1. Overview of C Programming

Theory

C was developed by Dennis Ritchie at Bell Labs between 1969 and 1973. It evolved from the B language and was crucial for rewriting UNIX. Major versions include K&R C, ANSI C, C99, and C11.

C is widely used due to its speed, hardware control, and portability. It is crucial in systems programming, embedded devices, and influencing languages like C++, Java, and Python.

Lab Exercise

Real-World Applications:

- Operating Systems: Linux Kernel, Windows.
- Embedded Systems: Automotive ECUs, Smart appliances.
- Game Development: Unreal Engine, Doom (1993).

2. Setting Up Environment

Theory

Installing GCC using MinGW, setting environment variables, and installing DevC++ IDE. Running first "Hello, World!" program.

Lab Exercise

Hello World Program:

```
#include <stdio.h>
 main() {
    printf("Hello, World!\n");
```

```
}
```

3. Basic Structure of a C Program

Theory

- **Header Files:** Contain declarations of library functions. Use `#include <stdio.h>` for I/O operations.
- **Main Function:** Entry point defined as `int main() { ... }`. It may return an integer to the operating system.
- **Comments:** Document code without execution. Single-line: `// comment`. Multi-line: `/* comment */`.
- **Declarations:** Specify data types (`int`, `float`, `char`) and allocate memory for variables.
- **Statements & Expressions:** Perform actions using operators and function calls.
- **Return Statement:** `return 0;` signals successful execution.

Lab Exercise

Program with Variables, Constants, and Comments:

```
#include <stdio.h>
 main() {
    int age = 25;
    float price = 99.99;
    char grade = 'A';

    printf("Age: %d\n", age);
    printf("Price: %.2f\n", price);
    printf("Grade: %c\n", grade);

}
```

4. Operators in C

Theory

Operators in C are used to perform operations on variables and values. C has a wide range of operators:

1. Arithmetic Operators

- Used for basic mathematical operations.
- +, -, *, /, %
- Example: sum = a + b;

2. Relational Operators

- Used to compare two values.
- ==, !=, >, <, >=, <=
- Example: a > b

3. Logical Operators

- Used to combine multiple conditions.
- && (AND), || (OR), ! (NOT)
- Example: (a > b) && (a > c)

4. Assignment Operators

- Used to assign values to variables.
- =, +=, -=, *=, /=, %=
- Example: a += 5;

5. Increment/Decrement Operators

- Used to increase or decrease the value by one.
- ++ (Increment), -- (Decrement)
- Example: a++, b--

6. Bitwise Operators

- Used to perform bit-level operations.
- &, |, ^, ~, <<, >>
- Example: a & b

7. Conditional (Ternary) Operator

- Used to replace if-else statement.
- Syntax: (condition) ? true_value : false_value;
- Example: int max = (a > b) ? a : b;

Lab Exercise

Program to Perform Arithmetic, Relational, and Logical Operations:

```
#include <stdio.h>
int main() {
    int a, b;

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Arithmetic Operations
    printf("\nArithmetic Operations:\n");
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n", a % b);

    // Relational Operations
    printf("\nRelational Operations:\n");
```

```
    printf("a == b: %d\n", a == b);
    printf("a != b: %d\n", a != b);
    printf("a > b: %d\n", a > b);
    printf("a < b: %d\n", a < b);
    printf("a >= b: %d\n", a >= b);
    printf("a <= b: %d\n", a <= b);

    // Logical Operations
    printf("\nLogical Operations:\n");
    printf("(a && b) = %d\n", a && b);
    printf("(a || b) = %d\n", a || b);
    printf("!(a == b) = %d\n", !(a == b));

    return 0;
}
```

5. Control Flow Statements

Theory

Control flow statements direct the order of execution of instructions:

1. if Statement: Executes a block if a condition is true.

```
if (condition) {
    // code
}
```

2. if-else Statement: Provides alternative execution when the condition is false.

```
if (condition) {
    // true block
} else {
```

```
    // false block
}
```

3. Nested if-else: An if or else block containing another if-else to handle multiple conditions.

4. switch Statement: Selects a code block among multiple cases based on an integer or character expression.

```
switch (expression) {
    case 1: // code; break;
    case 2: // code; break;
    default: // code;
}
```

Each control structure enhances decision-making in programs

Lab Exercise

Even-Odd Checker and Month Name Display:

```
#include <stdio.h>
 main() {
    int num, month;
    printf("Enter a number to check if it is even or odd:
");
    scanf("%d", &num);

    if (num % 2 == 0) {
        printf("The number %d is even.\n", num);
    } else {
        printf("The number %d is odd.\n", num);
    }

    printf("Enter a number (1-12) to get month name: ");
    scanf("%d", &month);
```

```
    switch(month) {
        case 1: printf("January\n"); break;
        case 2: printf("February\n"); break;
        case 3: printf("March\n"); break;
        case 4: printf("April\n"); break;
        case 5: printf("May\n"); break;
        case 6: printf("June\n"); break;
        case 7: printf("July\n"); break;
        case 8: printf("August\n"); break;
        case 9: printf("September\n"); break;
        case 10: printf("October\n"); break;
        case 11: printf("November\n"); break;
        case 12: printf("December\n"); break;
        default: printf("Invalid input!\n");
    }

}
```

6. Looping in C

Theory

Loops perform repetitive tasks until a condition is met. C provides three loop types:

while Loop: Checks the condition first; suitable when iteration count is unknown.

```
while (condition) {
    // code
}
```

for Loop: Ideal when the number of iterations is known.

```
for (init; condition; update) {
    // code
}
```

do-while Loop: Executes code at least once before checking the condition; useful for menu-driven programs.

```
do {
    // code
} while (condition);
```

Lab Exercise

Using Different Loops:

```
#include <stdio.h>

 main() {
    int i = 1;
    while(i <= 5) {
        printf("%d ", i);
        i++;
    }

    printf("\n");

    for (i = 1; i <= 5; i++) {
        printf("%d ", i);
    }

    printf("\n");

    i = 1;
    do {
        printf("%d ", i);
        i++;
    } while(i <= 5);
```

```
}
```

7. Loop Control Statements

Theory

Modify normal loop flow:

- **break:** Exit loop immediately.
- **continue:** Skip current iteration and move to next.
- **goto:** Jump unconditionally to a labeled statement (use sparingly to avoid spaghetti code).

Lab Exercise

Skip 3 and Stop at 5: `#include <stdio.h>`

```
 main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) break;
        if (i == 3) continue;
        printf("%d\n", i);
    }
}
```

8. Functions in C

Theory

A function is a named block of code that performs a specific task. Functions improve modularity and reuse:

- **Declaration (Prototype):** Informs the compiler about the function's name, return type, and parameters.

```
int add(int, int);
```

- **Definition:** Contains actual code logic.

```
int add(int a, int b) {
    return a + b;
}
```

- **Call:** Executes the function by passing arguments.

```
int result = add(5, 3);
```

Functions can return values or be `void`. Parameters are passed by value; pointers allow pass-by-reference.

Lab Exercise

Factorial using Function:

```
#include <stdio.h>
int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) fact *= i;
    return fact;
}
```

```
int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    printf("Factorial of %d is %d\n", number,
factorial(number));
    return 0;
}
```

9. Arrays in C

Theory

An array is a collection of elements of the same data type stored consecutively in memory. Arrays allow indexed access:

- **1D Arrays:** Linear list.

```
int nums[5] = {1,2,3,4,5};
```

- **2D Arrays:** Matrix with rows and columns.

```
int mat[2][3] = {{1,2,3},{4,5,6}};
```

Arrays provide efficient storage for multiple values and are zero-indexed.

Lab Exercise

1D and 2D Array Program:

```
#include <stdio.h>

 main() {
    int arr1D[5] = {1, 2, 3, 4, 5};
```

```
    printf("1D Array:\n");
    for(int i = 0; i < 5; i++) printf("%d ", arr1D[i]);

    int arr2D[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
    int sum = 0;
    printf("\n2D Array:\n");
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            printf("%d ", arr2D[i][j]);
            sum += arr2D[i][j];
        }
        printf("\n");
    }
    printf("Sum = %d\n", sum);

}
```

10. Pointers in C

Theory

A pointer is a variable that holds the memory address of another variable. Pointers enable dynamic memory management and efficient data handling:

- **Declaration:** `int *p;`
- **Initialization:** `p = &x;` (stores the address of `x`).
- **Dereferencing:** `*p` accesses the value at the pointed address.

Pointers are essential for arrays, functions (pass-by-reference), and dynamic data structures (linked lists).

Lab Exercise

Pointer Example:

```
#include <stdio.h>

 main() {
    int num = 10;
    int *ptr = &num;
    printf("Before: %d\n", *ptr);
    *ptr = 20;
    printf("After: %d\n", *ptr);

}
```

11. Strings in C

Theory

- A string in C is a sequence of characters terminated by a null character (`\0`). Strings are arrays of `char`:
- **Declaration:** `char str[20];`
- **Length:** Use `strlen(str)` from `<string.h>`.
- **Copy:** `strcpy(dest, src)`.
- **Concatenation:** `strcat(dest, src)`.
- **Compare:** `strcmp(s1, s2)`.

Lab Exercise

String Concatenation:

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

```
 main() {
    char str1[100], str2[100];
    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    str1[strcspn(str1, "\n")] = 0;

    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str2[strcspn(str2, "\n")] = 0;

    strcat(str1, str2);
    printf("Concatenated: %s\n", str1);
    printf("Length: %lu\n", strlen(str1));

}
```

12. Structures in C

Theory

A structure (`struct`) is a user-defined composite data type grouping variables of different types under a single name. Structures improve organization of complex data:

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

- **Declaration:** Defines the structure template.
- **Instantiation:** `struct Student s1;`
- **Initialization:** `struct Student s1 = {1, "Alice", 95.0};`
- **Access:** Use the dot operator: `s1.id`, `s1.name`.

Structures can be passed to functions or stored in arrays for record management

Lab Exercise

Student Structure Program:

```
#include <stdio.h>
struct Student {
    char name[50];
    int rollNumber;
    float marks;
};

 main() {
    struct Student students[3];
    for(int i=0;i<3;i++){
        printf("Enter details for student %d:\n",i+1);
        printf("Name: ");

fgets(students[i].name,sizeof(students[i].name),stdin);
        students[i].name[strcspn(students[i].name,"\n")] =
0;
        printf("Roll Number: ");
        scanf("%d",&students[i].rollNumber);
        printf("Marks: ");
        scanf("%f",&students[i].marks);
        getchar();
    }

    for(int i=0;i<3;i++){
        printf("\nStudent %d\n",i+1);
        printf("Name: %s\n",students[i].name);
        printf("Roll Number: %d\n",students[i].rollNumber);
        printf("Marks: %.2f\n",students[i].marks);
    }
```

```
}
```

13. File Handling in C

Theory

Opening, closing, reading, and writing files.

Lab Exercise

File Handling Program:

```
#include <stdio.h>

 main() {
    FILE *file = fopen("example.txt", "w");
    if(file == NULL){
        printf("Error opening file\n");
        return 1;
    }
    fprintf(file,"Hello, this is a test string.\n");
    fclose(file);

    file = fopen("example.txt", "r");
    if(file == NULL){
        printf("Error opening file\n");
        return 1;
    }
    char buffer[256];
    while(fgets(buffer, sizeof(buffer), file)){
        printf("%s", buffer);
    }
    fclose(file);
```

```
}
```