**Programming for Problem Solving**

**WEEK-1**

**LONG DESCRIPTIVE QUESTION**

**1, Explain program structure with an example program**

Programming refers to the organization and ordering of rules in a computer program. A well-structured program is easy to read, understand, maintain, and debug. This often includes many elements, including written statements, declarations, practices or procedures, and implementation policies. Here is a description of the program in a sample program in Python

# Example Python program to calculate the **factorial of a number**

# Function to calculate the factorial of a number

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

# Main program

if \_\_name\_\_ == "\_\_main\_\_":

# Input from the user

num = int(input("Enter a number: "))

# Check if the input is negative

if num < 0:

print("Factorial is not defined for negative numbers.")

else:

result = factorial(num)

print(f"The factorial of {num} is {result}")

Let's break down the structure of this program

**Comments:** Comments are used to provide a human-readable explanation of the code. Comments in Python begin with the # symbol. They are not executed and are only paperwork.

**Function Declaration:** The function defines a factorial function. Programs contain some code that can be reused. In this case, the factor calculates the factor of the number using iterations.

**Main Function** The main function starts with the if \_\_name\_\_ == "\_\_main\_\_": block. This section is executed only when the script is executed directly (not imported as a module). This is the central argument of the program.

**User Input:** The program uses the input() function to get user input. In this case, it prompts the user to enter a number, which is stored in the variable num.

**Conditional statements:** The program checks whether the input number is negative or not. If it is negative, it prints a message indicating that a factorial is not defined for negative numbers. Otherwise, it continues to calculate and print the attribute.

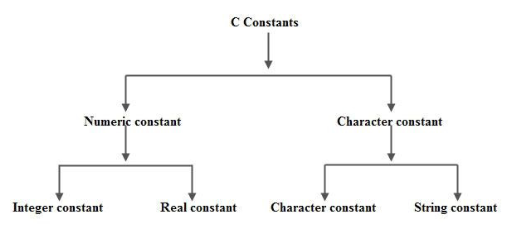
**Function Call:** The factor function is called as an argument with the user's number input, and the result is stored in a variable.

**Output:** Finally, the program prints the result, which is a factorial of the input number.

**2, Explain the constant types with block diagram**

There are many different types of data values that are implicitly declared as constants in C. The value of a constant cannot be changed during execution of the program, Value does not change during execution. Constant can be either a Number (or) a Letter. C Language supports several

Types of Constants:



* **Integer Constants:**

An integer constant is a sequence of digits from 0 to 9 without decimal points or fractional part or any other symbols. There are 3 types of integers namely decimal integer, octal integers and hexadecimal integer.

**Decimal Integers** consists of a set of digits 0 to 9 preceded by an optional + or – sign. Spaces, commas and non digit characters are not permitted between digits. Example for valid decimal integer constants are

1 inty = 123; //here 123 is a decimal integer constant

* **Real Constants:**

Real Constants consists of a fractional part in their representation. Integer constants are inadequate to represent quantities that vary continuously. These quantities are represented by numbers containing fractional parts like 26.082. Example of real

constants are

1 floatx = 6.3; //here 6.3 is a double constant

2 floaty = 6.3f; //here 6.3f is a float constant.

Real Numbers can also be represented by exponential notation. The general form for exponential notation is mantissa exponent. The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign.

* **Single Character Constants:**

A Single Character constant represent a single character which is enclosed in a pair of quotation symbols.

Example for character constants are

1 charp ='ok'; // p will hold the value 'O' and k will be omitted

2 chary ='u'; // y will hold the value 'u'

All character constants have an equivalent integer value which are called ASCII Values.

* **String Constants:**

A string constant is a set of characters enclosed in double quotation marks. The characters in a string constant sequence may be a alphabet, number, special character and blank space.

Example of string constants are

1 "VISHAL""1234""God Bless""!.....?"

**Backslash Character Constants [Escape Sequences]**

Backslash character constants are special characters used in output functions. Although they contain two characters they represent only one character. Given below is the table of escape sequence and their meanings.

|  |  |
| --- | --- |
| **Constant** | **Meaning** |
| ‘\a’ | .Audible Alert (Bell) |
| ‘\b’ | .Backspace |
| ‘\f’ | .Formfeed |
| ‘\n’ | .New Line |
| ‘\r’ | .Carriage Return |
| ‘\t’ | .Horizontal tab |
| ‘\v’ | .Vertical Tab |
| ‘\” | .Single Quote |
| ‘\”‘ | .Double Quote |
| ‘\?’ | .Question Mark |
| ‘\\’ | .Back Slash |
| ‘\0’ | .Null |

**3, State the rules in writing C Program?**

Writing a C program requires following certain rules and best practices to ensure code readability, maintainability, and correctness. Here are ten important rules to keep in mind when writing C programs:

**Include Necessary Headers:** Include the required header files at the beginning of your program. This is important to use standard functions and libraries. For example**, #include** **<stdio.h>** for input and output functions.

**Use Meaningful Variable Names:** Choose descriptive names for variables and identifiers to make your code self-explanatory. Avoid single-letter variable names like x or y.

**Indentation and Formatting:** Use consistent and clear indentation to make your code visually organized. Follow a consistent code formatting style to enhance readability.

**Comments:** Add comments to explain the purpose of your code, important algorithms, and complex sections. Well-placed comments can help others (and yourself) understand the code's logic.

**Modularize Your Code:** Break your program into functions or modules to promote code reusability and maintainability. Each function should ideally have a single, well-defined purpose.

**Error Handling:** Check for errors and handle them gracefully. Use appropriate error messages and return codes to make debugging easier.

**Avoid Global Variables:** Minimize the use of global variables as they can lead to unexpected side effects. Use local variables within functions whenever possible.

**Memory Management:** Properly allocate and deallocate memory using functions like “malloc”, “free”, “calloc”, or “realloc” to prevent memory leaks and buffer overflows.

**Input Validation:** Validate user input and external data to prevent security vulnerabilities and unexpected behavior. Ensure that the input meets the expected criteria.

**Testing and Debugging:** Regularly test your code with different inputs and edge cases to ensure correctness. Use debugging tools and techniques to identify and fix issues.

**4, When and why to use comments in programming? Write a simple C program to demonstrate the single line and multiline comments?**

Comments are used in programming to provide explanations, documentation, and context for the code. It is essential for proper understanding of code, both by you and by others who may read or apply your code. Here’s when and why you should use Comments

* When to Use Comments:

**Documentation:** Comments are often used to document the purpose of functions, variables, and code blocks, making it easier for others (and your future self) to understand the code's intent.

**Explanations:** Comments can explain complex algorithms, calculations, or any non-obvious parts of the code. They provide a narrative that clarifies what the code is doing.

**TODOs and Notes:** Comments can include notes for future work or improvements. For example, you can use comments like TODO: Implement error handling or NOTE: Optimize this algorithm later.

**Bug Tracking:** You can use comments to identify known issues, bugs, or areas of the code that require further investigation.

**Version History:** In collaborative projects, comments can indicate when and by whom specific code changes were made, aiding in version control and collaboration.

* Why to Use Comments:

**Readability:** Well-placed comments make code more readable and understandable, especially for complex logic or algorithms.

**Maintenance:** Comments help during code maintenance and debugging by providing context and clues for finding and fixing issues.

**Communication:** Comments serve as a form of communication between developers, conveying information about the code's behavior, purpose, and assumptions.

**Knowledge Transfer:** When developers change or join a project, comments are valuable for quickly understanding how the code works without needing to decipher every line of code.

Here's a simple C program that demonstrates the use of single-line and multi-line comments:

#include <stdio.h>

int main() {

// This is a single-line comment

printf("Hello, "); // This comment explains the next part of the code

printf("World!\n");

/\*

This is a multi-line comment.

It can span multiple lines and is often used for more extensive explanations.

\*/

return 0;

}

In this example:

// This is a single-line comment is a single-line comment. Anything following // on the same line is considered a comment and is not executed.

/\* ... \*/ is a multi-line comment. Anything enclosed within /\* and \*/ is considered a comment, and it can span multiple lines.

Both types of comments are used to provide explanations and make the code more understandable.