Unit - 2
Introduction to C Programming

By Er. Abinash Adhikari

Overview and History of C

- ❖ **Origins of C**
  - ➢ Developed by Dennis Ritchie at Bell Labs in the early 1970s.
  - ➢ Originally, C was created as a tool to help with system programming, particularly to rewrite the Unix operating system, which was initially developed in assembly language.
  - ➢ Ritchie and his colleague, Ken Thompson, needed a language that was more portable than assembly but still provided low-level memory access for effective system programming.
- ❖ **Why C was Needed**
  - ➢ Flexibility and Control: Assembly language offered detailed control over hardware but was complex and not portable.
  - ➢ Efficiency: Unlike high-level languages of the time, C could compile into efficient machine code, which was crucial for system programs that required high performance.
  - ➢ Portability: Assembly code had to be rewritten for every new machine, but C allowed Unix to be more easily ported across different machines with fewer changes to the code.

❖ Legacy and Influence

- ➢ Foundation for Modern Languages: C influenced the development of many modern languages, such as C++, Java, and Python. These languages inherit syntax and structure from C, making it a foundational language for software development.
- ➢ Widely Used for System Programming: Operating systems, databases, compilers, and embedded systems still rely heavily on C because of its speed and low-level memory management capabilities.
- ➢ Longevity: Despite its age, C remains popular because of its efficiency, simplicity, and close-to-hardware capabilities, making it ideal for performance-critical applications.

Features of C

- ❖ **Low-Level Access to Memory**
  - ➢ C provides powerful memory management capabilities, allowing direct manipulation of memory addresses using pointers.
  - ➢ This feature enables developers to optimize applications by controlling memory usage efficiently, which is crucial for system-level programming like operating systems, device drivers, and embedded systems.
- ❖ **Structured Programming Language**
  - ➢ C is structured, meaning it allows programs to be broken down into smaller, manageable functions or modules.
  - ➢ Each function performs a specific task, making it easier to test, debug, and maintain code.
  - ➢ C supports various control structures like if, for, while, and switch statements, allowing for organized and readable code.

- ❖ **Portability**
  - ➢ Programs written in C can be run on different types of machines with little or no modification, thanks to the language's low-level access and structured design.
  - ➢ This feature allows C code to be reused across multiple platforms, which is one of the reasons why Unix was initially rewritten in C, making it portable to different hardware systems.
- ❖ **Rich Standard Library**
  - ➢ C comes with a vast library of built-in functions, especially for handling input/output operations, memory allocation, string manipulation, mathematical computations, and file operations.
  - ➢ The standard library reduces the need for rewriting commonly used functionality, saving development time and ensuring reliability.

- **Efficient and Fast Execution**
  - C compiles directly into machine code, resulting in high-performance executables with minimal overhead.
  - This efficiency makes C ideal for applications where performance is critical, such as operating systems, embedded systems, and high-performance computing.
- **Extensibility**
  - C allows for the creation of user-defined functions, which can extend its functionality.
  - These functions can be reused and combined to create complex programs, allowing developers to build libraries and modules that enhance program flexibility and functionality.

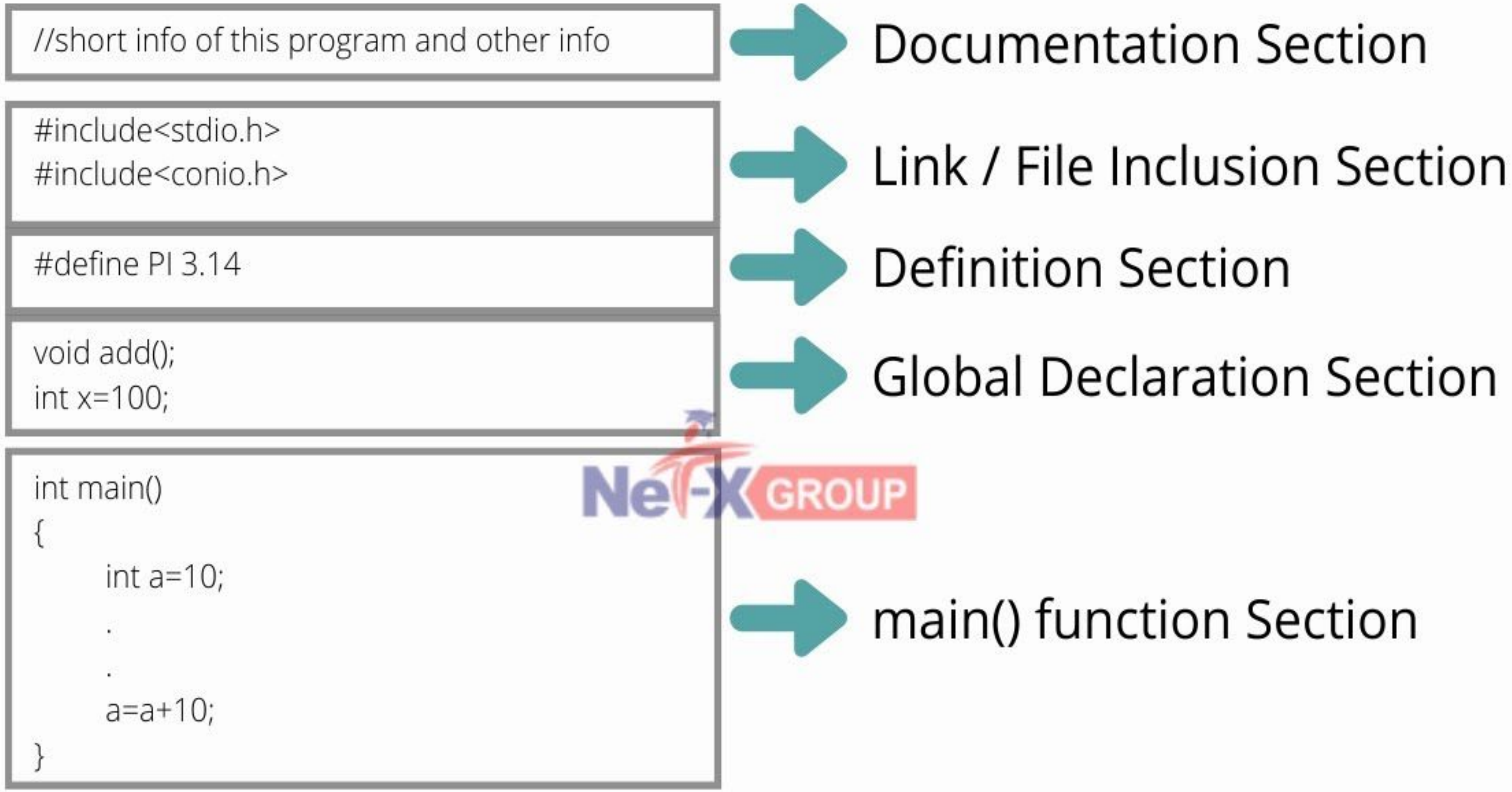
Advantages of C

1. Efficiency and Performance
2. Portability
3. Flexibility and Low-Level Access
4. Structured Language
5. Rich Library of Functions
6. Foundation for Modern Languages

Disadvantages of C

1. No Object-Oriented Programming (OOP) Support
2. Manual Memory Management
3. Limited Error Handling
4. Lack of Security Features
5. No Built-In Support for Threads or Multi-Threading
6. Verbose Syntax

Structure of C Programming



```
//short info of this program and other info
```

```
#include<stdio.h>
#include<conio.h>
```

```
#define PI 3.14
```

```
void add();
int x=100;
```

```
int main()
{
        int a=10;
        .
        .
        a=a+10;
}
```

Structure of a C Program

A typical C program has a specific structure that helps organize code into readable and functional segments. Here’s a basic breakdown:

- ❖ **Preprocessor Directives**
  - ➢ Located at the beginning of the program, preprocessor directives tell the compiler to include or define certain resources before compiling the code.
  - ➢ Common directives include #include (for including libraries) and #define (for defining constants).
  - ➢ Example: #include <stdio.h>
- ❖ **Global Declarations (optional)**
  - ➢ Global variables, constants, or function prototypes can be declared here, making them accessible throughout the entire program.

❖ main() Function

- ➢ Every C program must have a main() function, which is the entry point of the program.
- ➢ The main() function may return an int type and optionally accept parameters.

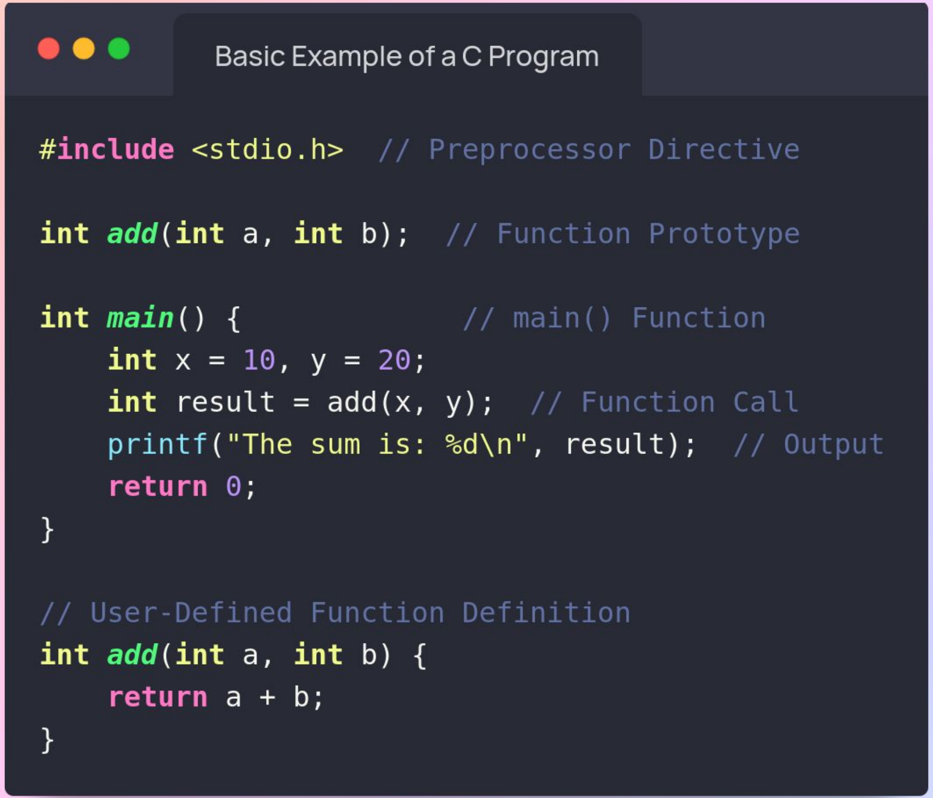
```
int main() {
        // code statements
        return 0;
}
```

❖ Variable Declarations and Executable Statements

- ➢ Inside main() (or any other function), you declare variables and write statements that perform specific tasks.
- ➢ Code statements are executed sequentially within the function.

❖ User-Defined Functions (optional)

- ➢ Additional functions can be defined after main() to break down the program into smaller, manageable parts.
- ➢ This modular approach improves code readability and reusability.


Basic Example of a C Program

```c
#include <stdio.h>  // Preprocessor Directive

int add(int a, int b);  // Function Prototype

int main() {             // main() Function
    int x = 10, y = 20;
    int result = add(x, y);  // Function Call
    printf("The sum is: %d\n", result);  // Output
    return 0;
}

// User-Defined Function Definition
int add(int a, int b) {
    return a + b;
}
```

Compiling Process in C

The compiling process in C converts source code into an executable file through several stages:

- **Preprocessing**
  - The preprocessor handles all preprocessor directives (like #include and #define).
  - It removes comments, expands macros, and includes necessary header files into the source code.
  - Result: An expanded source code file.
- **Compilation**
  - The compiler translates the preprocessed code into assembly code for the specific machine architecture.
  - Syntax errors and warnings are reported at this stage.
  - Result: Assembly code file (.s or .asm file).
- **Assembly**
  - The assembler converts the assembly code into machine code (binary format).
  - Result: An object file (.o or .obj file) containing machine code.
- **Linking**
  - The linker combines all object files, along with libraries and functions required by the program, into a single executable file.
  - If any referenced functions are missing, the linker will report errors at this stage.
  - Result: Executable file (e.g., a.out on Unix or .exe on Windows).

Character Set Used in C

The character set includes all characters allowed in C programs, which are classified into different types:

- ❖ Letters: Uppercase (A-Z) and lowercase (a-z) alphabets.
- ❖ Digits: Numbers from 0-9.
- ❖ Special Characters: Symbols such as +, -, *, /, &, %, #, @, etc.
- ❖ White Space: Spaces, tabs, newline characters, etc.
- ❖ Escape Sequences: Represented by a backslash (\\), followed by a character, e.g., \n for new line, \t for tab, \\\\ for backslash.

Data Types in C

Data types specify the type of data a variable can hold. C has several basic data types:

- ❖ Basic Data Types:
  - ➢ Integer (int): Holds whole numbers, e.g., int x = 5;
  - ➢ Character (char): Holds single characters, e.g., char letter = 'A';
  - ➢ Floating Point (float): Holds decimal numbers, e.g., float num = 5.5;
  - ➢ Double Precision (double): Holds large decimal numbers with more precision, e.g., double value = 3.14159;
- ❖ Derived Data Types:
  - ➢ Array: Collection of variables of the same type.
  - ➢ Pointer: Variable that stores the memory address of another variable.
  - ➢ Structure: Collection of variables of different data types.
  - ➢ Union: Similar to a structure but shares memory for all members.
- ❖ Void Type:
  - ➢ void: Indicates no data type or empty, commonly used in functions that don’t return a value.

Variables in C

Variables are named storage locations in memory that hold values and can be modified. They must be declared with a data type before use.

- ❖ **Syntax:**
  - ➢ data_type variable_name = value;
  - ➢ Example: int age = 25;
- ❖ **Rules:**
  - ➢ Variable names must start with a letter or underscore.
  - ➢ They can contain letters, digits, and underscores but not spaces or special characters.
  - ➢ C is case-sensitive, so age and Age are different.

C Tokens

Tokens are the smallest units in a C program and include:

- **Keywords:**
  - Reserved words with special meaning, e.g., int, return, void, if, else, while, for, etc.
  - Cannot be used as identifiers (variable names).
- **Identifiers:**
  - Names given to variables, functions, arrays, etc.
  - Must be unique within their scope and follow the naming rules.

- **Constants:**
  - Fixed values that cannot be altered during program execution.
  - Types of constants:
    - Integer Constants: e.g., 10, -50
    - Floating Point Constants: e.g., 3.14, -0.76
    - Character Constants: Single characters enclosed in single quotes, e.g., 'A'
    - String Constants: Series of characters enclosed in double quotes, e.g., "Hello"
- **Operators:**
  - Symbols that perform operations on variables and values:
  - Arithmetic Operators: +, -, *, /, %
  - Relational Operators: ==, !=, <, >, <=, >=
  - Logical Operators: && (AND), || (OR), ! (NOT)
  - Assignment Operators: =, +=, -=, etc.

Header Files

Header files contain definitions of functions and macros that can be included in multiple programs. Common header files in C are:

- <stdio.h>: Standard input and output functions, e.g., printf(), scanf()
- <stdlib.h>: General utilities like memory allocation, e.g., malloc(), free()
- <math.h>: Mathematical functions, e.g., sqrt(), pow()
- <string.h>: String manipulation functions, e.g., strcpy(), strlen()
- <ctype.h>: Character handling functions, e.g., isalpha(), isdigit()

Library Functions in C

Library functions are predefined functions provided by C’s standard library to perform common tasks. Some commonly used functions include:

- **Input/Output Functions (<stdio.h>):**
  - printf(): Prints output to the console.
  - scanf(): Reads input from the console.
- **String Functions (<string.h>):**
  - strcpy(): Copies one string to another.
  - strlen(): Returns the length of a string.
- **Mathematical Functions (<math.h>):**
  - sqrt(): Returns the square root of a number.
  - pow(): Raises a number to a power.
- **Memory Allocation Functions (<stdlib.h>):**
  - malloc(): Allocates memory.
  - free(): Frees allocated memory.

Preprocessor Directives

Preprocessor directives in C are commands that are processed by the preprocessor before the code is compiled. They begin with a # symbol and do not require a semicolon. Common preprocessor directives include:

- **#include**
  - Used to include standard libraries or other files in a program.
  - Example: #include <stdio.h> includes the standard I/O library for functions like printf and scanf.
- **#define**
  - Defines constant values or macros, which are code snippets replaced by the defined value before compilation.
  - Example: #define PI 3.14 defines PI as 3.14 throughout the code.

❖ #undef

- ➢ Used to undefine a macro or constant defined with #define.
- ➢ Example:

```
#define SIZE 100
#undef SIZE
```

❖ Conditional Compilation (#if, #ifdef, #ifndef, #else, #endif)

- ➢ Controls which parts of the code are compiled.
- ➢ Useful for including/excluding code based on certain conditions or for platform-specific code.
- ➢ Example:

```
#ifdef DEBUG
    printf("Debug mode\n");
#endif
```

Escape Sequences

Escape sequences are special character combinations in C that start with a backslash (\) and represent a specific non-printable character. They are commonly used for formatting output or representing special characters.

- ❖ **Newline (\n)**
  - ➢ Moves the cursor to the next line.
  - ➢ Example: printf("Hello\nWorld"); outputs:

    Hello

    World
- ❖ **Tab (\t)**
  - ➢ Inserts a horizontal tab space.
  - ➢ Example: printf("Hello\tWorld");
  - ➢ outputs: Hello   World

- ❖ **Backslash (\\)**
  - ➢ Prints a backslash character (\).
  - ➢ Example: printf("C:\\Program Files");
  - ➢ Outputs: C:\Program Files
- ❖ **Single Quote (\')**
  - ➢ Used to print a single quote.
  - ➢ Example: printf("It\'s a sunny day.");
  - ➢ Outputs: It's a sunny day.
- ❖ **Double Quote (\")**
  - ➢ Prints a double quote.
  - ➢ Example: printf("\"Hello, World\"");
  - ➢ Outputs: "Hello, World"
- ❖ **Alert/Bell (\a)**
  - ➢ Produces an audible alert (beep) on the system.
  - ➢ Example: printf("\a");

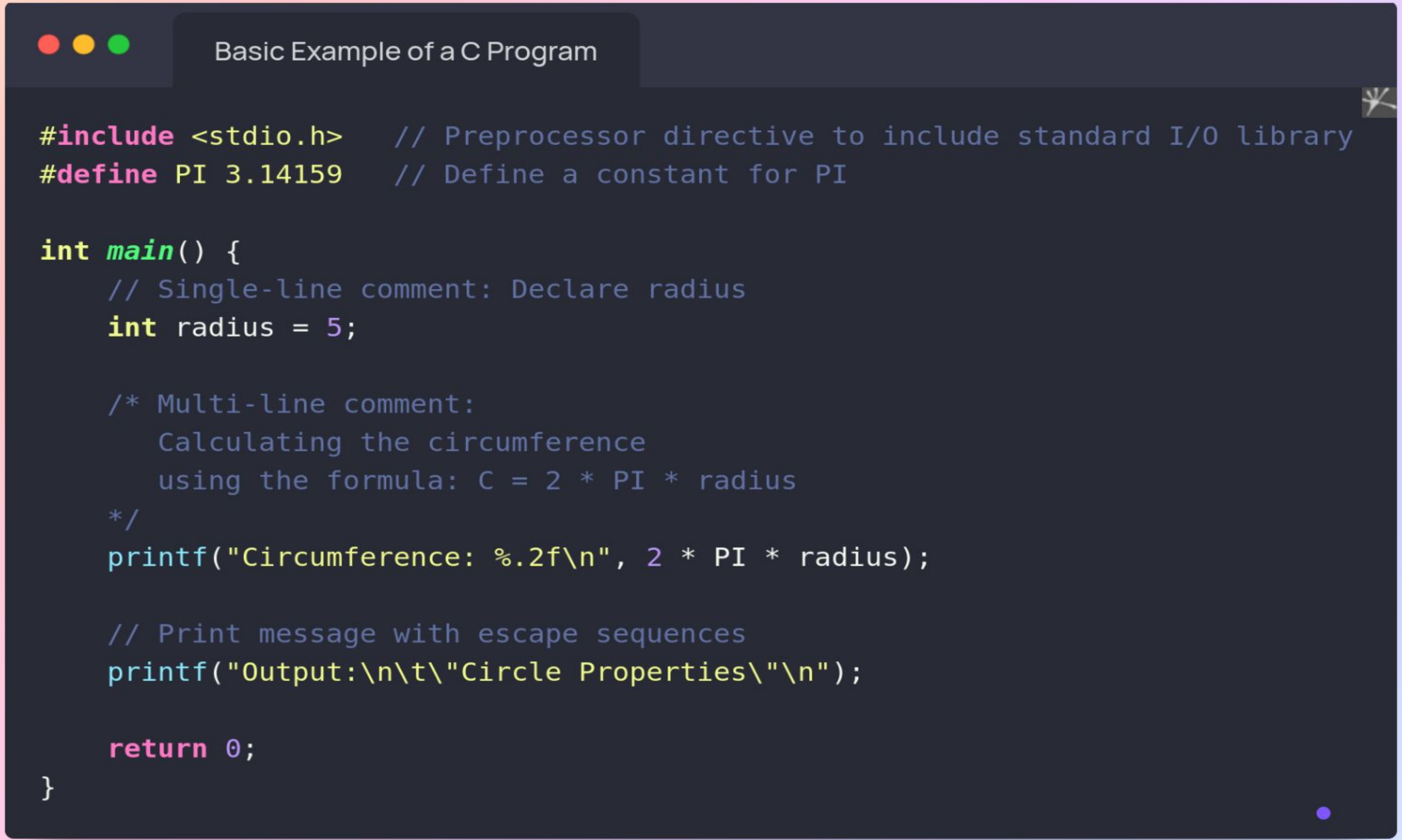
Comments

Comments are non-executable lines in the code meant for documentation or explanation. They are ignored by the compiler and do not affect the program’s output. Comments improve code readability and are useful for explaining complex logic, providing reminders, or noting sections for future reference.

- **Single-Line Comment**
  - Begins with // and spans a single line.
  - Example:

    ```
    int x = 5;  // Declaring variable x with value 5
    ```
- **Multi-Line Comment**
  - Begins with /* and ends with */, allowing multiple lines of comments.
  - Example:

    ```
    /*
      This is a multi-line comment.
      Used for longer explanations.
    /*
    ```
  - int y = 10;


Basic Example of a C Program

```c
#include <stdio.h>   // Preprocessor directive to include standard I/O library
#define PI 3.14159   // Define a constant for PI

int main() {
    // Single-line comment: Declare radius
    int radius = 5;

    /* Multi-line comment:
       Calculating the circumference
       using the formula: C = 2 * PI * radius
    */
    printf("Circumference: %.2f\n", 2 * PI * radius);

    // Print message with escape sequences
    printf("Output:\n\t\"Circle Properties\"\n");

    return 0;
}
```

Formatted Input/Output Functions

Formatted I/O functions in C provide control over how data is read or displayed. The two primary functions are printf() for output and scanf() for input.

- **printf() - Output**
  - printf() is used to display output on the screen, allowing for specific format specifications.
  - Syntax:printf("format string", arguments);
  - Format Specifiers:
    - %d for integers
    - %f for floating-point numbers
    - %c for characters
    - %s for strings
  - Example:
    - int age = 20;
    - printf("Age: %d\n”, age); // Outputs: Age: 20

- **scanf() - Input**
  - scanf() is used to read input from the user based on specified format specifiers.
  - Syntax:
    - scanf("format string", &variable);
  - Note: The & symbol (address-of operator) is used with variables to store the input in the specified memory address.
  - Example:
    - int age;
    - printf("Enter your age: ");
    - scanf("%d", &age); // User inputs an integer value

Unformatted Input/Output Functions

Unformatted I/O functions handle characters or strings without format control. These functions are commonly used for basic character or string input/output operations.

Character Input/Output Functions

- **getchar() - Reads a single character from the user.**
  - Syntax: int getchar(void);
  - Example:

    ```
    char ch;
    printf("Enter a character: ");
    ch = getchar();
    printf("You entered: %c\n", ch);
    ```

- **putchar() -** Outputs a single character to the screen.
  - Syntax: int putchar(int char);
  - Example:

    ```
    char ch = 'A';
    putchar(ch);  // Outputs: A
    ```
- **getc(FILE *stream) -** Reads a character from a specified file or input stream.
  - Example:

    ```
    char ch;
    ch = getc(stdin);  // Reads a character from the standard input
    ```
- **putc(int char, FILE *stream) -** Writes a character to a specified file or output stream.
  - Example:

    ```
    putc('B', stdout);  // Outputs: B
    ```

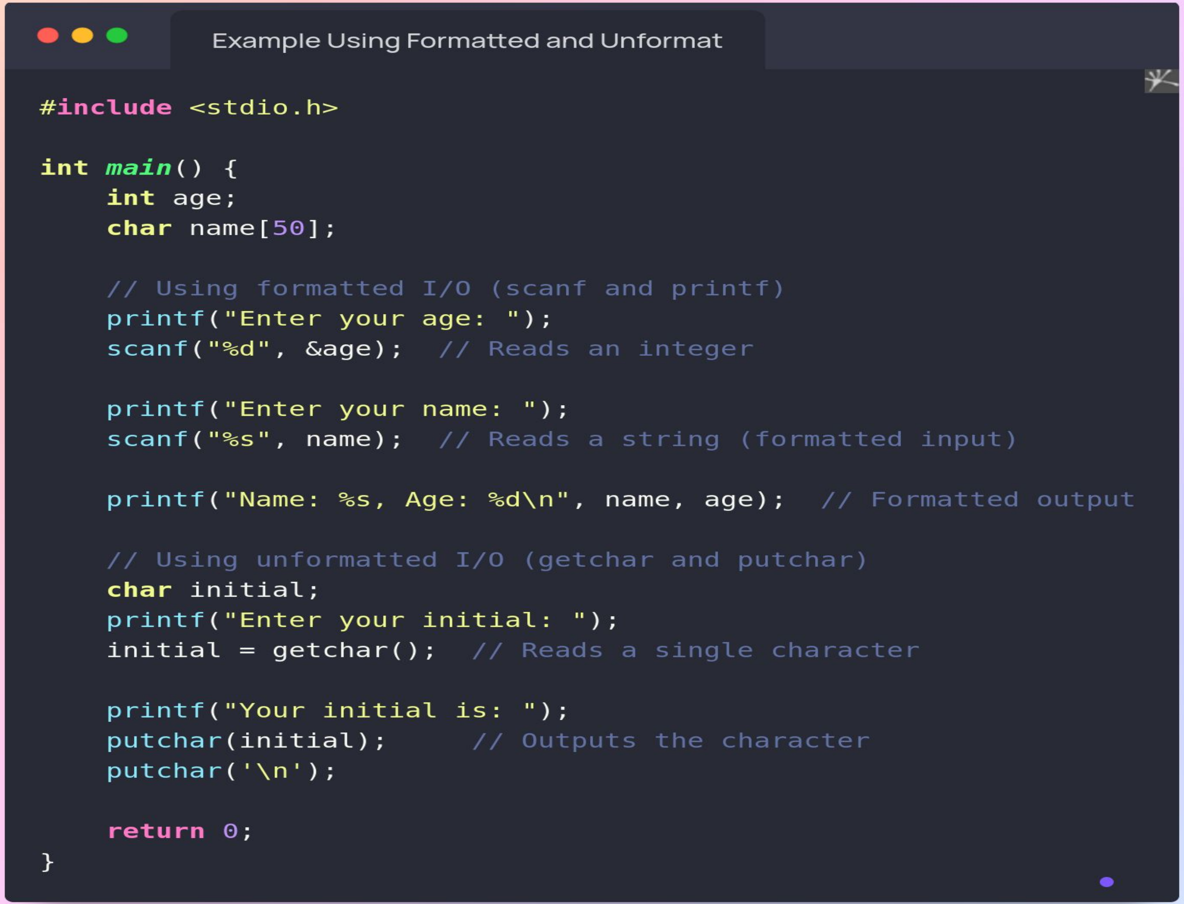
String Input/Output Functions

- ❖ **gets()** - Reads a string from the user until a newline character is encountered.
  - ➢ Syntax: char *gets(char *str);
  - ➢ Note: gets() is not recommended due to potential buffer overflow issues. Safer alternatives, like fgets(), are often preferred.
  - ➢ Example:

```
char name[50];
printf("Enter your name: ");
gets(name);
printf("Hello, %s\n", name);
```

- ❖ **puts()** - Outputs a string to the screen and adds a newline at the end.
  - ➢ Syntax: int puts(const char *str);
  - ➢ Example:

```
char name[] = "Alice";
puts(name);  // Outputs: Alice (followed by a newline)
```


Example Using Formatted and Unformat

```c
#include <stdio.h>

int main() {
    int age;
    char name[50];

    // Using formatted I/O (scanf and printf)
    printf("Enter your age: ");
    scanf("%d", &age);  // Reads an integer

    printf("Enter your name: ");
    scanf("%s", name);  // Reads a string (formatted input)

    printf("Name: %s, Age: %d\n", name, age);  // Formatted output

    // Using unformatted I/O (getchar and putchar)
    char initial;
    printf("Enter your initial: ");
    initial = getchar();  // Reads a single character

    printf("Your initial is: ");
    putchar(initial);     // Outputs the character
    putchar('\n');

    return 0;
}
```