C Programming Course

Module 1: Introduction to C Programming

1.1 What is C?

C is a general-purpose, high-level programming language designed for system programming. It provides low-level memory access, making it suitable for applications that require efficiency, such as operating systems, embedded systems, and compilers.

Example use cases:

- Operating systems (Unix/Linux)
- Compilers
- Embedded systems programming

1.2 Setting Up the Development Environment

To write and execute C programs, you need a **C compiler** and a development environment.

Install GCC on Linux:

```
sudo apt update
sudo apt install build-essential
Install GCC on Windows:
Download MinGW and set it up in your system's PATH.
```

1.3 Basic Structure of a C Program

A C program typically contains:

Header files for libraries Main function where execution begins Statements that define the program logic

Example:

```c
#include <stdio.h>  // Header file

int main() {  // Main function, program starts here
    printf("Hello, World!\n");  // Print statement
```

```
    return 0;  // Exit status
}
```

Module 2: Variables and Data Types

2.1 Introduction to Variables

A variable is a storage location identified by a name, used to hold data values.

Example:

```
int age = 25;  // Integer variable
float height = 5.9;  // Floating-point variable
char gender = 'M';  // Character variable
```

2.2 Data Types in C

C supports several built-in data types:

- int: Integer
- float: Floating point number
- double: Double precision floating point number
- char: Single character

Example:

```
int a = 10;  // Integer
float pi = 3.14;  // Float
char letter = 'A';  // Character
```

2.3 Constants and Literals

Constants are fixed values that do not change during program execution. Use const to define constants.

Example:

```
const float PI = 3.14159;  // Constant value for Pi
```

2.4 Input and Output Functions

printf(): Prints output to the console. scanf(): Takes input from the user.

Example:

```c
int age;
printf("Enter your age: ");
scanf("%d", &age);  // Taking user input
printf("Your age is: %d\n", age);  // Displaying the age
```

Module 3: Operators

3.1 Arithmetic Operators

C supports basic arithmetic operations:

- +: Addition
- -: Subtraction
- *: Multiplication
- /: Division
- %: Modulo (remainder)

Example:

```c
int a = 10, b = 3;
int sum = a + b;  // sum = 13
int diff = a - b;  // diff = 7
int prod = a * b;  // prod = 30
int div = a / b;   // div = 3
int mod = a % b;   // mod = 1 (remainder)
```

3.2 Relational Operators

Relational operators compare two values and return a boolean result (0 or 1):

- ==: Equal to
- !=: Not equal to
- > *: Greater than*

- <: Less than
- =: Greater than or equal to
- <=: Less than or equal to

Example:

```
int x = 10, y = 5;
if (x > y) {
    printf("x is greater than y\n");  // x is greater than y
}
```

3.3 Logical Operators

Logical operators combine multiple conditions:

- &&: Logical AND
- ||: Logical OR
- !: Logical NOT

Example:

```
int a = 5, b = 10;
if (a > 0 && b < 20) {
    printf("Both conditions are true\n");
}
```

3.4 Bitwise Operators

Bitwise operators operate on the bits of numbers:

- &: AND
- |: OR
- ^: XOR
- ~: NOT
- <<: Left shift
- : Right shift

Example:

```
int x = 5;  // 0101 in binary
int y = 3;  // 0011 in binary
int result = x & y;  // 0001 (bitwise AND)
printf("%d\n", result);  // Prints 1
```

3.5 Assignment Operators

Assignment operators assign values to variables. Examples:

- =: Simple assignment
- +=: Addition assignment
- -=: Subtraction assignment

Example:

```
int a = 5;
a += 10;  // a = a + 10, so a becomes 15
a *= 2;   // a = a * 2, so a becomes 30
```

3.6 Other Operators

Increment/Decrement Operators: ++ and -- Ternary Operator: Shorthand for if-else

Example:

```
int num = 5;
num++;  // num becomes 6
```

Module 4: Control Structures

4.1 Conditional Statements

- if: Executes a block if the condition is true.
- else if: Executes a block if the previous conditions are false.
- else: Executes a block if all previous conditions are false.

Example:

```c
int age = 18;
if (age >= 18) {
    printf("Adult\n");
} else {
    printf("Minor\n");
}
```

4.2 Loops

Loops are used to repeat a block of code:

- for loop: When the number of iterations is known.
- while loop: When the number of iterations is unknown but a condition must be met.
- do-while loop: Executes the block at least once before checking the condition.

Example:

```c
for (int i = 0; i < 5; i++) {
    printf("%d\n", i);  // Prints 0 to 4
}
```

4.3 Program Flow Control

- break: Exits a loop.
- continue: Skips the current iteration and continues with the next iteration.

Example:

```c
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;  // Skips 3
    }
    printf("%d\n", i);
}
```

Module 5: Functions in C

5.1 Introduction to Functions

Functions are reusable blocks of code. They can take inputs (parameters) and return outputs (return values).

Example:

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

int main() {
    printf("%d\n", add(3, 4));  // Prints 7
}
```

5.2 Function Types

Functions can return different data types:

- int, float, void (no return value).

Example:

```
void printMessage() {
    printf("Hello, World!\n");
}

int main() {
    printMessage();  // Calling a void function
}
```

5.3 Scope and Lifetime

Local variables: Declared inside functions. Global variables: Declared outside functions.

Example:

```
int globalVar = 10;  // Global variable

void printGlobalVar() {
    printf("%d\n", globalVar);  // Access global variable
}

int main() {
    printGlobalVar();
}
```

5.4 Recursion

Recursion is when a function calls itself.

Example (Factorial of a number):

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    printf("%d\n", factorial(5));  // Prints 120
}
```

Module 6: Arrays

6.1 Introduction to Arrays

An array is a collection of variables of the same type stored in contiguous memory locations.

Example:

```
int arr[5] = {1, 2, 3, 4, 5};
printf("%d\n", arr[2]);  // Prints 3 (index starts at 0)
```

6.2 Multi-dimensional Arrays

Arrays with more than one dimension (e.g., matrices).

Example:

```
int matrix[2][2] = {{1, 2}, {3, 4}};
printf("%d\n", matrix[1][0]);  // Prints 3
```

6.3 Array Operations

Arrays can be traversed and manipulated using loops.

Example:

```
int arr[] = {1, 2, 3, 4};
for (int i = 0; i < 4; i++) {
    printf("%d ", arr[i]);  // Prints 1 2 3 4
}
```

Module 7: Strings

7.1 Introduction to Strings

Strings in C are arrays of characters terminated by a null character ('\0').

Example:

```
char str[] = "Hello, World!";
printf("%s\n", str);  // Prints "Hello, World!"
```

7.2 String Functions

Common string functions:

- strlen(): Returns the length of a string.
- strcpy(): Copies one string to another.
- strcmp(): Compares two strings.

Example:

```
char str1[] = "Hello";
char str2[] = "World";
if (strcmp(str1, str2) == 0) {
    printf("Strings are equal.\n");
} else {
    printf("Strings are different.\n");
}
```

Module 8: Pointers

8.1 Introduction to Pointers

What is a Pointer?

A pointer in C is a variable that stores the **memory address** of another variable. Instead of holding a data value, a pointer holds the address where the actual data is stored in memory.

- A pointer can point to any data type: integer, float, character, etc.
- It is declared using the asterisk (*) symbol, indicating that the variable is a pointer.

Example:

```
int x = 10;   // Regular integer variable
int *p = &x;  // Pointer variable p, holding the address of x
```

In this example, p is a pointer to an integer, and &x represents the address of variable x.

Pointer Declaration and Initialization A pointer is declared by using the asterisk (*) symbol followed by the variable name. Pointers can be initialized to NULL (no address) or with the address of an existing variable using the address-of operator (&).

Example:

```
int a = 5;
int *p;      // Pointer declaration
p = &a;      // Pointer initialization (storing address of a)
```

Dereferencing Pointers (*)

- Dereferencing a pointer means accessing the value at the address stored in the pointer. - This is done using the asterisk (*) symbol.

Example:

```
int a = 5;
int *p = &a;  // p points to the address of a
printf("%d\n", *p);  // Dereferencing p to print the value of a
(output: 5)
```

Pointer Arithmetic

Pointer arithmetic allows you to manipulate the memory address stored in a pointer. You can perform operations like incrementing (p++) or decrementing (p--) the pointer. These operations adjust the pointer by the size of the type it points to.

For example, if p points to an integer (int), incrementing the pointer (p++) will move it by sizeof(int) bytes.

Example:

```
int arr[] = {10, 20, 30};
int *p = arr;  // p points to the first element of arr

printf("%d\n", *p);    // Output: 10 (first element)
p++;                   // Move to next element
printf("%d\n", *p);    // Output: 20 (second element)
```

8.2 Pointers and Arrays

Array Names as Pointers

In C, the name of an array is essentially a pointer to the first element of the array. This means you can use pointers to manipulate arrays.

Example:

```
int arr[] = {1, 2, 3};
int *ptr = arr;  // ptr points to the first element of arr
printf("%d\n", *ptr);  // Output: 1
```

Pointer to an Array

A pointer to an array is different from a pointer to an individual array element. A pointer to an array holds the address of the entire array.

Example:

```
int arr[3] = {1, 2, 3};
int (\*p)[3] = &arr; // p points to the entire array arr

printf("%d\n", (\*p)[1]); // Output: 2 (second element of the
array)
```

Multidimensional Arrays with Pointers

Pointers can be used with multidimensional arrays, where a pointer to an array can point to the entire array, or you can use pointer arithmetic to access elements.

Example:

```
int matrix[2][2] = {{1, 2}, {3, 4}};
int (\*ptr)[2] = matrix; // Pointer to a 2-element array

printf("%d\n", ptr[1][0]); // Output: 3 (second row, first
element)
```

8.3 Pointers to Functions

Function Pointers

A function pointer is a pointer that points to a function instead of data. It can be used to call functions dynamically, allowing for more flexible code, such as implementing callbacks.

Example:

```
#include <stdio.h>

void greet() {
printf("Hello, world!\n");
}

int main() {
void (\*func_ptr)() = greet; // Function pointer initialization
func_ptr(); // Calling the function via the pointer
return 0;
}
```

Passing Functions as Arguments

You can pass function pointers as arguments to other functions. This is particularly useful for callback functions or implementing event-driven programming.

Example:

```
#include <stdio.h>

void printMessage(void (\*message)()) {
message(); // Call function via the pointer
}

void hello() {
printf("Hello from function pointer!\n");
}
```

```
int main() {
printMessage(hello); // Passing function hello as an argument
return 0;
}
```

8.4 Dynamic Memory Allocation

- Using malloc(), calloc(), realloc(), and free()

Dynamic memory allocation allows you to allocate memory at runtime using the following functions:

- malloc(size): Allocates uninitialized memory.
- calloc(num, size): Allocates zero-initialized memory.
- realloc(ptr, size): Resizes a previously allocated block.
- free(ptr): Deallocates previously allocated memory.

Example:

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

int main() {
int _arr = (int _)malloc(5 \* sizeof(int)); // Allocate memory
for 5 integers
if (arr == NULL) {
printf("Memory allocation failed!\n");
return 1;
}

    // Assign values
    for (int i = 0; i < 5; i++) {
        arr[i] = i + 1;
    }

    // Print the values
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);  // Free allocated memory
    return 0;

}
```

Memory Leaks and Avoiding Them

Memory leaks occur when dynamically allocated memory is not freed. To avoid memory leaks:

- Always use free() after you are done using dynamically allocated memory.
- Avoid multiple allocations without freeing the previous ones.

Module 9: Structures and Unions

9.1 Introduction to Structures

Defining and Declaring Structures

A structure in C is a user-defined data type that groups related variables of different types into a single unit. It is defined using the struct keyword.

Example:

```
struct Person {
char name[50];
int age;
};

int main() {
struct Person p1 = {"John", 25};
printf("Name: %s, Age: %d\n", p1.name, p1.age); // Output: John
25
return 0;
}
```

Accessing Structure Members

To access the members of a structure, you use the dot (.) operator for an instance of the structure.

Example:

```
struct Person p1 = {"John", 25};
printf("Name: %s\n", p1.name); // Accessing the 'name' member
```

Structure Initialization

Structures can be initialized at the time of declaration or separately.

Example:

```
struct Person p1 = {"John", 25}; // Initialization at declaration
```

9.2 Nested Structures

Structures within Structures Structures can contain other structures as members. This is called nested structures.

Example:

```
struct Address {
char city[30];
char country[30];
};

struct Person {
char name[50];
int age;
struct Address addr; // Nested structure
};

int main() {
struct Person p1 = {"John", 25, {"New York", "USA"}};
printf("Name: %s, City: %s\n", p1.name, p1.addr.city); //
Accessing nested structure
return 0;
}
```

Passing Structures to Functions

Structures can be passed to functions by value or by reference (using pointers).

Example:

```
void printPerson(struct Person p) {
printf("Name: %s, Age: %d\n", p.name, p.age);
}

int main() {
struct Person p1 = {"John", 25};
printPerson(p1); // Pass by value
return 0;
}
```

9.3 Unions

Defining and Using Unions

A union is similar to a structure, but it allows storing different data types in the same memory location. Only one member can hold a value at any given time.

Example:

```
union Data {
int i;
float f;
char str[20];
};

int main() {
union Data data;
data.i = 10;
printf("i: %d\n", data.i); // Output: 10
data.f = 220.5;
printf("f: %.2f\n", data.f); // Output: 220.5
return 0;
}
```

Difference Between Structures and Unions

- Structure: Each member has its own memory location.
- Union: All members share the same memory location.

Module 10: File Handling

10.1 Introduction to File Handling

File Operations: open, read, write, close

File handling allows programs to read and write to files. The functions used include:

- fopen(): Opens a file.
- fclose(): Closes a file.
- fread(), fwrite(): Read and write data to files.

Example:

```
FILE _file = fopen("test.txt", "w");
if (file == NULL) {
printf("Error opening file\n");
return 1;
}
fprintf(file, "Hello, File!\n");
fclose(file);
```

File Pointers and File Stream

In C, files are accessed through file pointers, which are of type FILE _. These pointers are used by various file functions like fopen, fclose, fread, and fwrite.

10.2 Reading and Writing Files

Reading from Files Using fscanf()/fgets()

- fscanf() reads formatted input from a file.
- fgets() reads a line from a file.

Example (Reading):

```
FILE *file = fopen("test.txt", "r");
char str[100];
fgets(str, 100, file);
printf("Read from file: %s", str);
fclose(file);
```

Writing to Files Using fprintf()/fputs()

- fprintf() writes formatted output to a file.
- fputs() writes a string to a file.

Example (Writing):

```
FILE *file = fopen("test.txt", "w");
fprintf(file, "Hello, world!\n");
fputs("This is a test.", file);
fclose(file);
```

10.3 File Operations Functions

- fopen(): Opens a file.
- fclose(): Closes a file.
- fread(): Reads data from a file.
- fwrite(): Writes data to a file.
- fseek(): Moves the file pointer to a specific position.
- ftell(): Returns the current position of the file pointer.

Example (Using fseek() and ftell()):

```
FILE \*file = fopen("test.txt", "r");
fseek(file, 0, SEEK_END); // Move to end of file
long size = ftell(file); // Get the current position (file size)
printf("File size: %ld bytes\n", size);
fclose(file);
```