COMPLETE NOTES ON 'C'

Instagram : @ Curios_ . programer

Telegram : @ Curious_Coder

Written by :

Sai Sumanth (IT Student)

Index

Date
Page

Date
Page

INTRODUCTION TO "C"

* What is C programming language?

→ C programming language, often simply called C, is a powerful and popular programming language.

→ It is widely used for developing a wide range of software applications, from simple programs to complex operating systems.

→ C is considered a "low-level" language, which means it provides direct access to the computer's hardware and memory.

→ This feature makes it highly efficient and allows programmers to have precise control over their programs.

→ C is known for its simplicity and elegance, making it a great choice for beginners to start their programming journey.

→ Learning 'C' helps in understanding the fundamental concepts of programming and builds a strong foundation for learning other languages.

→ It is a versatile language that can be used in various domains, such as system programming, game development, and embedded systems.

1. Introduction to C Programming

1.1 History and Evolution of C programming

→ C was developed in the early 1970's by Dennis Ritchie at Bell Labs.
→ It was initially created as a language to write the Unix operating system.
→ As Unix gained popularity, so did C. In the late 1970's, the first official specification of the C language, called "K&R C", was published by Brian kernighan and Dennis Ritchie.
→ This specification became the foundation for C's widespread use and further development.
→ In the 1980s, the American National Standards Institute (ANSI) standardized the C language, resulting the creation of ANSI C or C89.
→ This standardization ensured portability and compatibility of C programs across different systems.
→ Over the years, C has evolved with the release of new standards like C99 and C11, introducing additional features and improvements to the language.
→ C's simplicity, efficiency, and versatility have contributed to its longevity and continued relevance in modern programming.
→ It remains one of the most widely used programming languages today, forming the basics for many other languages and frameworks.

1.2 Characteristics and Features of C.

C programming language has several characteristics and features that make it highly versatile and widely used.

→ Firstly, C is a structured language, meaning it allows programmers to break down complex problems into smaller, more manageable parts through functions and modules.

→ This makes the code easier to understand and maintain.

→ Secondly, C is a low-level language, providing direct access to hardware resources, such as memory and registers.

→ This gives programmers precise control over their programs' performance and efficiency.

→ Additionally, C supports wide range of data types, allowing programmers to work with integers, floating-point numbers, characters and more.

→ It also provides powerful operators for performing mathematical and logical operations.

→ Another important feature of C is its extensive library functions, which provide pre-written code for common tasks, saving time and effort.

→ C is known for its efficiency, as it allows for direct manipulation of memory and offers features like pointers, which enable advanced memory management.

→ Lastly, C has large community of developers and a vast collection of resources, making it easier to find support and learn from others.

1.3. Advantages and Disadvantages of C

1.3.1 - Advantages of C

1. **Efficiency :** C is known for its efficiency in terms of memory usage and execution speed. It allows for low-level manipulation of hardware, resulting in fast and optimized programs.
2. **Portability:** C programs can be easily ported to different platforms and operating systems, making them highly versatile.
3. **Wide Range of Applications:** C is widely used in various domains, including system programming, game development, embedded systems and more. Its versatility makes it a valuble skill for programmers.
4. **Powerful and Flexible:** C provides a wide range of features and powerful tools, such as pointers and libraries, which allow for advanced programming techniques and flexibility.
5. **Large Community and Resources:** C has a vast community of developers, making it easier to find support, libraries, and learning resources.

1.3.2. Disadvantages of C

1. **Steeper Learning Curve :** Compared to some high-level languages, C canbe more challenging for beginners due to its low-level nature and complex concepts like pointers.
2. **Lack of Built-in Safety Features:** C does not have built-in safety features like automatic

memory management or bounds checking. which means programmers need to be careful with memory allocation and error handling to avoid bugs and crashes.

3. Verbosity: C can be verbose. meaning it requires writing more code to acheive certain tasks compared to high-level languages. This can make programs longer and potentially more prone to errors.

4. Lack of Object-Oriented Programming (OOP) Support: C is a procedural language and does not have native support for OOP concepts like classes & inheritance. However, it is still possible to implement OOP-like structures in C with manual coding.

5. Limited Standard Library: The standard library in C is relatively small compared to some other programming languages. This means that the developers may need to rely on external libraries for certain functionalities.

1.4 Structure of a C program.

A C program is composed of various elements, including comments, preprocessor directives, function declarations and the main function. Let's break down the C program step by step using the "Hello world" example.

1. Comments: Comments are used to provide explanatory notes within the code. They are not executed by the compiler but serve as useful information for programmers. In C, comments can be single-line (denoted by //) or multi-line (enclosed between /* and */).

2. Preprocessor Directives: Preprocessor directives begin with a hash symbol (#) and are processed by the preprocessor before compilation. They are used to include header files, define constants, or perform other preprocessing tasks. The most common directive is #include, which allows us to include standard libraries or user-defined header files.

3. Main Function: Every C program must have a main function, which serves as the entry point for program's execution. The main function has a specific format: int main(). It returns an integer value indicating the program's status to the operating system. Within the main function, we write the program's logic or a sequence of instructions.

4. The "Hello World" Example: Here's the "Hello world" program written in C:

```
#include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}
```

Explanation:

→ The #include directive includes the standard input/output header file, which contains the declaration of the printf function.

→ The main function is declared as int main() and serves as starting point of the program.

→ The printf function is used to print the text "Hello, World!" to the console.

→ The return statement with the value 0 indicates that the program executed successfully.

1.5. Basic Input/output Operations.

In programming, input/output (I/O) operations are essential for interacting with users and exchanging data with the computer. In this blog post, we will explore the basic input/output operations in C programming language. Understanding how to read input from the user and display output is crucial for building interactive and useful programs. Let's dive in and learn how to perform these operations in C!

1. Printing Output: In C, we use the printf() function to display output on the screen. It allows us to print text, numbers, and other types. The general syntax for printf() is:

```
printf("Format String", variables);
```

The format string contains placeholders that are replaced by the corresponding variables. For example, to print a simple message, we can use:

```
printf("Hello, World!");
```

2. Reading Input: To receive input from the user, we use the scanf() function. It allows us to read different types of data, such as integers, floating-point numbers and characters. The syntax for scanf() is:

```
Scanf("Format String", &variables);
```

The format string specifies the type and format of the input, and the ampersand(&) is used to indicate the memory location where the input is stored. For example, to read an integer from the user, we can use:

```
int num;
scanf("%d", &num);
```

3. Using Escape Sequences: Escape sequences are special characters that have specific meanings in C. They are used to represent non-printable characters or perform certain actions. For instance:

→ "\n" represents a newline character and is used to move the cursor to the nextline.
→ "\t" represents a tab character and is used for indentation.
→ "\\" represents a backslash character when you need to print a literal backslash.

* Here's a simple program that demonstrates both input and output operations:

```
#include <stdio.h>
int main() {
  int age;
  printf("Enter your age: ");
  scanf("%d", &age);
  printf("Your age is %d\n", age);
  return 0;
}
```

→ In this program, we prompt the user to enter their age using printf(). Then, we use scanf() to read the user's input and store it in the variable "age". Finally, we display the age using printf() along with the entered value.

classmate
Date
Page

2. Data Types and Variables.

2.1. Fundamental Datatypes in C.

In C programming language, datatypes define the nature of variables and determine the kind of data they can store. Each type has its characteristics and range of values.

1. Integer Types:
   - Char : It represents a single character and can store values from -128 to 127 (or) 0 to 255 (if unsigned).
   - int : It is used to store integers and has a range of -32,768 to 32,767.
   - Short : It stores small integers with a range of -32,768 to 32,767.
   - long : It can store larger integers with a range of -2,147,483,648 to 2,147,483,647.

2. Floating-point types:
   - float : It represents single-precision floating-point numbers with a range of approximately ±3.4e-38 to ±3.4e+38.
   - double : It represents double-precision floating-point numbers with a range of approximately ±1.7e-308 to ±1.7e+308.

3. Other Data Types:
   - void : It denotes the absence of any type and is often used as a return type for functions that do not return value.
   - Bool : It represents a Boolean value, which can be either true (1) or false (0).

2.2 Declaring and Initializing Variables.

In C programming, variables are used to store and manipulate data. Before using a variable, it needs to be declared and optionally initialized.

Declaring Variables: To declare a variable, we need to specify its data type and a name that represents the variable. The general syntax for a variable declaration is,

```
data_type variable_name;
```

Eg:

```
int age;        → variable_name.
float height;
char initial;
```

Initializing Variables: Variable intialization is the process of assigning an initial value to a variable at the time of declaration. This ensures that the variable has a defined value before it is used in the program. This initialization can be done using the assignment operator (=). The general syntax for variable initialization is,

```
data_type variable_name = initial_value;
```

Eg:

```
int   age = 25;      → variable_name, initial_value.
float height = 1.75;
char  initial = 'J';
```

→ It is important to note that the variables must be initialized before they are used.

Date
Page

2.3 Type Modifiers.

In C programming, type modifiers are used to modify the behaviour and properties of data types. They allow us to fine-tune how variables are stored and used in memory.

1. Signed and Unsigned:
   - The "signed" modifier is used with the integer data types to allow both positive and negative values. It is the default behaviour if the "signed" keyword is not explicitly used.
   - The "unsigned" modifier is used with integer data types to allow non-negative values. It extends the range of positive values that can be stored in a variable but eliminates the ability to represent negative values.

2. Short and long:
   - The "short" modifier is used with integer data types to allocate less memory, resulting in a smaller range of values that can be stored. It typically uses half the memory compared to the default integer type.
   - The "long" modifier is used with integer data types to allocate more memory, allowing for a larger range of values. It typically uses double the memory compared to the default integer type.

3. Type Qualifiers:

- The "const" qualifier is used to declare a variable as constant, meaning its value cannot be changed once assigned.
- The "volatile" qualifier is used to indicate that a variable's value can be changed by external factors that are not with in the control of the program. This is commonly used for variables that can be modified by hard ware or other concurrent resources/processes.

Eg:

```
unsigned int count = 10;
short int temparature = -5;
long int population = 1000000L
const float PI = 3.14159;
volatile int sensorValue;
```

8.4. Constants and Literals.

In C programming Language, Constants and literals are used to represent fixed values that do not change during the execution of a program. They provide a way to store and refer to unchanging data throughout the code.

Constants in C:

Constants are fixed values that cannot be modified once assigned. They are created using the "const" keyword, which ensures that their values remain constant throughout the program. Constants

are typically used to store values that are known and will not change during program execution, such as mathematical constants or configuration values.

Eg:

```
Const float  PI = 3.14159;
Const int    MAX_value = 100;
```

Literals in C:

Literals are specific values that are directly written in the code. They represent fixed data and can be of various types, such as integers, floating-point numbers, characters, or strings. Literals are used to provide immediate values for calculations or to initialize variables.

Eg:

```
int   score = 85;
float pivalue = 3.14;
char  grade = 'A';
char  message[] = "Hello, world!";
```

In the above example, we assigned literals to variables. The literal values, such as 85, 3.14, 'A', "Hello, world!", are directly written in the code and represent fixed data.

2.5. Operators and Expressions.

In C, Operators and expressions form the foundation for performing computations and manipulating data. Understanding how operators works & how expressions are constructed is crucial for writing effective and functional code.

Operators in C:

Operators are symbols that perform specific operations on one or more operands. They allow us to perform mathematical calculations, make comparisions, and manipulate data.

1. Arithmetic Operators:

Arithmetic operators are used to perform basic mathematical operations.

- Addition (+) : Adds two operands.
- Subraction (-): Subracts one operand from another.
- Multiplication(*): Multiplies 2 operands.
- Division (/) : Divides one operand by another.
- Modulo(%) : Calculates the remainder.

2. Relational Operators:

Used to compare values & determine relation-ships:

- Equality (==) : Checks if 2 operands are equal.
- Inequality (!=) : Checks if 2 operands are not equal.
- Greater than (>) or Greater than or equal to (>=): Checks if operand is greater or equal to the another.
- Less than(<) or Less than or equal to (<=): Checks if one operand is lesser or equal to the another.

3. Logical Operators:

Used to perform logical operations on Boolean values.

- AND(&&) : Returns true if both are true.
- OR(||) : Returns true if atleast one of the operands is true.

- NOT (!) : Reverses the logical state of an operand.

Expressions in C:

Expressions are combination of variables, constants, literals, and operators that produce a value. They are used to perform computations and assign results to variables.

Eg:

```
int a=10, b=5, c=7;
int result = (a+b)*c;
```

2.6. Type Casting.

In C language, typecasting allows you to convert a value from one data type to another.

Type Casting involves explicitly specifying the desired data type for a value or variable. It allows you to change the interpretation of data without altering its underlying representation. By typecasting, you can ensure that data is treated appropriately, during operations and assignments.

Syntax:

```
(type) value
```

Eg:

```
int a = 10;
float b = 3.14;
int result;
result = (int) b;    // Typecasting float to int
printf ("%d\n", result); // output = 3
```

```
result = (int) a/3; // Typecasting float to int a/3
printf("%d\n", result); // Output = 3.
```

→

Use Cases:-

1. When you need to perform arithmetic operations involving different data types.

2. When you want to assign a value of one data type to a variable of a different data type.

3. When you want to ensure compatibility and prevent loss of data during assignments or calculations.

It's important to note that typecasting may result in loss of precision or truncation of data, so it should be used with caution.

3. Control Flow Statements.

3.1 Decision-making Statements.

In C language, decision-making statements are used to control the flow of execution based on certain conditions. These statements allow your program to make decisions and choose different paths to follow.

1. if - Statement:

The "if" statement allows you to execute a block of code if a specified condition is true. If the condition is false, the code block is skipped. The general syntax is,

```
if(condition)
{
    // code to be executed if the condition true.
}
```

Eg:

```
int age = 25;
if (age >= 18)        // (25 >= 18) → satisfied
{
    printf("You are eligible to vote.\n"); //printed
}
```

2. if-else Statement:

The "if-else" statement allows you to execute one block of code if a condition is true and another block of code if the condition is false. The general syntax is:

```
if (condition)
{
}
else
{ }
```

Eg:-

```
int marks = 85;
if (marks >=80)  //(85 >=80) → satisfied
{
    printf("you passed the exam.\n");//executed
}
else
{
    printf("you failed the exam.\n");
}.
```

3. Nested if-else Statement:

The nested if-else statement allows you to have multiple levels of if-else statements within each other. This allows for more complex decision-making scenarios. The general syntax is,

```
if (condition1)
{   // code
  if (condition2)
  {
    // code
  }
 else
 {
   //code
 }
}
else
{
    // code
}
```

Eg:

```
int num = 10;
if (num > 0)      // (10>0) → satisfied
{
    printf ("Number is positive."); //executed.
    if (num % 2 == 0)   (10%2==0) → true
    {
        printf("It is also even.\n"); //executed
    }
    else
    {     printf("It is odd.\n");
    }
}
else
{
    printf ("Number is non-positive.\n");
}
```

4. Switch Statement:

The "switch" statement allows you to select one of many code blocks to execute based on the value of the expression. It provides a more consise way to handle multiple conditions. The general syntax is:

```
Switch (condition/expression)
{
    case value1 : //code executable
                  break;
    Case value2 : // code executable
                  break;
    // Additional cases.....
    default:  // code executable
    break;
}
```

Eg:

```
#include <stdio.h>
int main()
{
    int n;
    scanf("%d", &n);
    switch(n)
    {
        case 1 : printf("Addition\n");
                 break;
        case 2 : printf("Subtraction\n");
                 break;
        case 3 : printf("Multiplication\n");
                 break;
        case 4 : printf("division\n");
                 break;
        default : printf("Invalid input");
                 break;
    }
    return 0;
}
```

Output :

1 → I^st execution & input given
Addition → Output.

4 → II^nd execution & input given to
division → Output.

6 → III^rd execution & input given to
invalid input. → Output.

note: If the any of the input is not in the specified case then it will directly goes to "default" case.

3.2 Looping Statements

In C programming language, looping statements are used to repeat a block of code multiple times. They allow you to automate repetitive tasks and iterative or iterate over data structures.

1. for Loop :

The "for" loop is used when you know the number of times you want to repeat a code block. It consists of 3 parts : Initialization, condition, and increment / decrement. The general syntax is :

```
for (initialization ; condition ; increment/decrement)
{
    // code to be executed repeatedly
}
```

Ex :

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

Output :

```
1
2
3
4
5
```

→ As the "i" initialized with "1" & "i" is as long as less than or equal to "5" it will print value of "i". In each iteration "i" is increased by "1".

2. While Loop:

The "while" loop is used when you want to repeat a code block as long as a specific condition is true. The condition is checked before each iteration. The general syntax is:

```
while (condition)
{
    // code to be executed repeatedly
}
```

Ex:

```
int count = 1;              //initialization
while(count <= 5)           //condition check.
{
    printf ("%d ", count);  //output
    count++;                //increment
}
```

Output:

1 2 3 4 5.

3. do-while Loop:

The "do-while" loop is similar to the while loop but with a slight difference. It executes the code block at least once before checking the condition.

→ The condition is checked at the end of each iteration. The general syntax:

```
do
{
    // code executed repeatedly.
} while(condition);
```

Ex:

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

→ the do-while loop will execute the block at least once and then continue to execute it as long as the "count" variable is less than or equal to 5.

Output:

1 2 3 4 5

3.3 Jump Statements.

In C programming, jump statements allow you to control the flow of your program by altering the normal sequence of execution.

1. Break Statement:

The "break" statement is used to immediately exit a loop or switch statement. when encountered, it terminates the current loop or switch block and transfers control to the next statement after the loop or switch.

→ It helps in ending the execution permanently or prematurely. It is often used to exit a loop based on certain conditions.

Ex: Code:

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

output:

1 2.

→ when value of "i" becomes 3, break statement encountered and the loop is terminated. As a result in the output only "1" and "2" will be printed.

2. Continue Statement:

The "continue" statement is used to skip the current iteration of a loop and move to the next iteration. When encountered, it bypasses the remaining code within the loop block for the current iteration and proceeds to next iteration.

→ It is often used to skip certain iterations based on specific conditions.

Ex:

```
for(int i=1 ; i<=5 ; i++)
{
    if (i==3)
        Continue;
    printf("%d ", i);
}
```

Output :

1 2 4 5

→ As when the value of i is 3 then that iteration will be skipped & goes to next iteration.

3. goto Statement :

The "goto" statement is used to transfer control to a labelled statement within the same function.

→ It allows you to jump to a specific point in your code, which can be helpful in certain situations.

Ex :

```
int age;
Start:
printf ("Enter your age : ");
Scanf ("%d", &age);
if (age<0)
    goto start;
printf ("Your age is %d\n", age);
```

→ In the above example, the program prompts the user to enter their age.

→ If negative value is entered, the goto statement is used to jump back to the "Start" label, allowing the user to re-enter a valid age.

4. Arrays and Strings.

4.1 Introduction to Arrays

→ Arrays are an essential data structure in C programming language, They allow you to store multiple values of same datatype in a single variable.

→ Arrays have a fixed size, meaning you need to specify the number of elements they can hold declaring them.

→ They provide an efficient way to manage and manipulate a group of related data.

→ They play a crucial role in tasks like storing a list of names, recording scores, or managing large sets of data efficiently.

→ with arrays, you can work with multiple values conveniently and streamline your code.

4.2 One-dimensional and Multi-dimensional arrays

In C programming language, arrays can be classified as one-dimensional or multi-dimensional depending upon the number of dimensions they have.

1. One-dimensional Arrays:

A one-dimensional array is a

linear collection of elements of the same data type, It is like a simple list of values stored in a single row.

→ Each element in the array is accessed using its index, which starts from 0.

→ One-dimensional arrays are commonly used to store a sequence of values, such as a list of numbers, names or grades.

→ They provide a convenient way to organize and access data in a linear manner.

Eg: int a[5]; // declaration.
a[5] = {1, 5, 4, 15, 20}; // intialization

2. Multi-Dimensional Arrays:

A multi-dimensional array is an extension of the one-dimensional array, but it has two or more dimensions.

→ It is like a table or grid of elements arranged in rows and columns.

→ Each element in a multi-dimensional array is accessed using multiple indices.

→ Multi-dimensional arrays are commonly used to represent matrices, tables or other structured data.

Eg: int a[2][2];
a[2][2] = [[1,2],[3,4]];

→ Arrays, both one-dimensional and multi-dimensional, are powerful tools in C programming.

→ The number of dimensions in an array determines the complexity of organizing and accessing data.

4.3 Array Initialization and Manipulation.

Arrays are a fundamental data structure in C that allow you to store and manipulate collections of data.

Now, we will discuss initialize arrays, modify array elements, and perform common operation on arrays.

Array Initialization:

To initialize an array, you assign values to its elements during declaration or after declaration using assignment statements.

→ During declaration, you can provide the initial values enclosed in curly braces{} within the square brackets [].

Ex:

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

Accessing Array Elements:

Array elements are accessed using their indices, starting from 0. To retrive

the value of an element, you use the array name followed by the index in square brackets.

Ex:-

```
int thirdNumbers = numbers[2];
```

→ In the above example, we access the 3rd element of the "numbers" array and assign it to the variable "thirdNumber".

Modifying Array Elements:-

Array elements can be modified by assigning new values to them using their indices.

Ex:-

```
numbers[1] = 10;
```

→ In the above example, we change the value of the second element of the "numbers" array to 10.

Common Array Operations:-

Arrays support various operations, such as finding the length of an array, traversing array elements using loops, and performing calculations on array elements.

Ex:-

```
int length = sizeof(numbers)/sizeof(numbers[0]);

for(int i=0 ; i<length ; i++)
{
    printf("%d ", numbers[i]);
}
```

→ In the above example, "length" is getting the size of the "numbers" array. We then use a for-loop to traverse the array and print each element.

4.4 Introduction to Strings.

→ In C programming language, strings are used to store and manipulate sequence of characters.

→ Strings are essential for working with textual data like names, sequences, or any other collection of characters.

→ In C, strings are represented as arrays of characters terminated by a null character '\0'. The null character indicates the end of the string.

Ex:-

```
Chare greeting[10] = "Hello"; //declaration & initialization.
```

→ we declared string "greeting" and initialize with the value "Hello". The size of the array should be large enough to accomdate the characters in the string, including the null character.

→ Strings are widely used in programming, particularly when dealing with textual data.

→ They provide a convinient and efficient way to work with words, sentences, and other

Character Sequences.

4.5 String Manipulation Functions

→ In C programming language, string manipulation functions are used to perform various operations on strings.

→ These functions provide convenient ways to manipulate and process textual data.

1. Strlen():

Syntax:

```
size_t strlen(const char* str)
```

→ The strlen() function calculates the length of a string by counting the number of characters until it reaches the null character '\0'.

→ It returns the length of the string as a size_t (unsigned integer) value.

2. Strcpy():

Syntax:

```
char* strcpy(char* destination, const char* source)
```

→ The strcpy() function is used to copy the contents of one string to another.

→ It takes destination string and source string as arguments.

→ The function copies the characters from the source string to the destination string, including the null character '\0'.

→ It returns pointer to the destination string.

3. Strcat():

Syntax:

```
char* strcat(char* destination, const char* source);
```

→ The strcat() function concatenates (joins) two strings together.

→ It appends the characters from the source string to the end of the destination string, overwriting the null character of the destination string.

→ The function returns a pointer to the destination string.

4. Strcmp():

Syntax:

```
int strcmp(const char* str1, const char* str2);
```

→ The strcmp() function compares two strings lexicographically.

→ It returns an integer value that indicates the relationship between the strings.

→ values : 0 if the strings are equal.
-1 if first string is less than 2nd string.
+1 if 1st string is greater than 2nd

string.

5. strstr():

Syntax:

```
char * strstr (const char* haystack, const char* needle)
```

→ The strstr() function searches for the first occurance of a substring within a string.

→ It takes haystack string (larger string to search within) and the needle string (the substring to search for).

→ It returns a pointer to the first occurance of the needle string within the haystack string.

→ It returns null pointer, if the substring is not found.

5. Functions.

5.1 Defining and Calling Functions.

→ In C programming, function play a crucial role in organizing and modularizing code.

→ A function is a block of code that performs a specific task.

→ It allows you to break down your program into smaller, manageable parts, making it easier to understand, debug and maintain.

→ Defining a functions involves specifying its name, return type, and parameters.

→ The basic syntax of a function definition in C is as follows:-

```
return_type function_name (parameters)
{
    // Function body
    // code to be executed when
    // the function is called.
}
```

→ The 'return_type' is the datatype of the value that the function returns.

→ It can be 'void' if the function doesn't

return any value.

→ The function_name is the identifier used to call the function.

→ To call a function, you simply use its name followed by paranthesis containing any required arguments.

Ex:

```
function_name (argument1, argument2, ....);
```

→ When a function is called, the program execution jumps to the function's definition, executes the code inside the function, and then returns back to where it was called from.

5.2. Function Prototypes.

→ In C programming, function prototypes are used to declare functions before they are defined or called.

→ Function prototypes provide important information about the functions, such as their names, return types and parameter types.

→ This declaration helps the compiler understand the function's signature and how it should be used in the program.

→ The syntax for a function prototype is similar to the function's definition but without the function body.

→ It includes the function's name, return type, and parameter list.

Ex:-

```
int addNumbers(int num1, int num2);
```

→ "addNumbers" is the function name mentioned in the above prototype example.

→ Function prototypes are typically placed at the beginning of a program or in a header file that is included in multiple source files.

→ It informs the compiler about the existence and signature of the function, enabling it to perform necessary checks.

→ The basic syntax:-

```
return_type function_name(parameter1, parameter2);
```

5.3 Return Types and Parameters.

→ The return type of a function determines the type of value it can return to the caller.

→ It can be any valid datatype in C, such as

'int', 'float', 'char', or even user-defined types.

→ Parameters are variables declared within the paranthesis of a function's definition or the prototype.

→ They act as placeholders for the values that are passed to the function when it is called.

→ Functions may have zero or more parameters.

→ parameters enable functions to work with external values and perform operations based on those values.

→ The return type and parameters of a function define its signature.

→ when defining a function, you should carefully choose the appropriate return type and parameters based on the task the function is intended to perform.

5.4 Recursive Functions

→ Recursive functions are functions that call themselves within their own definition.

→ They provide an elegant way to solve problems that can be divided into smaller sub-problems of the same kind.

→ To write a recursive function, you need to define a base case that terminates the recursion and one or more recursive calls that solve the smaller sub-problems.

→ It's important to ensure that the base case is reached to avoid infinite recursion.

Ex:

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

→ In the above example, the "factorial" function calls itself with a smaller value of 'n' until the base case ('n==0') is reached.

→ The factorial of a non-negative integer 'n' is computed by multiplying 'n' with factorial of 'n-1'.

5.5 Function Overloading.

→ In C programming, function overloading refers to the ability to define multiple functions

with the same name but different parameter lists.

→ The compiler determines which function to call based on the number, types, and order of arguments provided during the function call.

→ Overloaded functions provide flexibility and code reusability.

→ They allow you to perform similar operations on different data types without having to define seperate function names for each variation.

Ex:

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

→ In this example, the `add` function is defined twice with different parameter types (`int` and `float`).

→ The compiler selects the appropriate `add` function based on the argument types used in the function call.

→ Function overloading is particularly useful when you want to perform similar operations on different data types or when you need to provide default values for optional arguments.

Ex:

```
float CircleArea = calculateArea(5.0);
float rectangleArea = calculateArea(4.0, 6.0);
float triangleArea = calculateArea(3.0, 8.0);
```

→ In the above calls, the compiler selects the correct version of the function based on the number and type of arguments

→ Function overloading in C allows you to define multiple functions with the same name but different parameter lists.

→ By choosing the appropriate function based on the arguments provided, you can handle various scenario's without creating seperate functions.

6. Pointers.

6.1 Introduction to pointers.

Pointers are powerful and fundamental concepts used to work with memory addresses. A pointer is a variable that holds the memory address of another variable.

→ Pointers are declared using asterisk(*) symbol before the variable name.

Ex:

```
int *ptr; // declares a pointer to an integer.
```

→ Pointers store memory addresses, which represent the location where a variable is stored in the computer's memory.

→ The address-of operator (&) is used to obtain the memory address of a variable.

Ex:

```
int num = 5;
int *ptr = &num;
```

* Assigns the memory address of 'num' to the pointer 'ptr'.

→ pointers allow us to indirectly access and manipulate data by dereferencing them using the dereference operator (*).

Ex:

```
*ptr = 10;
```

* 10 is assigned memory location of ptr.

→ pointers are used for dynamic memory allocation, such as creating arrays and structures at runtime.

→ It is important to initialize pointers before using them to avoid accessing uninitialized or invalid memory addresses.

→ Pointers provide a way to optimize code and efficiently manage memory resources in C programs.

6.2 Pointer Arithmetic.

→ In C programming, pointer arithmetic refers to performing mathematical operations on pointers.

→ pointer arithmetic is a powerful tool that enables efficient manipulation of data structures and arrays.

→ pointer arithmetic is performed on pointers that point to elements of an array or allocated memory.

→ when adding or subracting an integer value to/from a pointer, the result is adjusted based on the size of the data type the pointer points to.

→ pointer arithmetic is closely related to array indexing. In fact, array indexing is a form of pointer arithmetic.

Ex:

arr[2] is equivalent to *(arr+2).

→ pointer arithmetic can be used to iterate over an array or access elements at specific memory locations.

→ Accessing memory beyond the allocated space can result in undefined behaviour and potential crashes.

→ Pointers can be incremented or decremented by any integer value.

Ex:

→ "ptr++" moves the pointer to next memory location.

→ "ptr--" moves it to the previous location.

→ pointer arithmetic can be combined with other operators, such as assignment and conditional operators, to perform more complex operations.

→ Understanding and utilizing pointer arithmetic can greatly enhance your ability to work with data structures and efficiently manage memory in C programs.

6.3 Pointers and Arrays.

Arrays:-

Contiguous blocks of memory that store multiple values.

→ Declaration : int arr[5];

→ Initialization : int arr[] = {1, 2, 3};

→ Accessing elements : arr[index];

→ Array decay : Arrays can decay into pointers to their first elements.

→ Array of pointers : Arrays can hold pointers as their elements.

Pointers:-

Variables that store memory addresses.

Declaration : int *ptr;

Initialization : int *ptr = #

Accessing elements : *ptr (dereferencing).

Address Arithmetic : 'ptr+1' (moves to the next memory location).

Relationship : pointers and arrays are closely related and can be used interchangably.

Pointer to Array :- pointers can point to the first element of array.

⇒ Pointers can be used to access and manipulate array elements.

→ It allows for efficient traversal and manipulation of arrays using pointer arithmetic.

6.4 Pointers to Functions.

→ Pointer to a function refers to a variable that can store the memory address to a function.

1. Function Pointers:- Function pointers are variables that store the memory address of a function.

2. Declaration:- Function pointers are declared by specifying the return type and parameter types of the function they will point to.

Ex:-

```
int (*ptr)(int, int);
```

⇒ declares a function pointer named ptr that points to a function taking 2 integers as arguments & returning an integer.

3. Assignment:-

Function pointers are declared by specifying the return type and parameter types of the function they will point to.

2

→ Function pointers can be assigned the address of

a compatible function using the '&' operator or simply using the function name.

Ex:

```
ptr = &add; (or)
ptr = add;
```

assigns the address of the function 'add' to the function pointer 'ptr'.

4. Calling Functions: To call a function using a function pointer, you can use the dereference operator "*" or simply use function pointer name followed by paranthesis.

Ex:

```
result = (*ptr)(2,3); (or)
result = ptr(2,3);
```

invokes the function pointed to by 'ptr' with arguments 2 and 3.

5. Callback Functions:

These are the functions that are passed as arguments to other functions, allowing dynamic behaviour based on runtime conditions.

6. Dynamism and Flexibility:

Function pointers provide the ability to dynamically choose and invoke functions during runtime, enabling adaptable program behaviour.

7. Function Pointer Arrays:

Function pointers can be stored in arrays, allowing you to create tables

of functions and access them using array indexing.

8. Compatibility:

Function pointers must have compatible signatures with the functions they point to.

→ The return type and parameter types of the function pointer must match the corresponding types of the function it is assigned to.

6.5 Dynamic Memory Allocation.

Dynamic memory allocation allows you to allocate and deallocate memory at runtime, providing flexibility and efficient memory management.

1. malloc :

Syntax:

```
void* malloc(size_t size);
```

→ Allocates a block of memory of the specified size in bytes.

→ Returns a void pointer (void*) pointing to the beginning of the allocated memory.

Ex: `int* ptr = (int*) malloc(10 * sizeof(int));`

allocates memory for an integer array of size 10.

2. Calloc:

Syntax:

```
void* calloc(size_t num, size_t size);
```

→ Allocates block of memory for an array of 'num' elements, each of 'size' bytes.

→ Initializes the allocated memory to zero.

→ Returns void pointer (*void) pointing to the beginning of the allocated memory.

Ex: `int *ptr = (int*) calloc(10, sizeof(int));`
allocates memory for an integer array of size 10 with initialized value as zero.

3. realloc:

Syntax:

```
void * realloc(void* ptr, size_t size);
```

→ Changes the size of the previously allocated memory block pointed to by 'ptr'.

→ If possible, it resizes the block without moving its contents. Otherwise, it allocates a new block and copies the data.

→ Returns a void pointer (*void) pointing to the beginning of the resized or newly allocated memory block.

Ex:

ptr = (int*) realloc (ptr, 20 * sizeof(int));
resizes the previously allocated memory block to accomdate 20 integers.

4. free:

Syntax:

void free (void* ptr);

→ Deallocates the memory block pointed to by 'ptr', making it available for future use.

→ It is important to release the allocated memory to prevent memory leaks.

→ Ex:

free (ptr);
frees the previously allocated memory block.

7. Structures and Unions.

7.1 Defining Structures.

In C programming, structures provide a way to combine different data types into a single entity.

→ A structure is a user-defined data type that allows you to group related variables together.

→ It helps in organizing and managing complex data structures.

→ To define a structure, you specify its name and the variables it contains.

→ Each variable within the structure is called a member.

Ex:

```
Struct person
{
    Char name[50];
    int age;
    float height;
};
```

→ In this example, we defined a structure called 'person' with 3 members : 'name' (an array of characters to store the person's name), 'age' (an integer to store the person's age), and 'height' (a float to store the person's height).

Date
Page

7.2. Accessing Structure Members.

→ Once you have defined a structure, you can create variables of that structure type and access its members using the dot (`.`) operator.

→ The dot operator allows you to access individual members of a structure.

→ Example for creating a variable of the 'person' structure and accessing its members:

```
Struct person p;
strcpy (p.name , "John");
p.age = 25;
p.height = 1.75;
```

→ In this example, we created a variable 'p' of type 'person'.

→ We then assigned values to its members using the dot operator.

→ The 'strcpy' function is used to copy the string "John" into the "name" member.

7.3 Nested Structures.

→ In C, you can also define structures within structures, known as nested structures.

→ This allows you to create more complex data structures by combing multiple structures.

Ex:

```
Struct Date
{
    int day;
    int month;
    int year;
};

Struct Person
{
    char name[50];
    struct Date birthdate;
};
```

→ In this example, we defined a structure called 'Date' to represent a date with day, month, and year.

→ Then, we defined another structure called 'person', which includes the 'name' member and a nested structure 'birthdate' of type 'Date'.

7.4 Structure and Functions.

→ Structures can be passed as arguments to functions and returned from functions, allowing you to operate on complex data structures

Ex:

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

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

→ In this example, we defined a structure 'Person' and function 'printPerson' that takes a 'Person' structure as an argument.

→ The function prints the name and age of the person.

7.5 Introduction to Unions.

→ Unions, like structures, are user-defined data types that allow you to combine different data types.

→ However, unlike structures, unions can only hold one value at a time.

→ The memory allocated for a union is shared among its members.

Ex:-

```
union Shape
{
    int sides;
    float radius;
    double area;
};
```

→ In this example, we defined a union called `Shape'.

→ There are 3 members in the above union shape.

1. sides (integer to represent number of sides).
2. radius (a float to represent the radius)
3. area (a double to represent the area).

→ Only one member of the union can hold a value at a given time.

→ Structures & Unions are powerful tools in C.

8. File Handling.

8.1 File Concepts and Operations

File: A file is a named collection of related data storage medium, such as hard-drive or solid-state drive. It can be a text file or a binary file.

File Operations: C provides several functions for file operations, including opening, reading, writing, closing and manipulating data or files. These functions are part of the standard library, such as `fopen`, `fread`, `fwrite`, `fclose`, etc.

File Pointers: When working with files, you use a file pointer, which is a special variable that represents the file being accessed.

→ The file pointer keeps track of the current position in the file during read and write operations.

→ file pointers allows you to perform reading, writing, opening, closing files operations.

→ A file is a named collection of data stored on a storage device such as a hard disk.

8.8 Opening and Closing Files.

→ Before you can perform any operations on a file, you need to open it.

→ To open a file, you use the 'fopen()' function, which takes two parameters: the file name and the mode in which you want to open the file.

Ex:

```
FILE *filePointer;
filePointer = fopen("example.txt", "r");
```

→ In this example, we declared a file pointer 'filePointer' and used the 'fopen()' function to open the file named "example.txt" in read mode ("r").

→ The 'fopen()' function returns a pointer to the opened file, which we assigned to the file pointer.

→ After performing file operations, it's important to close the file using the 'fclose()' function.

→ Closing the file ensures that any changes made to the file are saved and frees up system resources.

Ex:

```
fclose(file pointer);
```

→ In this example, we used the 'fclose()' function to close the file pointed to by 'filePointer'.

8.3 Reading and Writing Files.

→ Once you have opened file, you can read data from it or write data to it using different file handling functions.

→ To read data from a file, you can use functions like 'fscanf' or 'fgets()'.

→ These functions allow you to read data from the file and store it in variables.

Ex:-

→ reading data from the file using "fgets()"

```
Char buffer[100];
fgets(buffer, 100, filepointer);
```

→ In this example, we used the 'fgets()' function to read a line of text from the file pointed to by 'filepointer' and store it in the character array 'buffer'.

→ To write data to a file, you can use functions like 'fprintf()' or 'fputs'.

→ These functions allow you to write data to

the file using formatted or non-formatted output.

Ex:-

writing data to the file using 'fprintf':-

```
int number = 42;
fprintf(filepointer, "The answer :%d", number);
```

→ In the above example, we used the 'fprintf()' function to write the formatted output "The answer 42" to the file pointed to by 'file pointer'.

8.4 Error Handling in File Operations.

→ When working with files, it's important to handle errors that may occur during file operations.

→ File operations can fail due to various reasons, such as the file not existing, insufficient permissions, or disk errors.

→ To handle errors in file operations, you can check the return value of the file handling functions.

→ Most file handling functions return a value that indicates success or failure.

→ you can use this value to determine if the operation was successful or if an error occurred.

Ex:

error handling when opening a file.

```
FILE *filePointer;
filePointer = fopen("example.txt", "r");
if (filePointer == NULL)
{
    printf("Error opening the file.\n");
    //Handle the error
}
```

→ In this example, we checked if the file pointer returned by 'fopen()' is 'NULL', which indicates that an error occured during file opening.