Parul® University NAAC A++ ACCREDITED UNIVERSITY

Centre for Distance and Online Education

Online MCA Program

Semester- I

(C)
ASSIGNMENT

Prepared by

Name of the Student :- Dev jitendrabhai kalwani

Enrollment no :- 2501j00600234

1. Explain Structure of C Programming

A C program follows a well-defined structure that helps the compiler understand how to execute the code. Although small programs may not include every section, the general structure remains consistent across all C applications.

1. **Documentation Section**
   This is the optional comment section where you describe the program's purpose, author name, creation date, and other details. It improves code readability. Comments can be written using /*...*/ for multiple lines or // for a single line.

2. **Preprocessor Directives**
   This section includes commands beginning with #, such as #include<stdio.h> or #define PI 3.14. These directives instruct the preprocessor to include header files or define constants before actual compilation starts.

3. **Global Declaration Section**
   Here, you declare global variables, constants, and function prototypes. These can be accessed by any function in the program. For example, int count; or void display();.

4. **Main Function Section**
   Every C program must contain a main() function because execution begins from here. It is usually written as:
   int main() or void main().
   Inside the main block { }, you write variable declarations and executable statements.

5. **Local Declarations**
   Within the main function or any other function, you can declare variables like int a, b;. These variables are only accessible within that function.

6. **Executable Statements**
   These are the instructions that the program actually performs, such as input/output operations, calculations, loops, and conditions. Example: printf("Hello");, sum = a + b;.

7. **User-Defined Functions**
   Large programs often include additional functions defined by the programmer. These functions help break the program into modular, reusable pieces.

2. Explain Datatypes in details

Data types in C define the type of data a variable can store and determine the amount of memory allocated to it. They also specify the operations that can be performed on that data. C provides several categories of data types to handle different kinds of values efficiently.

1. Primary (Basic) Data Types

These are the fundamental types used for simple data.

- int: Used to store whole numbers (positive or negative). Typically occupies 2 or 4 bytes depending on the system.
- float: Stores single-precision decimal numbers using 4 bytes.
- double: Stores double-precision decimal numbers and occupies 8 bytes.
- char: Holds a single character such as ‘A’, ‘b’, or symbols. It uses 1 byte.

Modifiers like short, long, signed, and unsigned can alter the size or range of int and char types.

Examples: unsigned int, long long int, signed char.

2. Derived Data Types

These are built using the basic data types.

- Array: Collection of similar data types stored in contiguous memory (e.g., int arr[5];).
- Pointer: Stores memory addresses of variables (e.g., int *p;).
- Function: A block of code that returns data of a specific type.
- Structure: Combines different data types under one name (e.g., student record).
- Union: Similar to structure but shares memory among members, saving space.

3. User-Defined Data Types

These allow programmers to create their own data types.

- typedef: Creates an alias name for an existing type.
- enum: Used to define a set of named integer constants, improving readability.

4. Void Data Type

void represents “no value.” It is used for functions that return nothing (e.g., void display();) or for generic pointers (void *ptr;).

3. Explain Algorithm and Flowchart with Example

Algorithm

An algorithm is a step-by-step procedure used to solve a problem or perform a task. It is written in simple, plain language so that anyone can understand the logic without needing to know a programming language. Algorithms are the foundation of all computer programs because they describe *what* needs to be done.

Characteristics of a good algorithm:

1. Clear and unambiguous – Each step must have a single meaning.
2. Finite – It must end after a definite number of steps.
3. Input and Output – Should clearly define what inputs are needed and what outputs will be produced.
4. Effective – Steps should be simple and executable.

Example Algorithm: Find the Largest of Two Numbers

1. Start
2. Input two numbers A and B
3. If A > B then
   Display “A is largest”
   Else
   Display “B is largest”
   End

Flowchart :-

A **flowchart** is a visual diagram that represents the flow of steps in an algorithm. It uses different symbols to show actions, decisions, inputs, and outputs. Flowcharts make it easy to understand logic and detect errors in a process.

Common Flowchart Symbols

- **Oval** → Start/End
- **Parallelogram** → Input/Output
- **Rectangle** → Process/Action
- **Diamond** → Decision (Yes/No)
- **Arrow** → Flow of direction

Flowcharts are widely used because they give a clear graphical view of the program's logic, making it easier for beginners and programmers to follow the sequence of steps.

Flowchart Example: Find the Largest of Two Numbers

Flow Description (text representation):

- **Start**
- **Input A, B**
- **Decision**: Is A > B?
  - **Yes** → Output: "A is largest"
  - **No** → Output: "B is largest"
- **End**

4. Explain Operators in details

Introduction

Operators in C are special symbols used to perform operations on variables and values. They help in performing arithmetic, logical decisions, comparisons, and many other tasks. Operators, combined with operands (variables or constants), form expressions that the compiler evaluates.

1. Arithmetic Operators

These operators perform mathematical operations.

- **+** (Addition)
- **-** (Subtraction)
- *** ** (Multiplication)
- **/** (Division)
- **%** (Modulus – remainder after division)

Example:

```
c = a + b;
```

---

2. Relational Operators

Used to compare two values. They always return **true (1)** or **false (0)**.

- **==** (Equal to)
- **!=** (Not equal to)
- **>** (Greater than)
- **<** (Less than)
- **>=** (Greater than or equal to)
- **<=** (Less than or equal to)

Example:

```
if (a > b)
```

---

3. Logical Operators

Used for combining multiple conditions.

- **&&** (Logical AND) – true only if both conditions are true
- **||** (Logical OR) – true if any one condition is true
- **!** (Logical NOT) – reverses the condition

Example:
if (a > 0 && b > 0)

---

4. Assignment Operators

Used to assign values to variables.

- **=** (Simple assignment)
- **+=, -=, *=, /=, %=**
- These modify the value and assign it back.

Example:
a += 5; (same as a = a + 5)

---

5. Increment and Decrement Operators

Used to increase or decrease a value by 1.

- **++** (Increment)
- **--** (Decrement)

They have two forms:

- **pre-increment** (++a) → increments first, then uses value
- **post-increment** (a++) → uses value, then increments

Example:
a++;

---

6. Bitwise Operators

Operate on bits of data.

- **&** (AND)
- **|** (OR)
- **^** (XOR)
- **<<** (Left shift)
- **>>** (Right shift)
- **~** (Bitwise NOT)

Example:
c = a & b;

7. Conditional (Ternary) Operator

A compact form of decision making.

Syntax:
condition ? value1 : value2;

Example:
max = (a > b) ? a : b;

---

8. Special Operators

- **sizeof** → gives size of data type or variable
- **&** (Address-of operator)
- ***** (Pointer dereference operator)
- **,** (Comma operator)

5. Explain Call by Value and Call by Reference

1. Call by Value

In Call by Value, a *copy* of the actual value is passed to the function. This means any changes made inside the function affect only the copied value, not the original variable.

How it works:

- A function receives separate memory locations for its parameters.
- The original variable remains unchanged even if the function modifies its parameter.

Example:

```
void change(int x) {
   x = 20;
}

int main() {
   int a = 10;
   change(a);
   printf("%d", a); // Output: 10
}
```

Here, a remains 10 because only a copy was changed.

Advantages:

- Safer since original data cannot be modified accidentally.
- Simple and easy to understand.

Disadvantages:

- Not suitable when you need to modify the original variables.
- Less efficient for large data because copies require more memory.

2. Call by Reference

In Call by Reference, instead of passing the value, the *address (reference)* of the variable is passed to the function. This gives the function direct access to the original data, allowing it to modify the actual variable.

How it works:

- Parameters become aliases for the original variables.
- Any change in the function reflects directly in the calling function.

Example:

```
void change(int *x) {
   *x = 20;
}

int main() {
   int a = 10;
   change(&a);
   printf("%d", a); // Output: 20
}
```

Advantages:

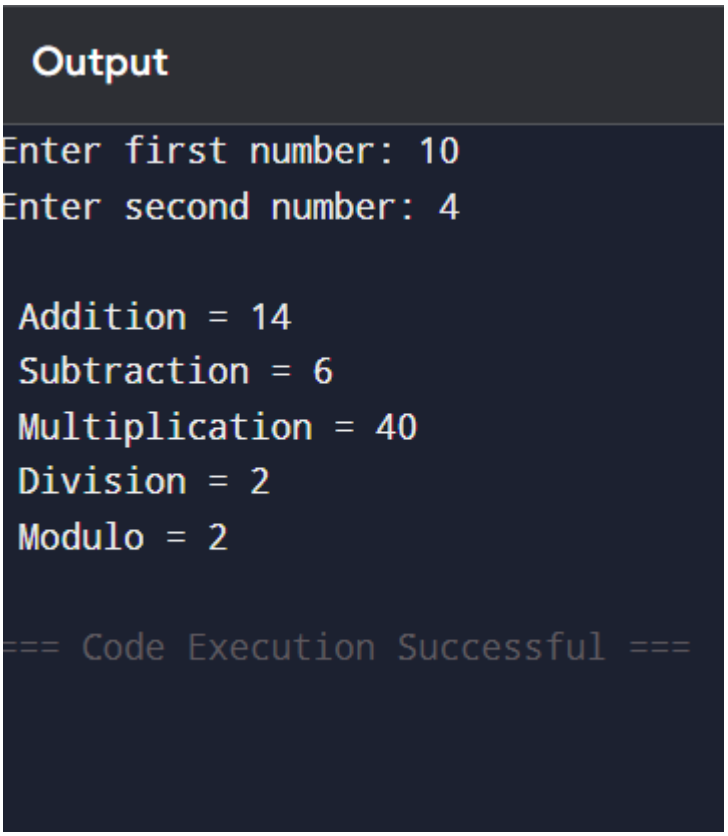
- Allows modification of original variables.
- Efficient for large structures or arrays (no copying).

Disadvantages:

- Less safe since original data can be unintentionally changed.
- Requires understanding pointers, which may be confusing for beginners.

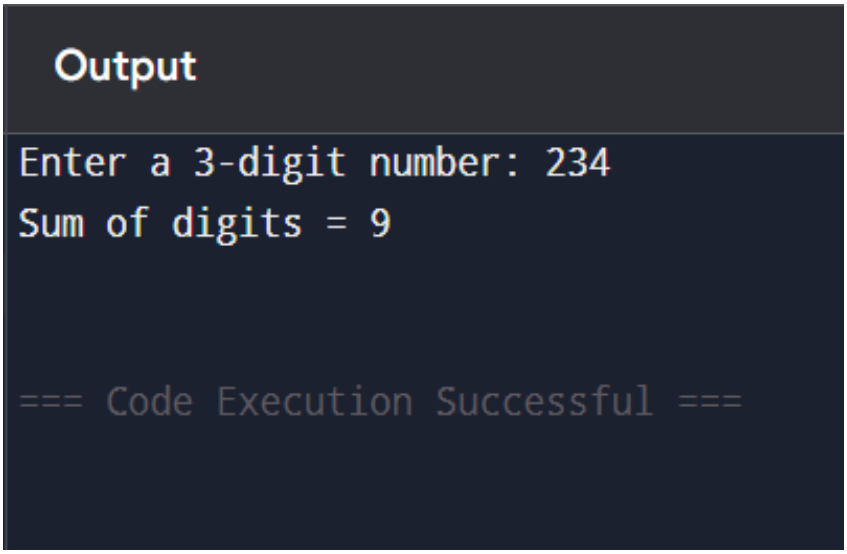
6. Write a program that reads two nos. from keyboard and gives their addition, subtraction, multiplication, division and modulo.

```
#include <stdio.h>

int main() {
   int a, b;

   printf("Enter first number: ");
   scanf("%d", &a);

   printf("Enter second number: ");
   scanf("%d", &b);

   printf("\n Addition = %d", a + b);
   printf("\n Subtraction = %d", a - b);
   printf("\n Multiplication = %d", a * b);

   if (b != 0) {
      printf("\n Division = %d", a / b);
      printf("\n Modulo = %d", a % b);
   } else {
      printf("\n Division and Modulo not possible (division by
zero).");
   }

   return 0;
}
```


Output
Enter first number: 10
Enter second number: 4
Addition = 14
Subtraction = 6
Multiplication = 40
Division = 2
Modulo = 2
=== Code Execution Successful ===

7. Write a program which calculates the summation of three digits from the given 3-digit number.

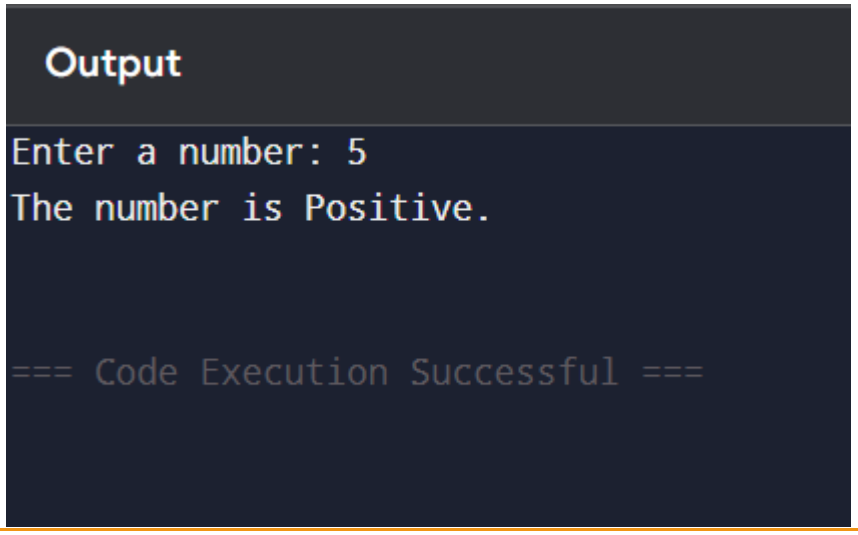
```
#include <stdio.h>

int main() {
   int num, a, b, c, sum;

   printf("Enter a 3-digit number: ");
   scanf("%d", &num);

   // Extract digits
   a = num / 100;       // First digit
   b = (num / 10) % 10;  // Second digit
   c = num % 10;        // Third digit

   sum = a + b + c;

   printf("Sum of digits = %d\n", sum);

   return 0;
}
```


Output
Enter a 3-digit number: 234
Sum of digits = 9
=== Code Execution Successful ===

8. Write a C Program which demonstrate Switch Statement.

```
#include <stdio.h>

int main() {
   int num, check;

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

   if (num > 0)
      check = 1;
   else if (num < 0)
      check = -1;
   else
      check = 0;

   switch (check) {
      case 1:
         printf("The number is Positive.\n");
         break;

      case -2:
         printf("The number is Negative.\n");
         break;

      case 3:
         printf("The number is Neutral (Zero).\n");
         break;
```

```
        default:
            printf("Invalid Input.\n");
    }

    return 0;
}
```



9. Write a C Program to Demonstrate Call be Value and Call by Reference

```
#include <stdio.h>

// Function for Call by Value
void callByValue(int x) {
    x = x + 10;   // Only local copy changed
    printf("Inside callByValue function: x = %d\n", x);
}

// Function for Call by Reference
void callByReference(int *y) {
    *y = *y + 10;  // Actual value changed
    printf("Inside callByReference function: y = %d\n", *y);
}

int main() {
    int a = 5, b = 5;

    printf("Before Call by Value: a = %d\n", a);
    callByValue(a);
    printf("After Call by Value: a = %d\n\n", a);

    printf("Before Call by Reference: b = %d\n", b);
    callByReference(&b);
    printf("After Call by Reference: b = %d\n", b);

    return 0;
}
```

Output

```
Before Call by Value: a = 5
Inside callByValue function: x = 15
After Call by Value: a = 5

Before Call by Reference: b = 5
Inside callByReference function: y = 15
After Call by Reference: b = 15


=== Code Execution Successful ===
```

10. Write a basic C Program which Differentiate while and do while Loop.

```
#include <stdio.h>

int main() {
   int x = 0, y = 0;

   printf("Demonstrating WHILE Loop:\n");
   while (x > 0) {   // Condition is false at the start
      printf("This will NOT be printed.\n");
      x--;
   }
   printf("While loop ended because condition was
false.\n\n");

   printf("Demonstrating DO-WHILE Loop:\n");
   do {
      printf("This line prints at least ONCE even if condition is
false.\n");
      y--;
   } while (y > 0);  // Condition checked after execution

   printf("Do-While loop ended.\n");

   return 0;
}
```

Output

```
Demonstrating WHILE Loop:
While loop ended because condition was false.

Demonstrating DO-WHILE Loop:
This line prints at least ONCE even if condition is false.
Do-While loop ended.


=== Code Execution Successful ===
```