C Programming (W4)

Welcome!!

Please check attendance individually.

(Mobile App)

Things to do today

01	Ch.1 ~ Ch.3
02	Debugging in VSC
03	Variables & Data types
04	Standard Input & Output

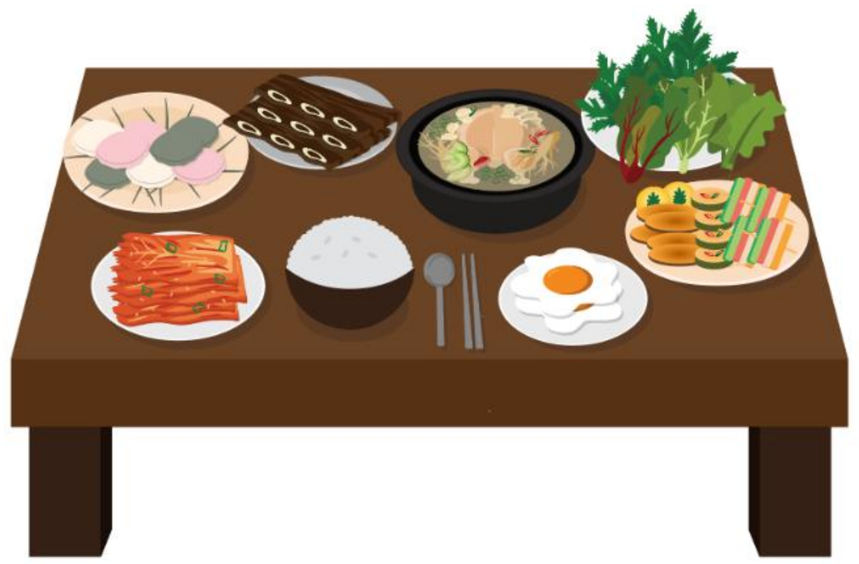
Debugging

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

  - github.com/prof-kweon/2025-Fall-C-Language/blob/main/Reference/lauch.json
  - github.com/prof-kweon/2025-Fall-C-Language/blob/main/Reference/tasks.json

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.

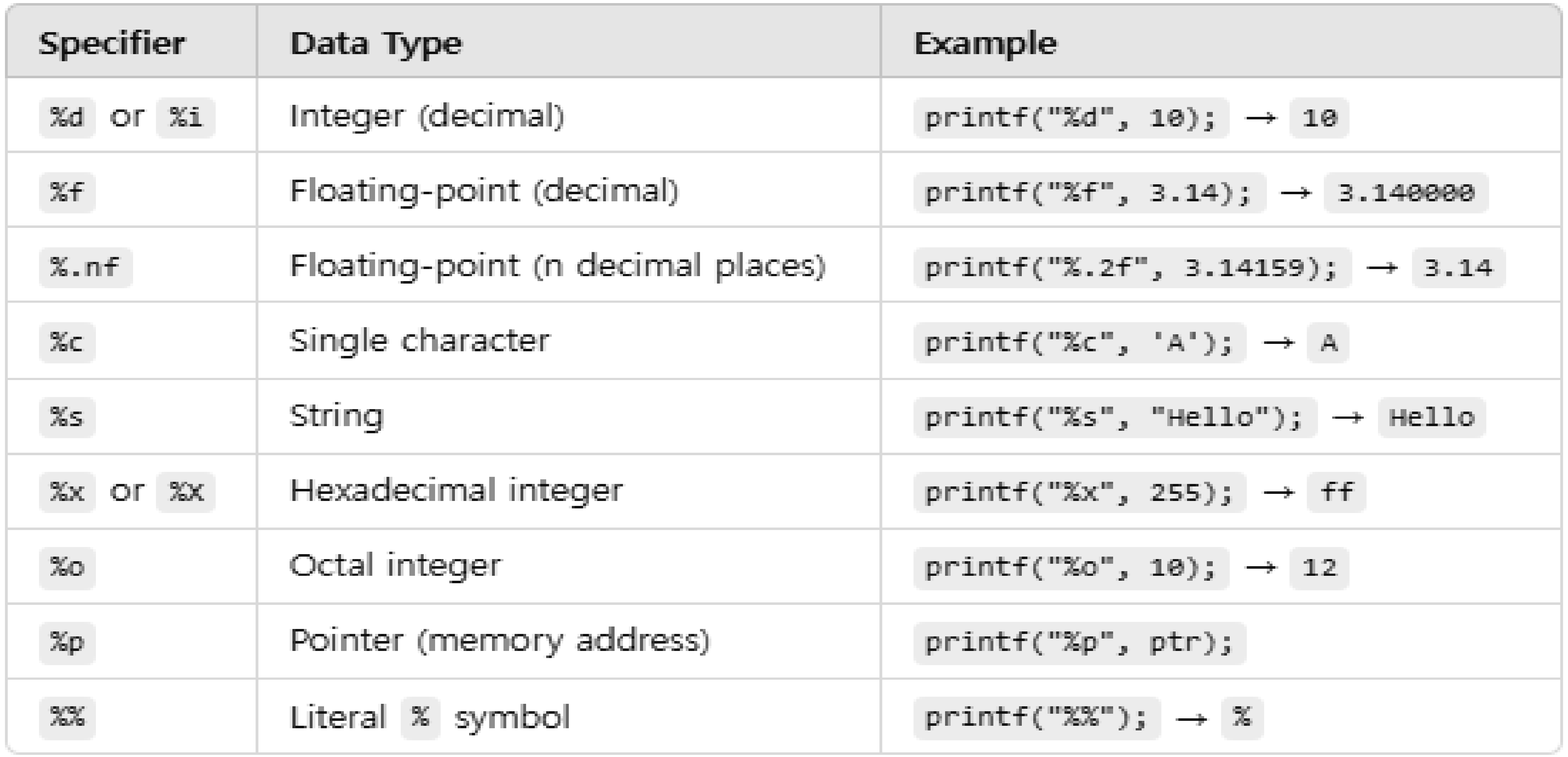
Data Type	Description	Size (bytes)	Range	Example
int	Integer data type for whole numbers.	4	-2,147,483,648 to 2,147,483,647	int num = 10;
short	Short integer data type. Smaller range than int.	2	-32,768 to 32,767	short num = 100;
long	Long integer data type, typically used for larger numbers.	8	-9,223,372,036,854,775,808 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() {}

Standard Output (printf)

1. Syntax of **printf**() : Grammar and Structure of language

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

- The first argument (format) is a string containing text and format specifiers
- The ellipsis (...) represents a variable number of arguments, which are inserted into the format specifier



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("%%");` → `%`

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
```

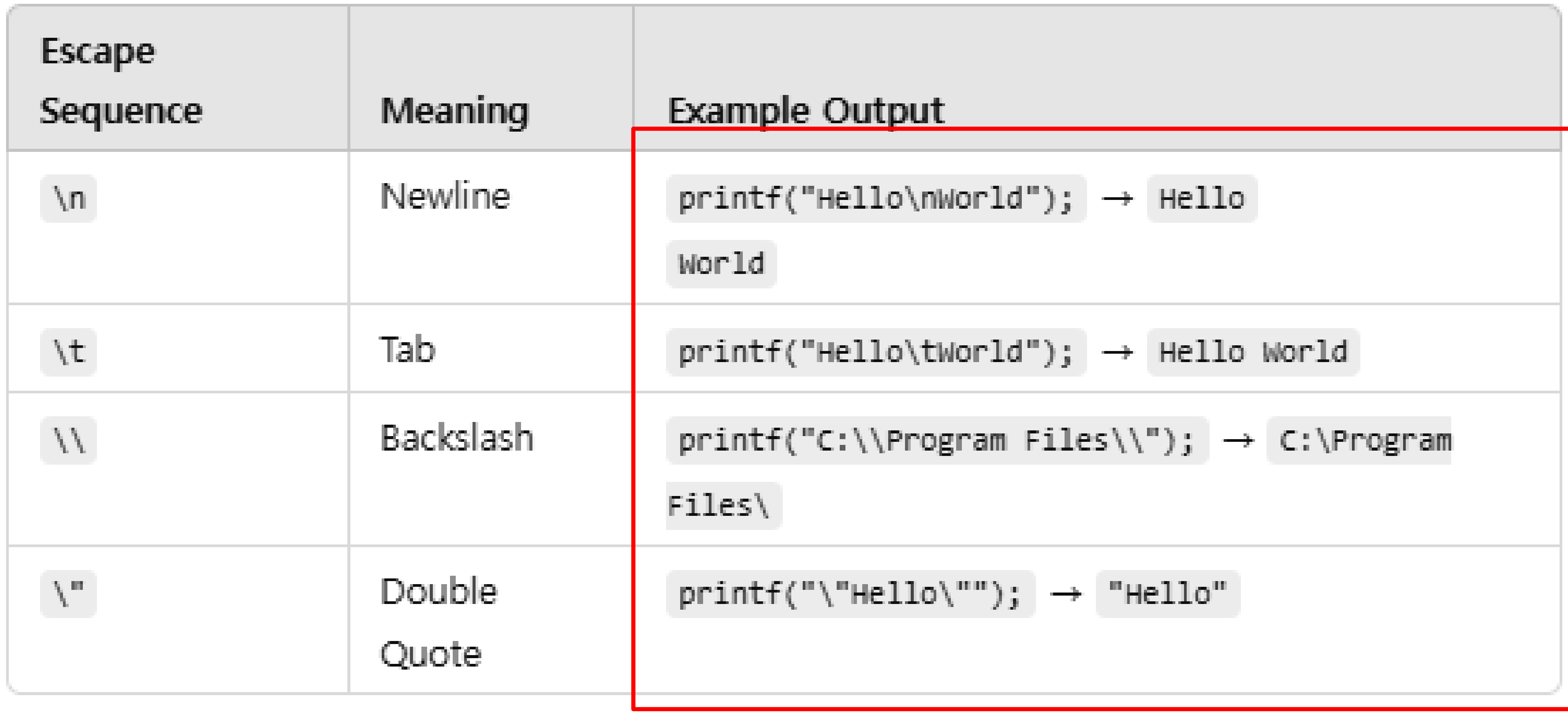
Practice

Standard Output (printf)

AI & Big Data

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"`

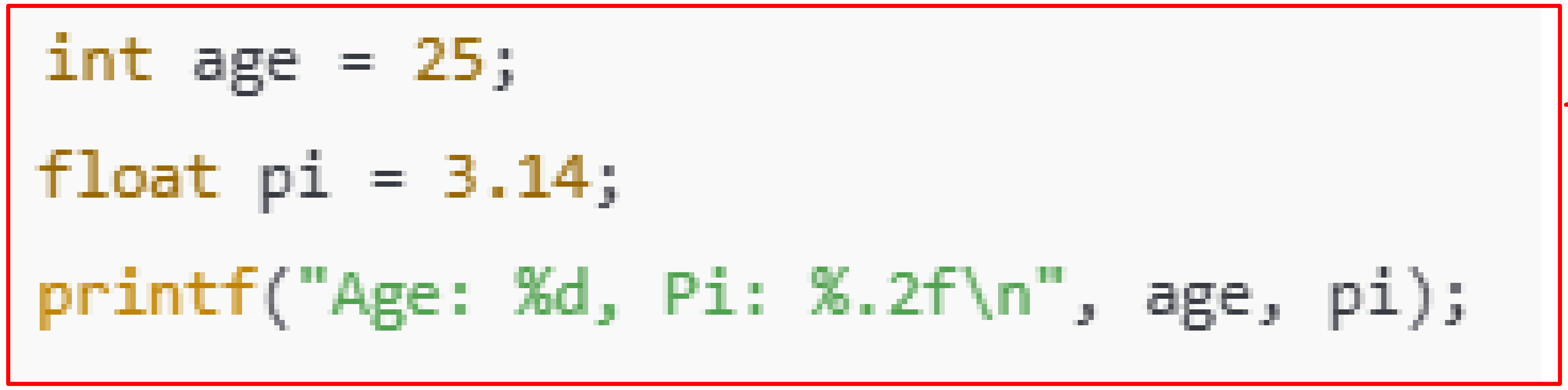
Practice

Standard Output (printf)

5. Printing Multiple Values

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

Practice

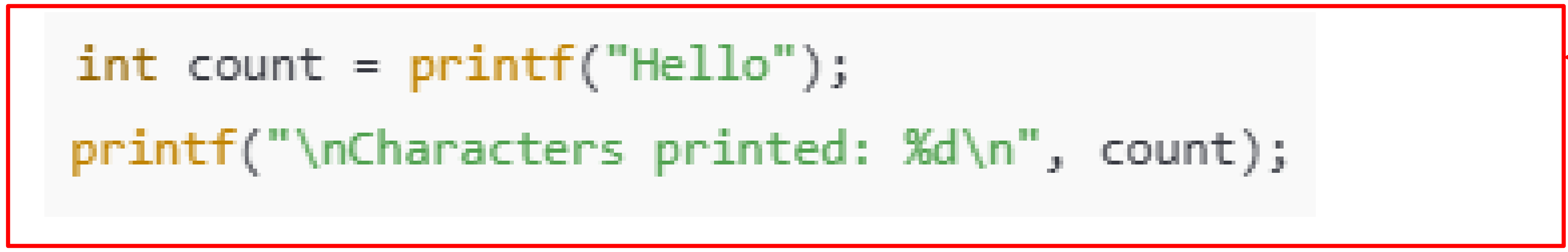


```c
int age = 25;
float pi = 3.14;
printf("Age: %d, Pi: %.2f\n", age, pi);
```

5. Return Value of printf

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

Practice



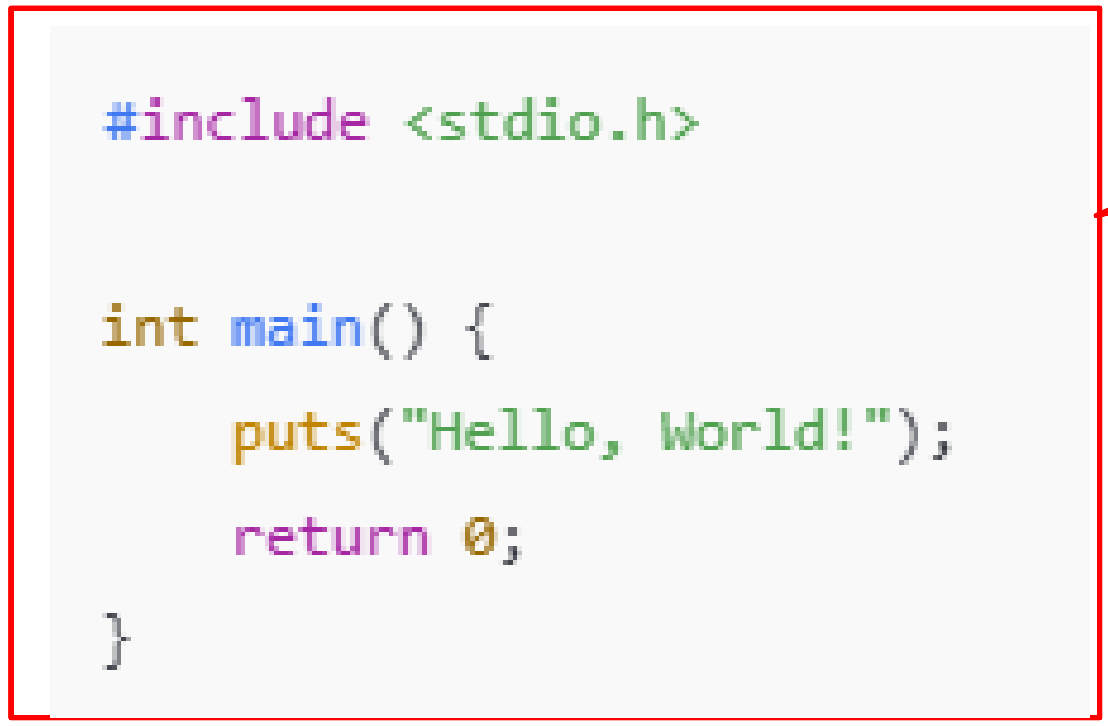
```c
int count = printf("Hello");
printf("\nCharacters printed: %d\n", count);
```

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;
}
```

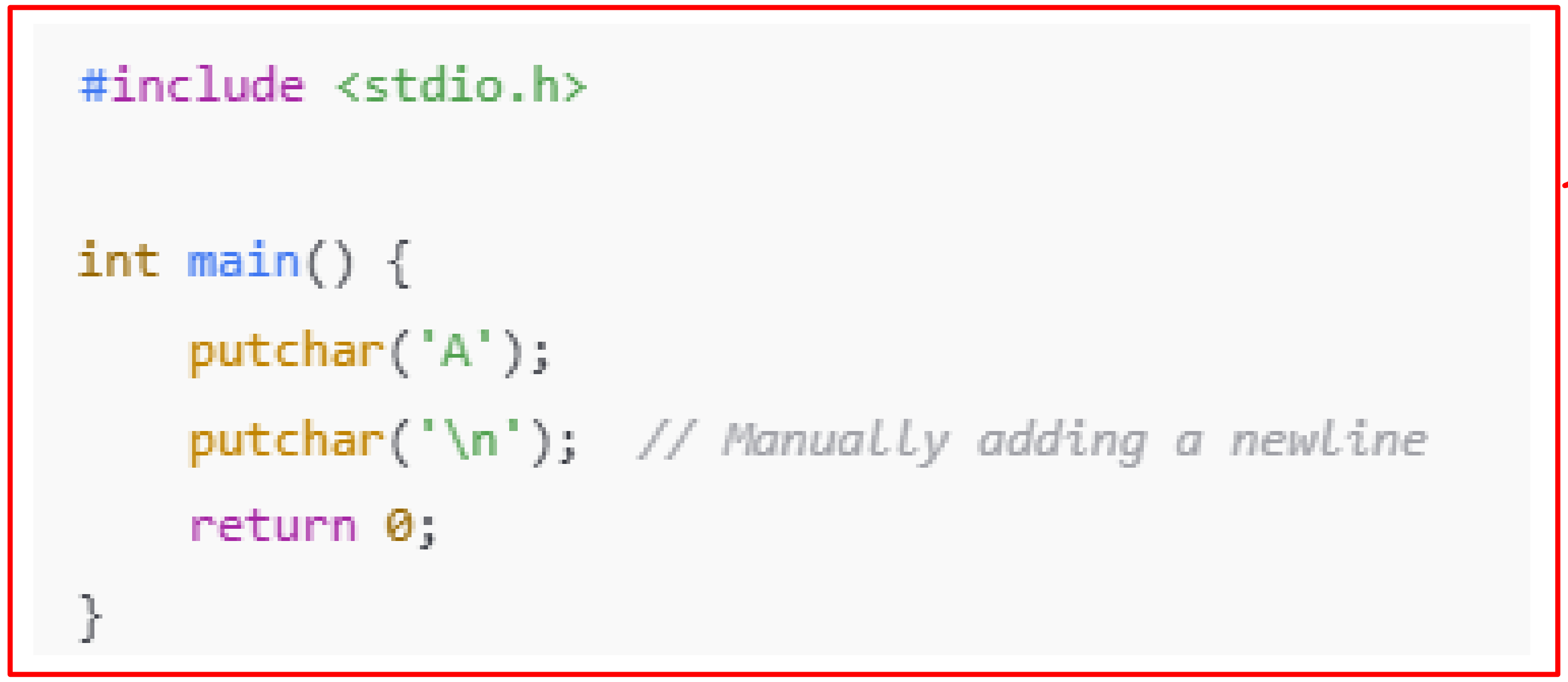
Practice

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

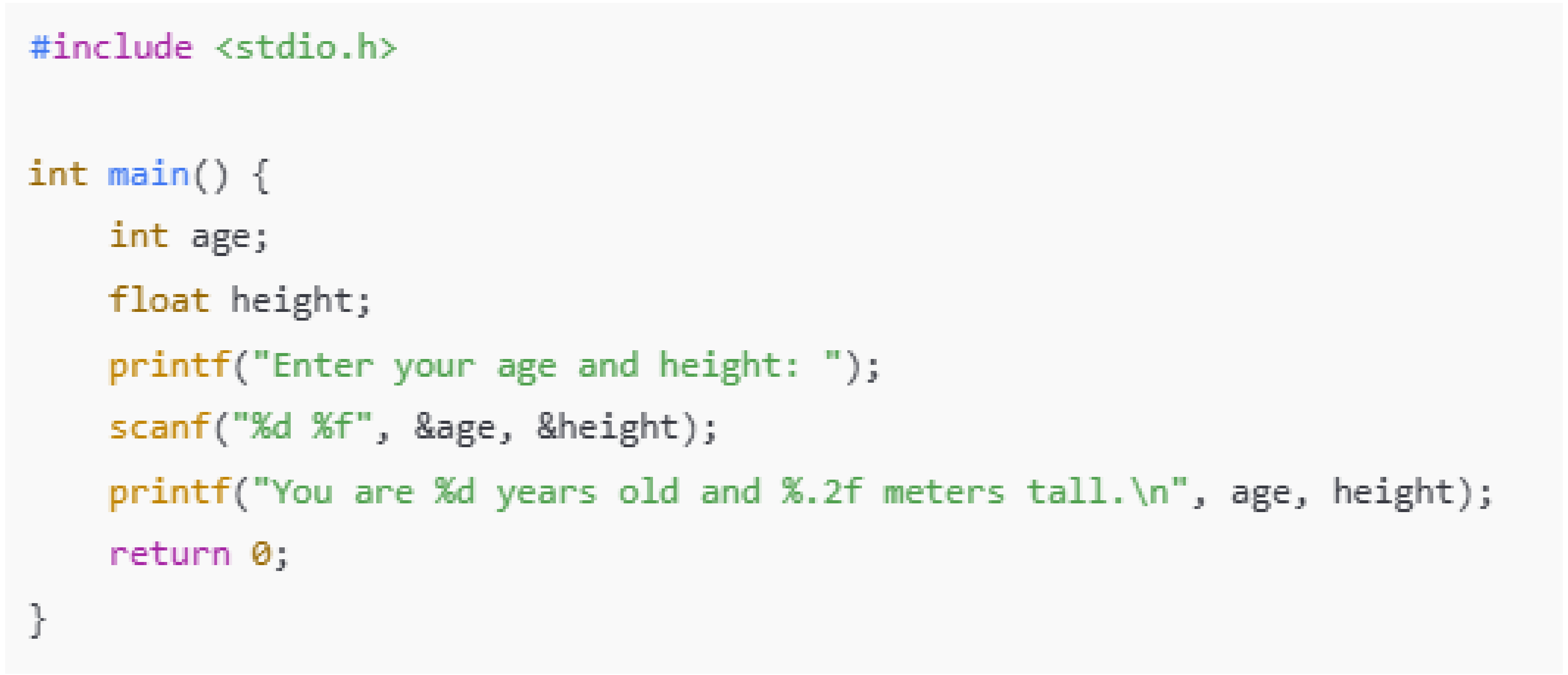
Standard Input (scanf)

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.).
- it remains '\n' in buffer. To avoid → getchar();

Practice



```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;
}
```

Standard Input (scanf)

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

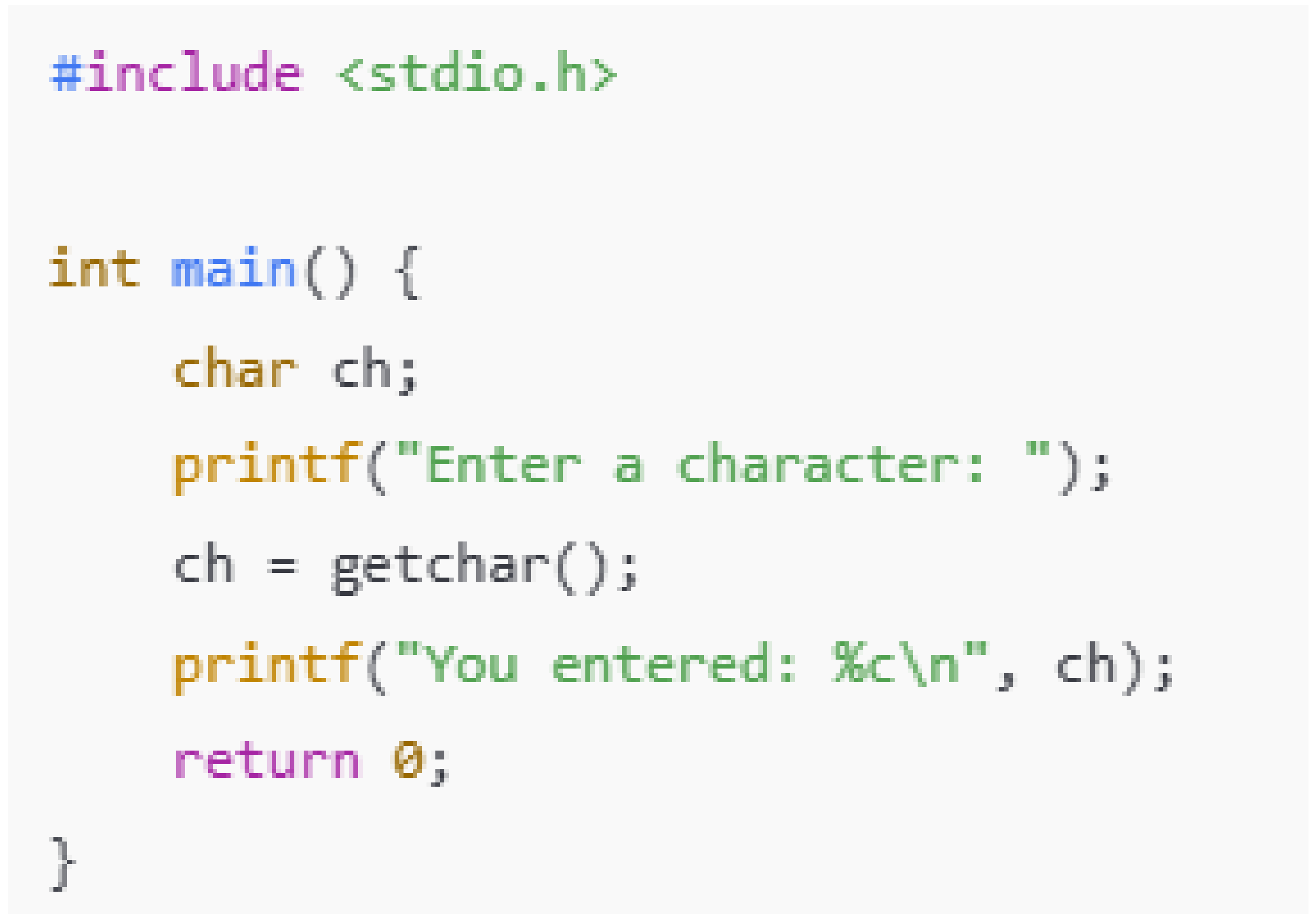
- Can read multiple values at once.
- Requires the address-of operator (&) for non-string variables.
- Stops reading at the first whitespace character (space, tab, or newline).
- Can cause buffer issues if not used carefully (e.g., failing to handle newline characters properly).

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.



```
#include <stdio.h>

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

Practice

AI & Big Data

Standard Input (Handling Newline (₩n) Issues)

Problem: Buffer Retains \n

If getchar is used after scanf, **it may read the leftover newline (\n)**.

```
#include <stdio.h>

int main() {
    int num;
    char ch;

    printf("Enter a number: ");
    scanf("%d", &num);  // Reads a number

    printf("Enter a character: ");
    ch = getchar();  // Problem: This reads the newline ('\n') from the buffer!

    printf("Number: %d, Character: %c\n", num, ch);
    return 0;
}
```

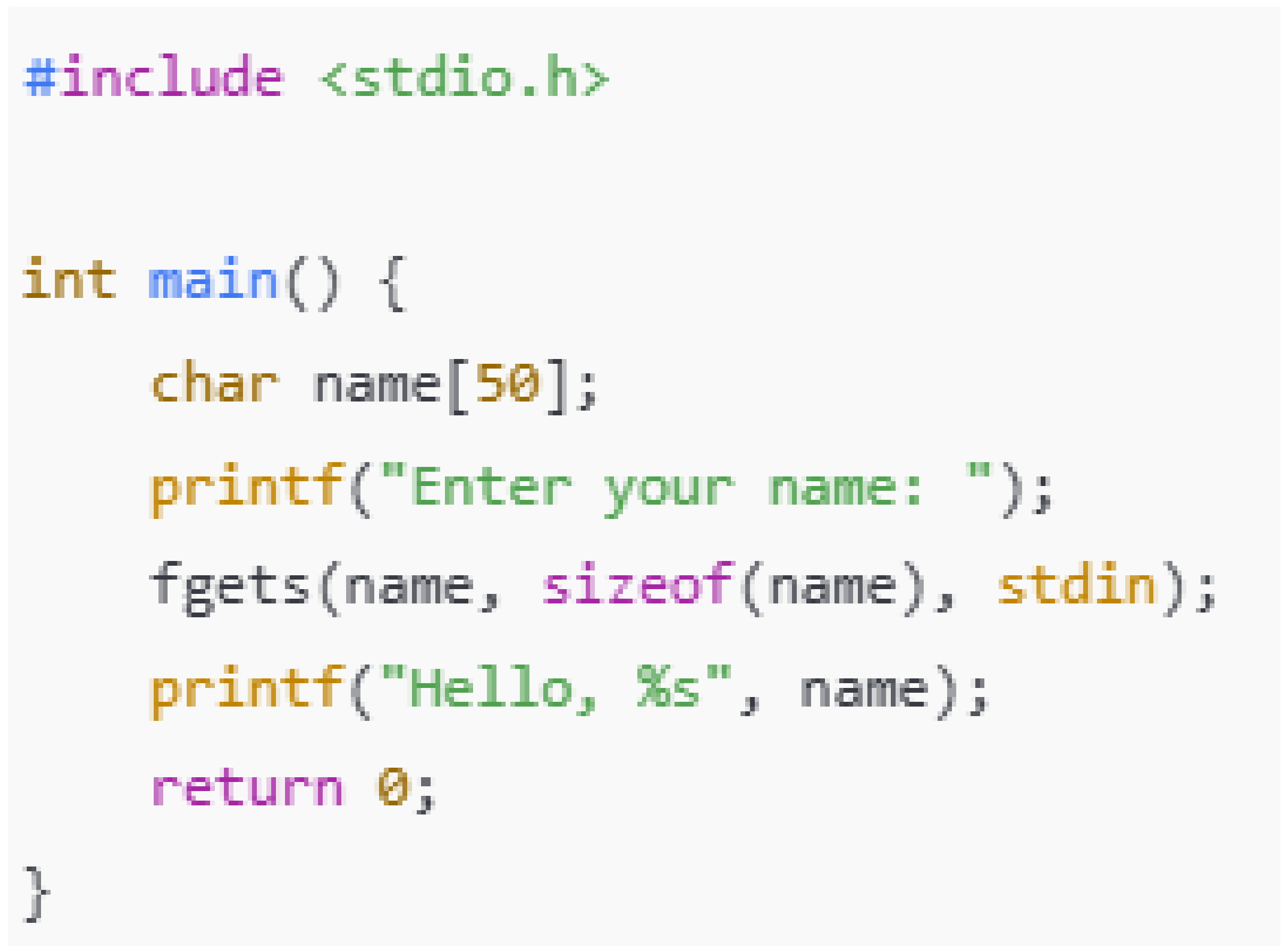
Practice

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.
- It includes ‘\n’ → hello\n\0



```c
#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.

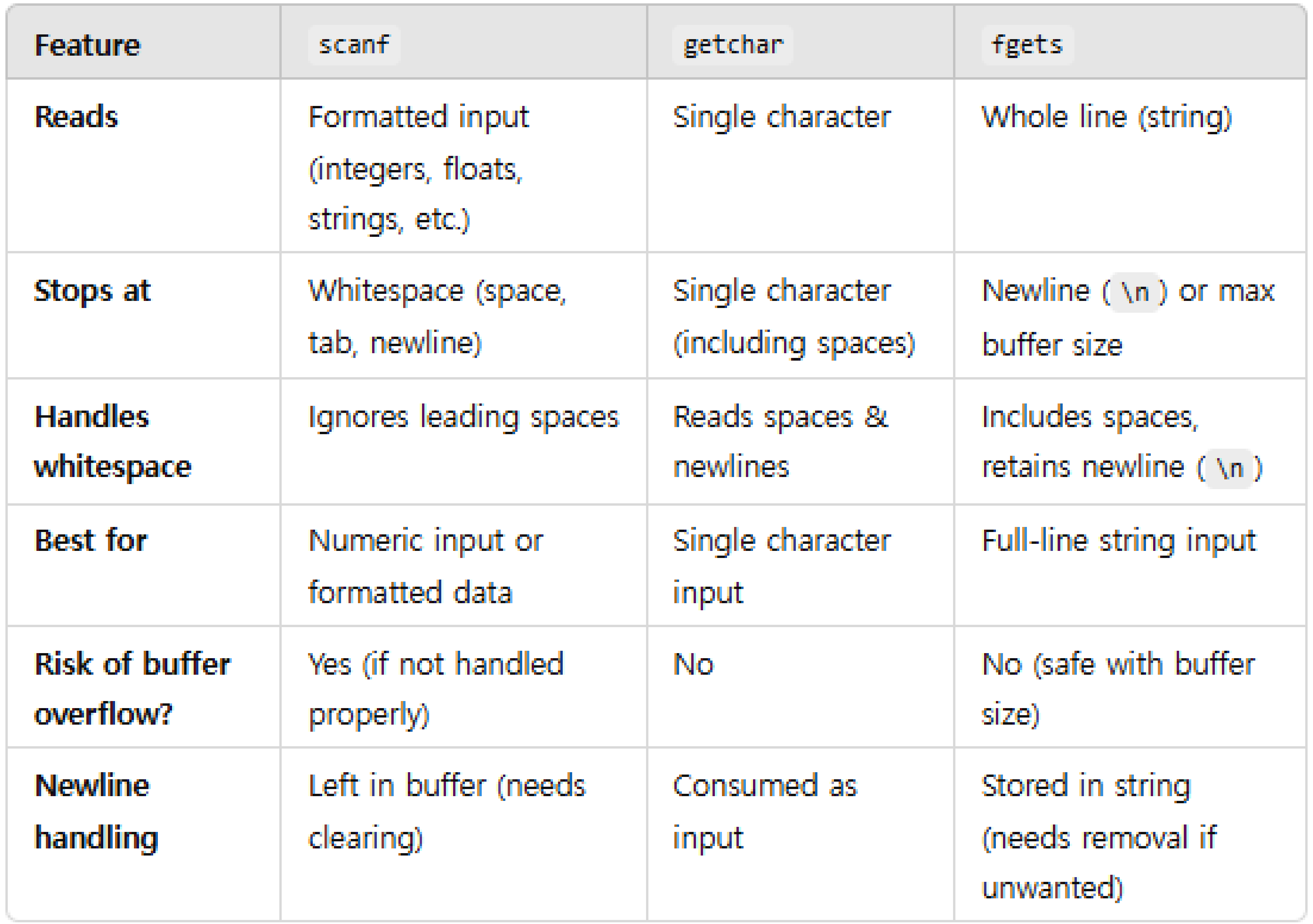
- Unlike scanf, it does not skip spaces.

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

Standard Input

Comparison of scanf, getchar, fgets

-

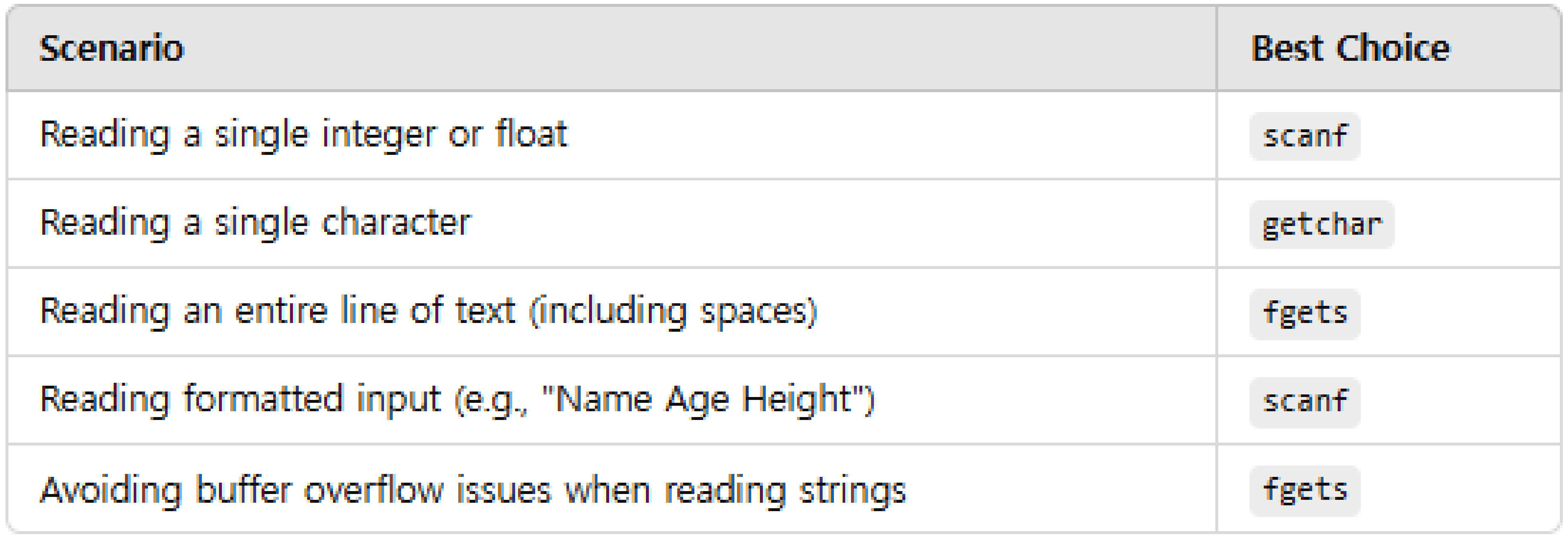


Feature	`scanf`	`getchar`	`fgets`
Reads	Formatted input (integers, floats, strings, etc.)	Single character	Whole line (string)
Stops at	Whitespace (space, tab, newline)	Single character (including spaces)	Newline (`\n`) or max buffer size
Handles whitespace	Ignores leading spaces	Reads spaces & newlines	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?

-

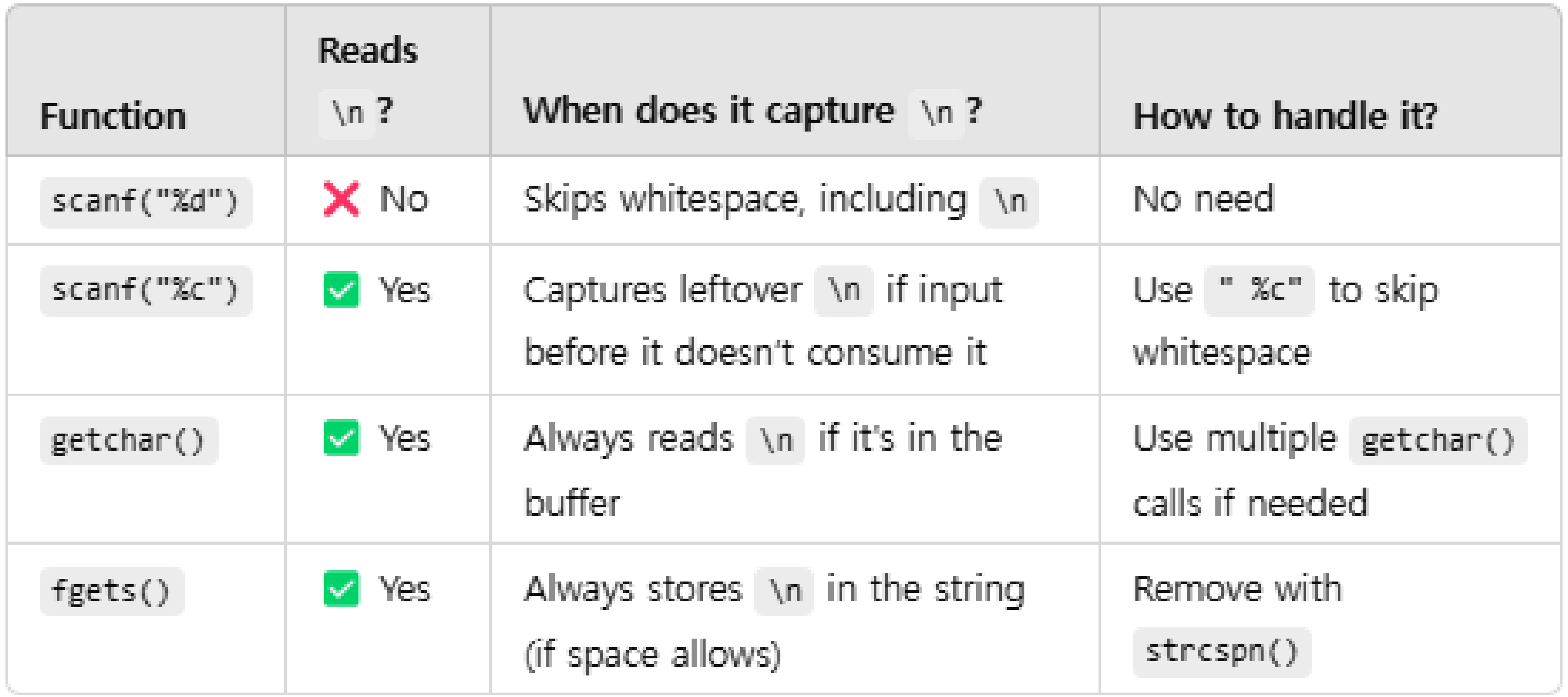


Scenario	Best Choice
Reading a single integer or float	`scanf`
Reading a single character	`getchar`
Reading an entire line of text (including spaces)	`fgets`
Reading formatted input (e.g., "Name Age Height")	`scanf`
Avoiding buffer overflow issues when reading strings	`fgets`

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()`

See you next week!
DO NOT miss the classes