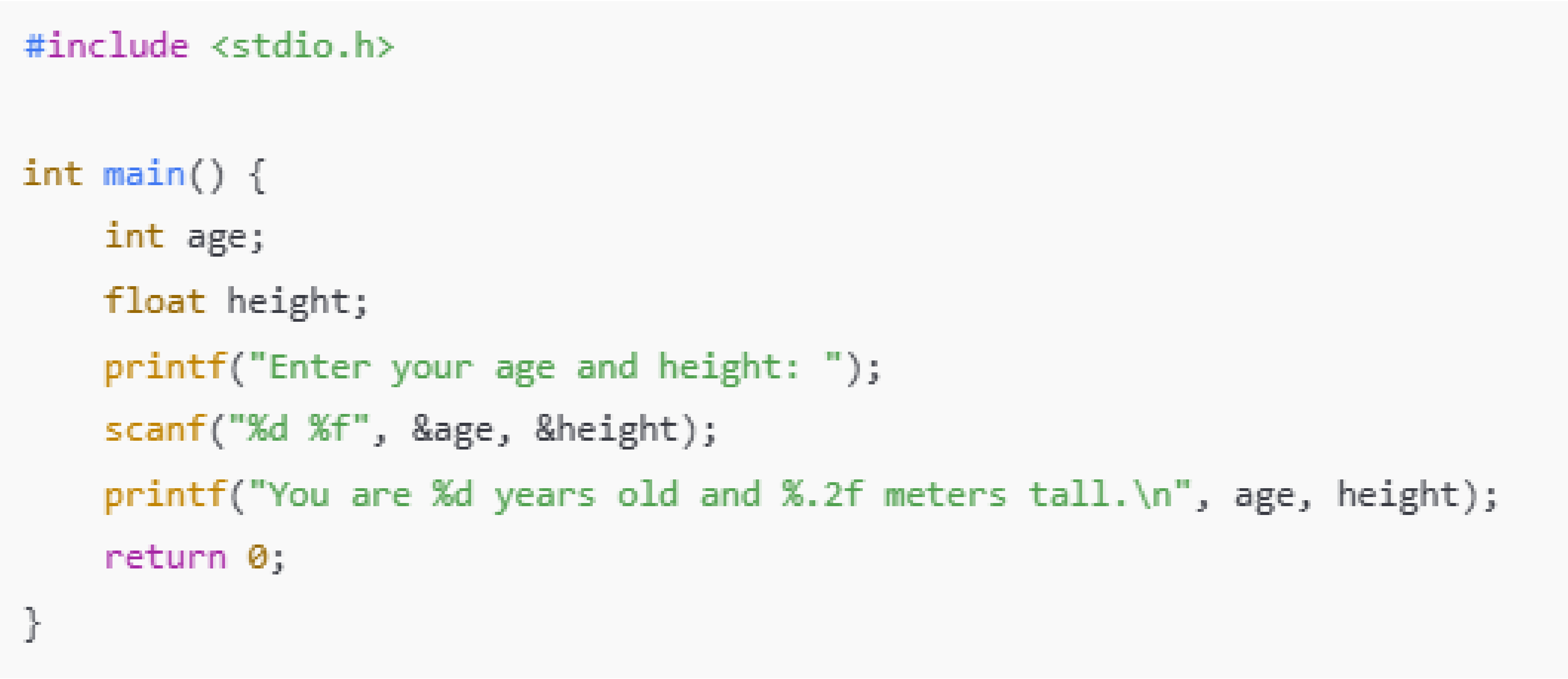
It uses functions like scanf(), getchar(), and fgets() to read user input

Standard Input (scanf)

1. Syntax of **scanf**()

```
#include <stdio.h>
int scanf(const char *format, ...);
```

- scanf reads formatted input from stdin (usually the keyboard).
- It requires format specifiers to determine the type of input.
- It stops reading when encountering whitespace (spaces, tabs, newlines, etc.).



```c
#include <stdio.h>

int main() {
    int age;
    float height;
    printf("Enter your age and height: ");
    scanf("%d %f", &age, &height);
    printf("You are %d years old and %.2f meters tall.\n", age, height);
    return 0;
}
```

Practice

Standard Input (scanf)

2. Key Characteristics of **scanf**()

- Can read multiple values at once.
- Requires the address-of operator (&) for non-string variables.
- Stops reading at the first whitespace character (space, tab, or newline).
- Can cause buffer issues if not used carefully (e.g., failing to handle newline characters properly).

Standard Input (getchar)

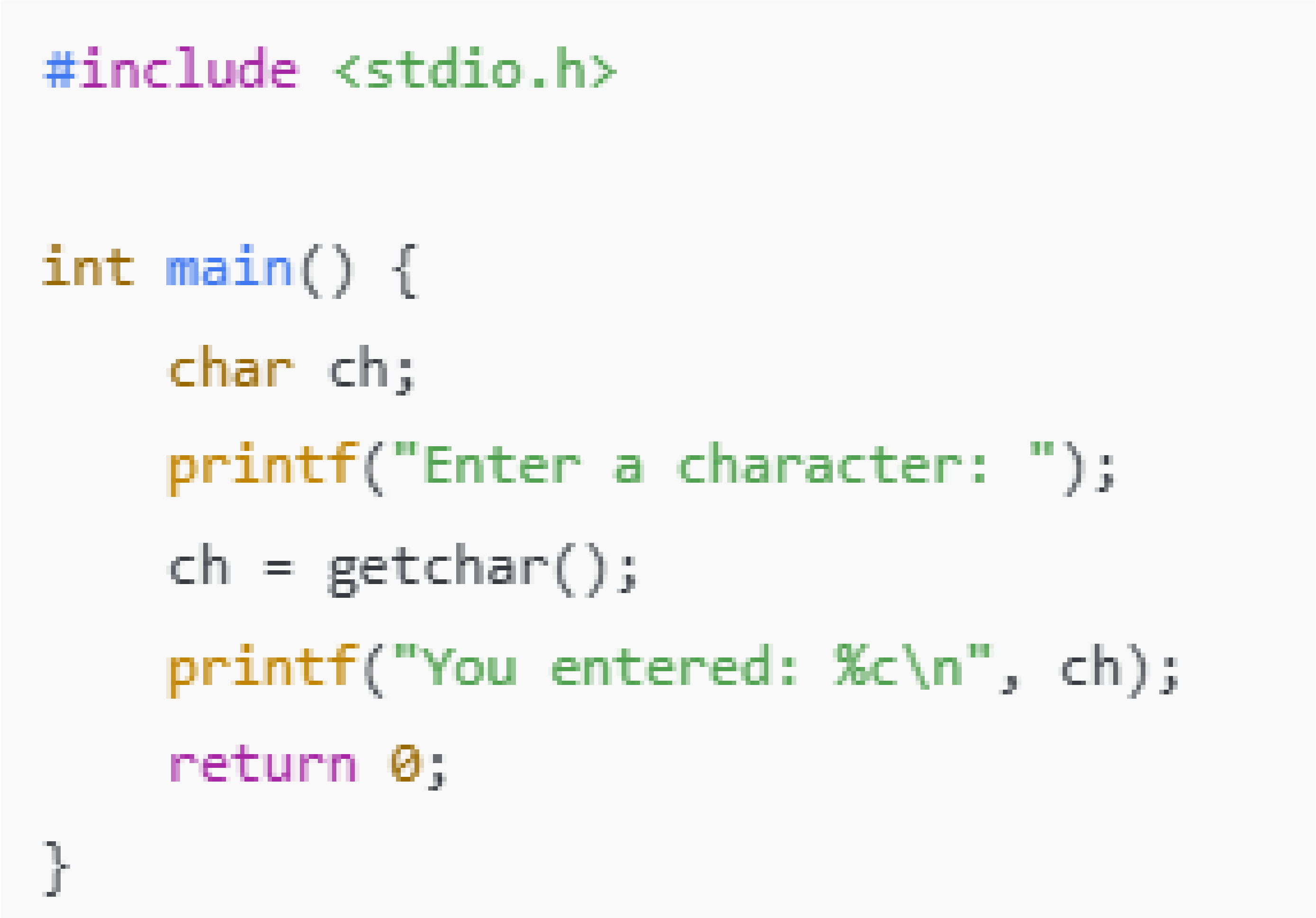
1. Syntax of **getchar**()

#include <stdio.h>
int getchar(void);

- getchar reads a single character from stdin.

- It includes whitespace characters like spaces and newlines.

- Returns the character as an unsigned char (cast to int) or EOF on error.



```c
#include <stdio.h>

int main() {
    char ch;
    printf("Enter a character: ");
    ch = getchar();
    printf("You entered: %c\n", ch);
    return 0;
}
```

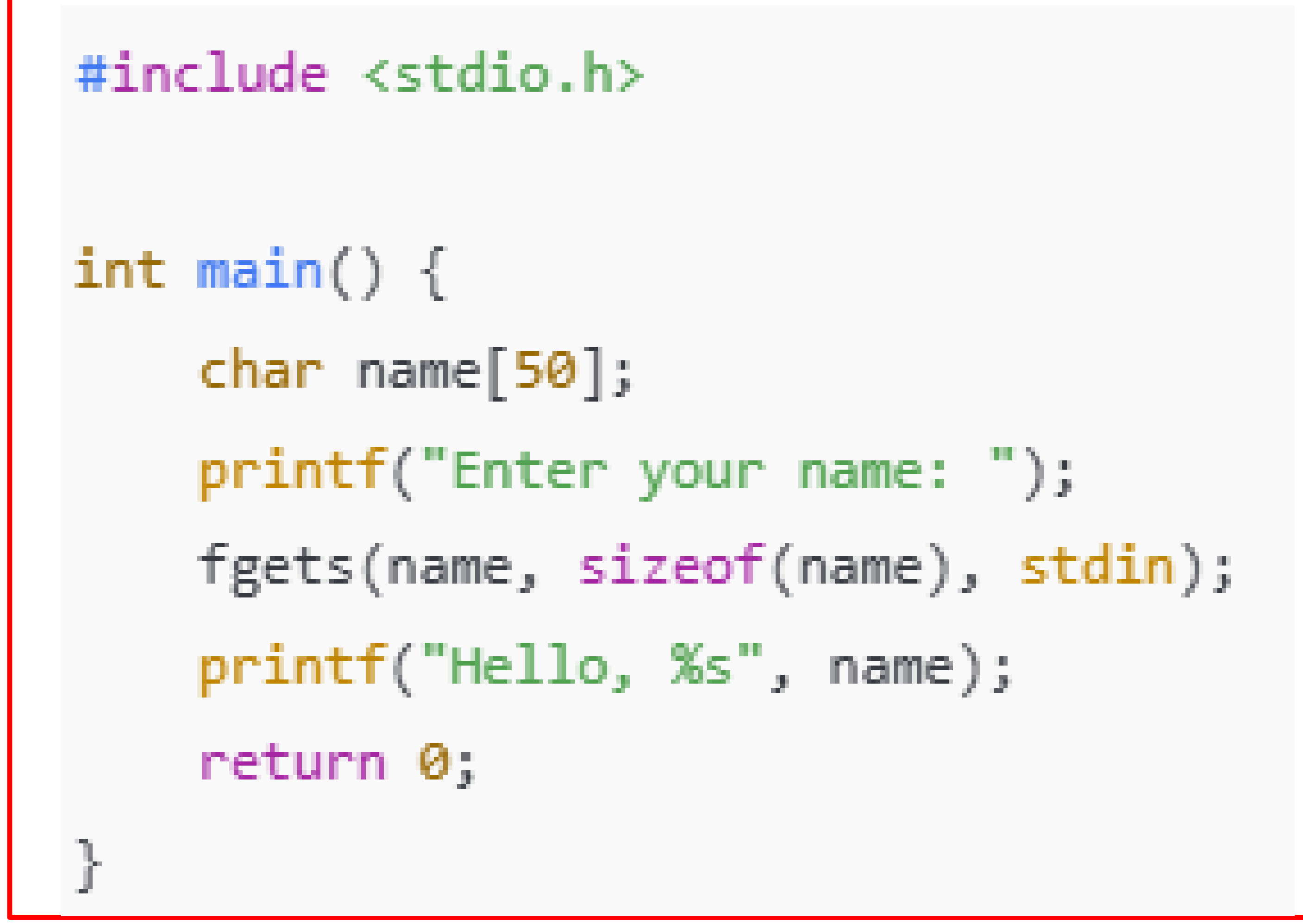
Practice

Standard Input (fgets)

1. Syntax of **fgets**()

```
#include <stdio.h>
char *fgets(char *str, int n, FILE *stream);
```

- fgets reads a whole line from the input (up to n-1 characters).
- It includes spaces and stops at a newline (\n).
- It prevents buffer overflow by specifying the maximum number of characters.



```
#include <stdio.h>

int main() {
    char name[50];
    printf("Enter your name: ");
    fgets(name, sizeof(name), stdin);
    printf("Hello, %s", name);
    return 0;
}
```

Practice

Standard Input (fgets)

2. Key Characteristics of **fgets**()

- Reads a full line, including spaces.

- Stops when newline (\n) or buffer limit (n-1 characters) is reached.

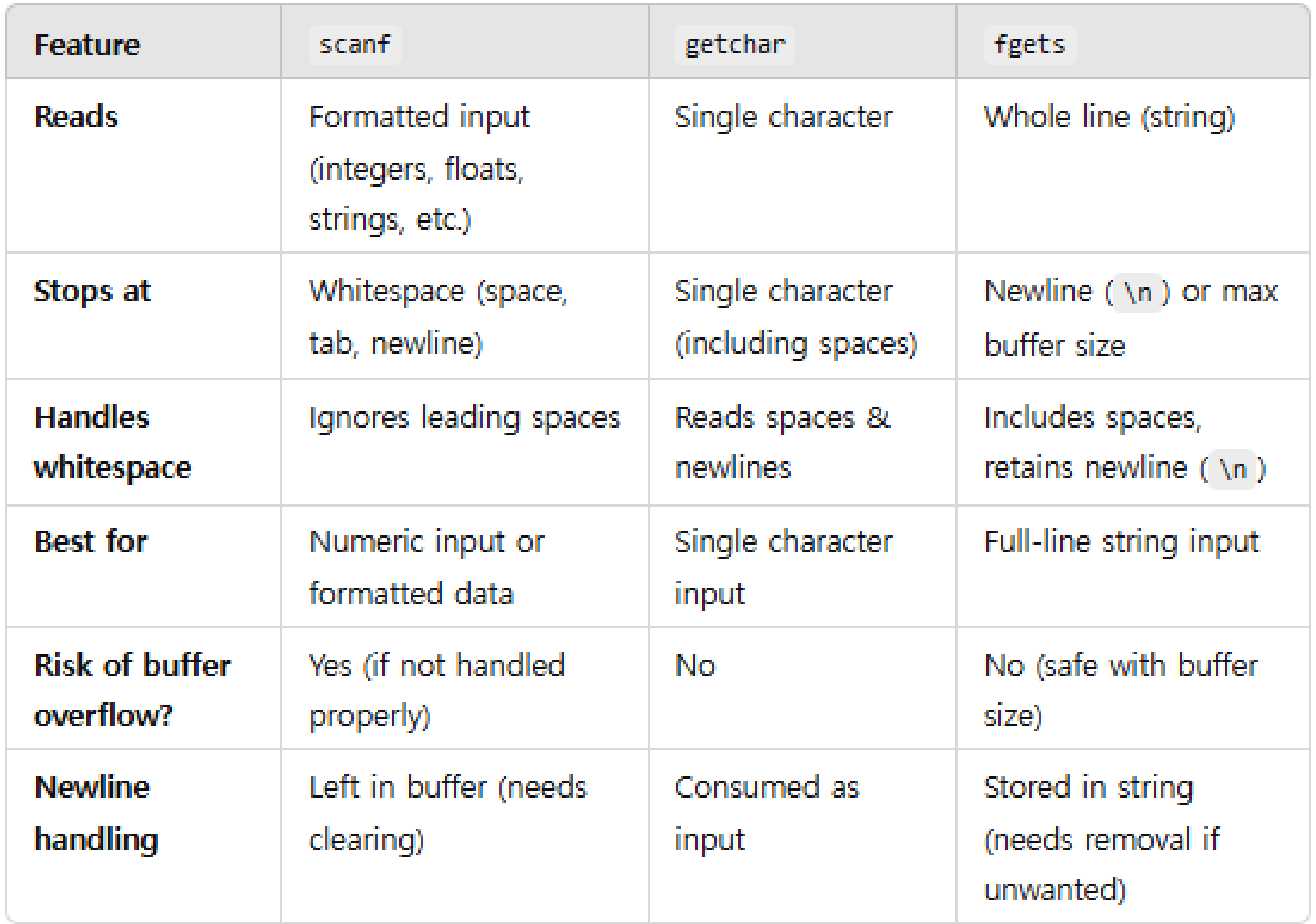
- Unlike scanf, it does not skip spaces.

- Adds a newline character (\n) if the user presses Enter.

Standard Input

Comparison of scanf, getchar, fgets

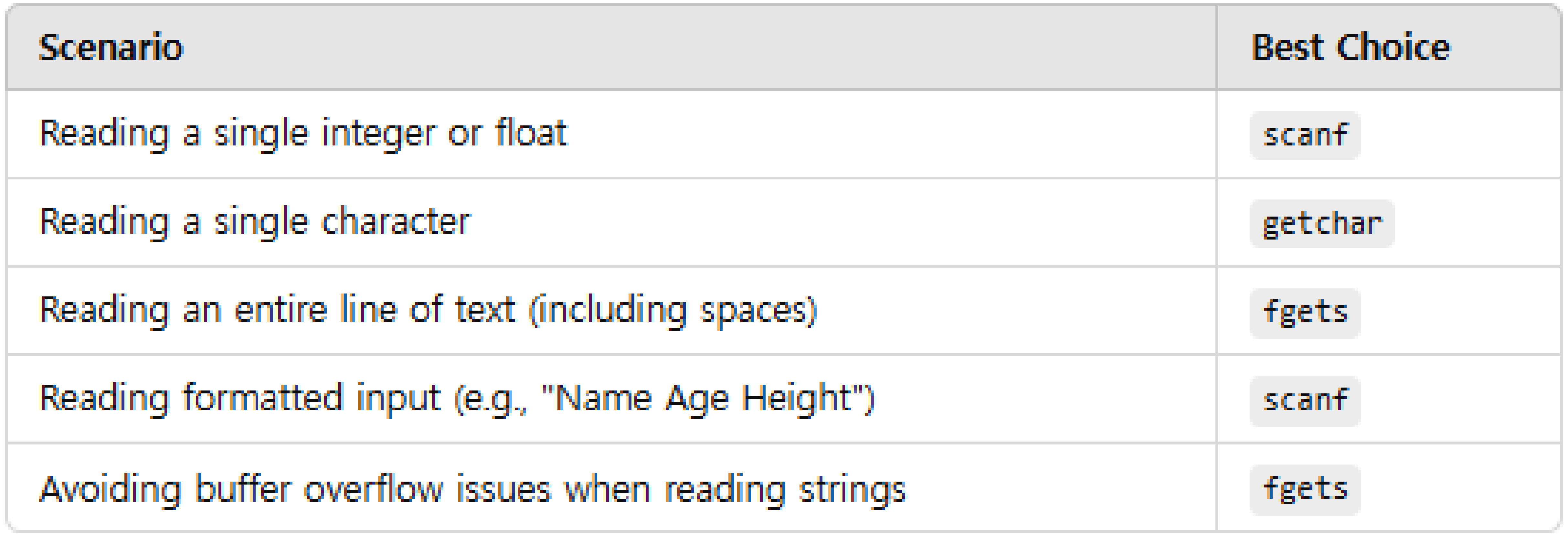
-



Feature	`scanf`	`getchar`	`fgets`
Reads	Formatted input (integers, floats, strings, etc.)	Single character	Whole line (string)
Stops at	Whitespace (space, tab, newline)	Single character (including spaces)	Newline (`\n`) or max buffer size
Handles whitespace	Ignores leading spaces	Reads spaces & newlines	Includes spaces, retains newline (`\n`)
Best for	Numeric input or formatted data	Single character input	Full-line string input
Risk of buffer overflow?	Yes (if not handled properly)	No	No (safe with buffer size)
Newline handling	Left in buffer (needs clearing)	Consumed as input	Stored in string (needs removal if unwanted)

Standard Input

When to use which?

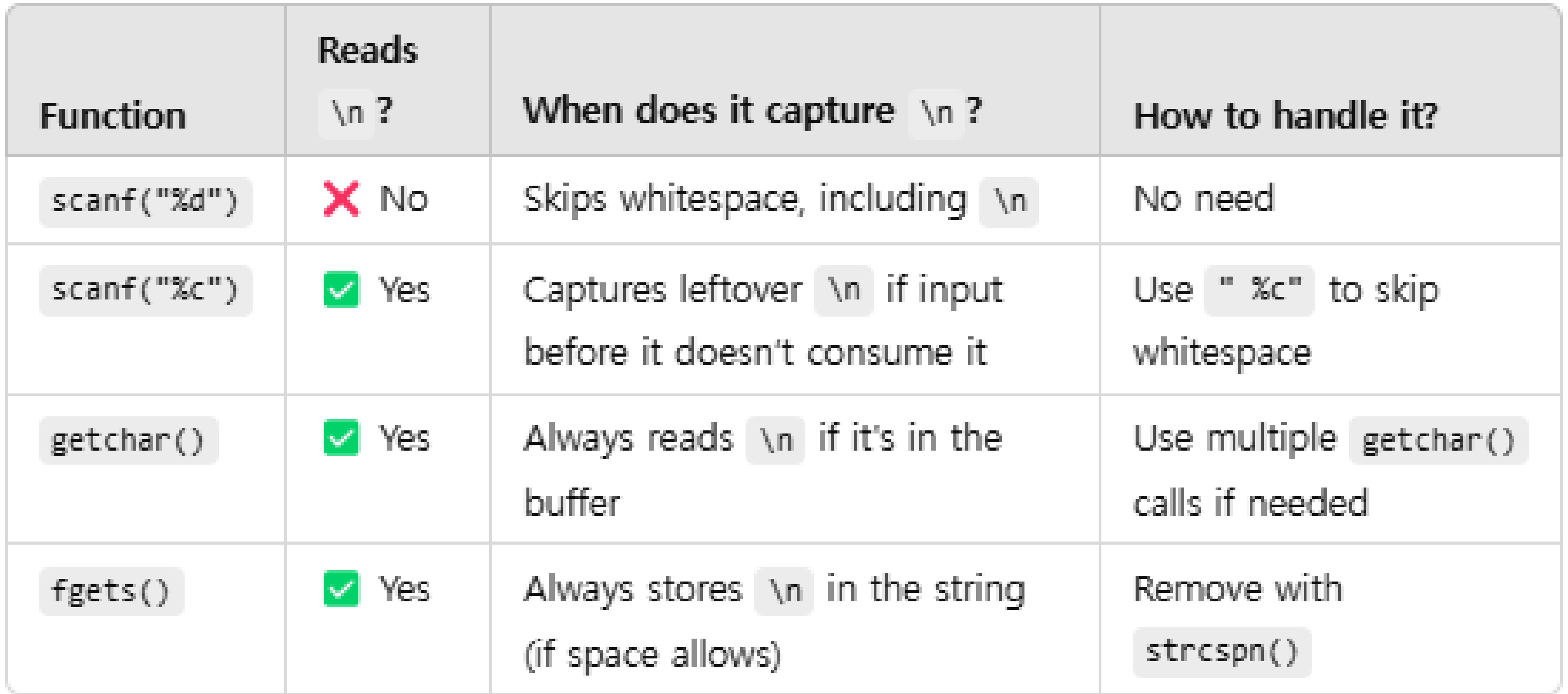


Scenario	Best Choice
Reading a single integer or float	`scanf`
Reading a single character	`getchar`
Reading an entire line of text (including spaces)	`fgets`
Reading formatted input (e.g., "Name Age Height")	`scanf`
Avoiding buffer overflow issues when reading strings	`fgets`

Standard Input

How each function handles Enter (\n)?

-



Function	Reads `\n` ?	When does it capture `\n` ?	How to handle it?
`scanf("%d")`	❌ No	Skips whitespace, including `\n`	No need
`scanf("%c")`	✅ Yes	Captures leftover `\n` if input before it doesn't consume it	Use `" %c"` to skip whitespace
`getchar()`	✅ Yes	Always reads `\n` if it's in the buffer	Use multiple `getchar()` calls if needed
`fgets()`	✅ Yes	Always stores `\n` in the string (if space allows)	Remove with `strcspn()`

Practice in the class

1. Make requirement list of C1-P1
2. Implement C1-P1
3. Upload it to a repo of your github
4. Ask your friends to evaluate your implementation
5. Explain your implementation to your friend
6. Switching roles, discuss about your friend's implementation

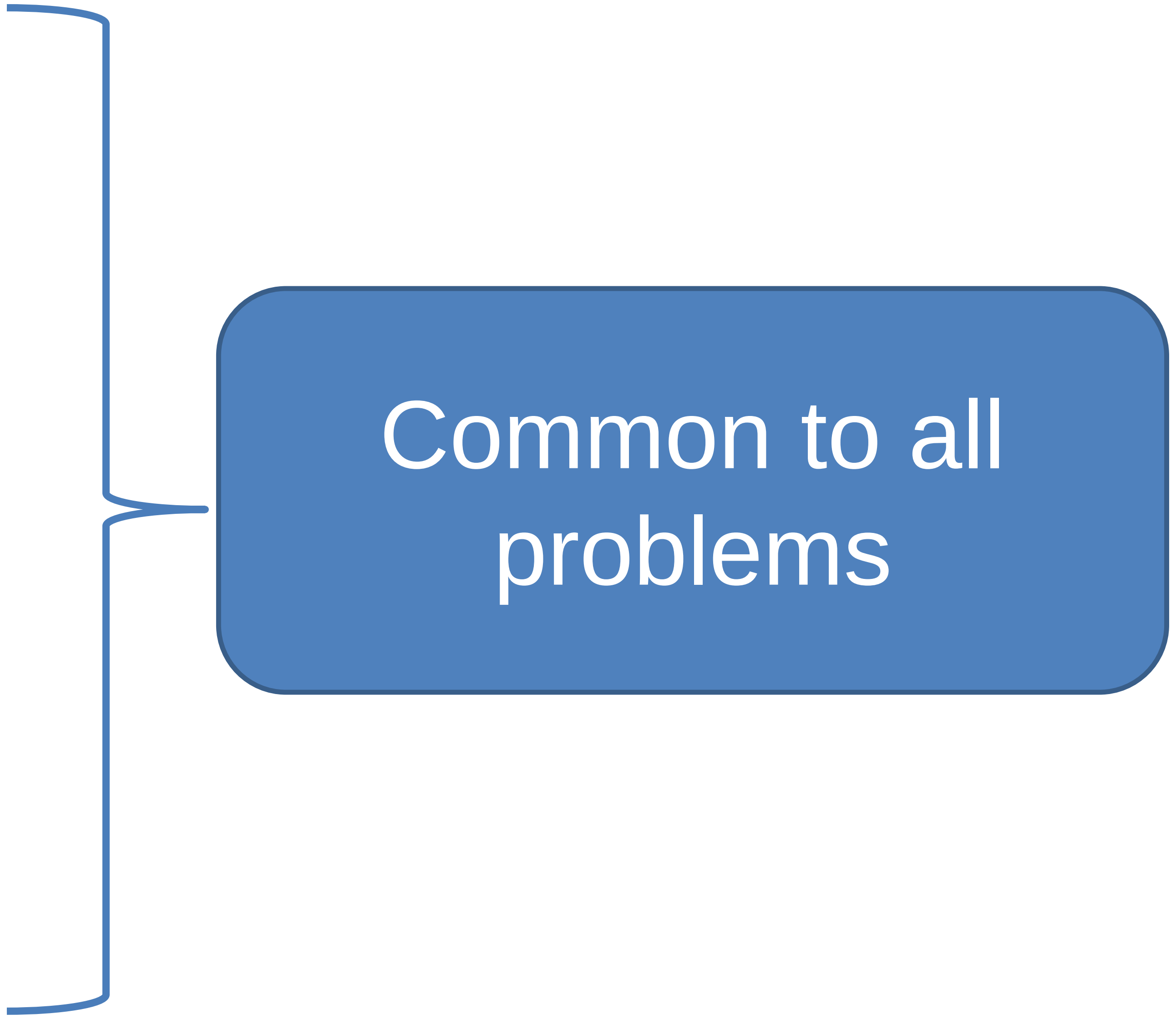
Requirements list

1. Development Environment & Standards

- Use a cross-platform development tool (e.g., VS Code, CLion, Code::Blocks).
- Follow the ANSI C standard syntax.
- Use GCC 9.x compiler.
- Only use the C standard library (stdio.h, stdlib.h, etc.).
- Follow ANSI C coding style (proper indentation, comments, and readable code).

2. Project Structure

- The project folder must be named Magrathea.
- All source code must include comments for clarity.
- The source code must compile successfully without errors or warnings.
- Code must not contain unnecessary or unrelated parts.
- All logic must be implemented within main() (no functions outside main()



Common to all problems

3. Problem specific requirements

- Print Arthur and the team members' basic information to the console, following the specified format:
- Each sentence in the "Introduction" section should break onto a new line after a period (.).

Homework

1. Read Write down “requirement list” of C1-P2

   (Put it in your source code with comments)

2. Implement C1-P2

3. Upload it into a repo of your github

Requirements list

1. User Input

- Prompt the user to enter the **current date** in the "yyyy-mm-dd" format.
- Prompt the user to enter their **name**.

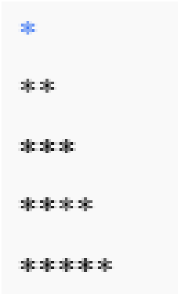
2. Processing the Input

- Display the message **“The input has been processed successfully.”**
- Ensure the entered values (name and date) are incorporated into the splash screen output.

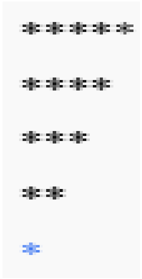
3. Splash Screen Output

- After processing, display the following splash screen format

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[Magrathea ver 0.1]
Magrathea, where a shining planet is created in a wasteland with no grass,
a place where unseen potential is discovered and gems are polished by the hands of experts,
Welcome to Magrathea.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[User]: [name]                              [Execution Time]: [date]
============================================================================================================
```



```
*
**
***
****
*****
```



```
*****
****
***
**
*
```

4. Bonus 1: Delay Before Display

- After the input has been processed, **clear the screen after 3 seconds** and then display the splash screen.
- Display a right-angled triangle and an inverted right-angled triangle made of * characters on the left and right edges of the splash screen.

See you next week!
DO NOT miss the classes

Debugging

- Practice with debugger
- VSC (launch.json, tasks.json)

Hello world

```
#include <stdio.h>

int main() {
   printf("Hello, World!\n");
   return 0;
}
```

```
print("Hello, World!")
```

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

```
#include <iostream>

int main() {
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```