C PROGRAMMING NOTES

UNIT-I

WHAT IS C PROGRAMMING?

C is a widely-used and influential programming language that was developed in the early 1970s by Dennis Ritchie at Bell Labs. It is known for its simplicity, efficiency, and flexibility, making it suitable for a wide range of applications, from system programming to application development. C has been the foundation for many other programming languages and is still used extensively in various industries.

Key Features of C:

1. Procedural Programming: C is primarily a procedural programming language, which means it follows a step-by-step approach to solve problems. It involves breaking down the program into functions or procedures that perform specific tasks.

2. Low-Level Control: C provides low-level memory manipulation features like pointers, which allow direct manipulation of memory addresses. This feature is essential for tasks like system programming, hardware interfacing, and memory management.

3. Portability: C was designed to be portable across different hardware platforms. It abstracts away many hardware-specific details while still allowing direct memory access when needed.

4. Efficiency: C's syntax and features are designed to be close to the machine level, which results in highly efficient code execution. It allows for direct manipulation of memory and hardware resources, making it suitable for performance-critical applications.

5. Modularity: C supports modular programming through functions, which allows code to be divided into smaller, reusable components. This promotes code organization, reusability, and easier maintenance.

6. Standard Library: C comes with a standard library that provides various functions for common tasks like input/output operations, string manipulation, memory allocation, and mathematical calculations.

Syntax and Basics:
Here's a simple overview of some key concepts in C programming:

- Data Types: C has basic data types like int (integer), float (floating-point), char (character), etc. It also allows the creation of user-defined data types using structures and enums.

- Variables: Variables are used to store data. They must be declared with a data type before being used. For example:

```
int age;
float temperature;
char initial;
```

- Functions: Functions in C allow code to be organized into reusable blocks. Every C program has a "main" function, which is the entry point of the program. Functions are defined with a return type, a name, and a set of parameters. For example:

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

- Control Flow: C supports control flow structures like if-else, switch-case, while loop, for loop, etc., to make decisions and repeat actions.

- Pointers: Pointers are variables that store memory addresses. They allow direct memory access and manipulation. Pointers are used for dynamic memory allocation, data structures, and efficient memory management.

- Arrays: Arrays are collections of elements of the same data type. They provide a way to store multiple values under a single variable name.

- Structures: Structures allow bundling together different data types into a single unit. They are used to create more complex data structures.

- Memory Allocation: C provides functions like "malloc" and "free" for dynamic memory allocation and deallocation, allowing programs to manage memory during runtime.

C is widely used in various domains, including system software (operating systems, compilers), embedded systems, game development, and more. Its simplicity, efficiency, and versatility have contributed to its continued relevance and popularity despite the emergence of newer programming languages.

C CHARACTER SET: -

The C character set, also known as the "source character set," is a collection of characters that can be used in C programming. It encompasses all the characters that are recognized and processed by a C compiler. The character set includes a range of letters, digits, punctuation marks, special symbols, and control characters.

The C character set is further divided into two categories:

1. Basic Source Character Set: This set includes characters that are necessary for the basic functioning of C programs. It consists of the following types of characters:
   - Alphanumeric Characters: The letters 'a' through 'z', 'A' through 'Z', and the digits '0' through '9' are part of this group.
   - Punctuation Marks: Common punctuation marks like period '.', comma ',', semicolon ';', and more.
   - Special Symbols: Special symbols such as parentheses '(', ')', braces '{', '}', square brackets '[', ']', etc.
   - Operators: Operators like plus '+', minus '-', multiplication '*', division '/', etc.
   - Whitespace Characters: Spaces, tabs, and newline characters that help format and structure code.

2. Extended Source Character Set: This set includes additional characters beyond the basic set, such as characters from non-English languages and symbols that may not be as commonly used.

The C character set serves as the foundation for the encoding of source code files. Characters from this set are used to write C programs, and they are interpreted by the C compiler to generate executable code. It's important to note that while the C character set defines the range of characters that can be used in source code, the actual encoding and representation of these characters can vary depending on the character encoding standard being used (e.g., ASCII, UTF-8, UTF-16, etc.).

Modern C compilers often support a wide range of characters from different languages and character sets, allowing developers to write programs that cater to diverse linguistic and cultural needs.

IDENTIFIERS AND KEYWORDS: -

In C programming, identifiers and keywords are fundamental components that play essential roles in defining and structuring your code. Let's explore what identifiers and keywords are:

Identifiers:

Identifiers are names used to identify various program elements, such as variables, functions, arrays, structures, and more. An identifier acts as a label that allows you to reference and work with different parts of your program. Here are some rules for forming valid identifiers in C:

1. An identifier must begin with an alphabetic character (a-z, A-Z) or an underscore ('_').
2. After the initial character, an identifier can consist of letters, digits (0-9), and underscores.
3. Identifiers are case-sensitive, meaning “myVariable” and “myvariable” would be treated as distinct identifiers.
4. Identifiers cannot be C keywords (reserved words) and must not contain spaces or special characters (except underscores).
5. Identifiers should be meaningful and descriptive to improve code readability.

Examples of valid identifiers:
- “counter”
- “_variableName”
- “sumOfValues”
- “MAX_LENGTH”

Keywords:

Keywords are reserved words in C that have predefined meanings and are used to define the structure, flow, and behavior of your program. These words cannot be used as identifiers because they are reserved for specific purposes by the C language itself. Using keywords as identifiers would result in a syntax error. Here are some examples of C keywords:

```
auto     break    case     char     const     continue     default
do       double   else     enum     extern    float        for
goto     if       int      long     register  return       short
signed   sizeof   static   struct   switch    typedef      union
unsigned void     volatile while
```

Since these words have specific meanings and roles within the language, they are off-limits for naming variables, functions, or other program elements.

To summarize, identifiers are names you give to your program's elements, while keywords are predefined words in C with special meanings. It's important to choose meaningful identifiers and avoid using keywords to ensure that your code is readable, maintainable, and free from syntax errors.

DATA TYPES: -

In C programming, data types specify the type of data that a variable can hold. They define the size and format of the stored value, as well as the operations that can be performed on that value. Data types are essential for accurately representing different kinds of information in a program. C provides several built-in data types, which can be broadly categorized into the following groups:

1. Basic Data Types:
   - int: Represents integer values. It typically occupies a certain number of bytes (e.g., 2 or 4) and can hold both positive and negative whole numbers.
   - char: Represents a single character. It is used to store individual characters, such as letters, digits, and symbols.
   - float: Represents floating-point values (real numbers). It is used for decimal or fractional numbers.
   - double: Represents double-precision floating-point values, offering higher precision than “float”.

2. Derived Data Types:
   - Arrays: Collections of elements of the same data type. Arrays allow you to store multiple values under a single variable name.
   - Pointers: Variables that store memory addresses. Pointers are used for memory manipulation, dynamic memory allocation, and more.
   - Structures: User-defined data types that allow you to group different variables of various data types into a single unit.
   - Unions: Similar to structures, but they allow you to store different data types in the same memory location, sharing memory space.
   - Enums: User-defined data types that consist of a set of named integer constants. Enums are used to define a set of related values.

3. Modifier Data Types:
   - signed: Used with integer data types to indicate that the variable can hold both positive and negative values (default for most compilers).

- unsigned: Used with integer data types to indicate that the variable can only hold non-negative values.
- short: Specifies that a variable will occupy less memory than its default data type counterpart.
- long: Specifies that a variable will occupy more memory than its default data type counterpart.

C allows you to create user-defined data types using structures and enums, giving you the flexibility to define data structures that match the requirements of your program.

Using appropriate data types is crucial for memory efficiency, accuracy, and program correctness. Choosing the right data type ensures that your variables can hold the required values and that arithmetic operations are performed correctly without unexpected truncation or overflow.

CONSTANTS: -

In C programming, constants are fixed values that cannot be altered during the execution of a program. They are used to represent specific values like numbers, characters, strings, and more, and they provide a way to make code more readable and maintainable by giving meaningful names to these fixed values.

Constants in C can be categorized into two main types:

1. Literal Constants: These are direct, explicit values that are written directly into the source code. They are categorized based on the data types they represent:

- Integer Constants: These are whole numbers without decimal points. For example: “123”, “-456”, “0”.
- Floating-Point Constants: These are numbers with decimal points. For example: “3.14”, “-0.005”.
- Character Constants: These are single characters enclosed in single quotes. For example: 'A', '5', '!'.
- String Constants: These are sequences of characters enclosed in double quotes. For example: "Hello, World!".

2. Symbolic Constants (Named Constants): These are constants defined using symbolic names to represent specific values. They are typically created using the “#define” preprocessor directive or the “const” keyword. Symbolic constants are often preferred because they make the code more readable, allow for easy changes in a single place, and can provide some level of type safety.

- Using #define:

```
#define PI 3.14159265
#define MAX_VALUE 100
```

- Using const Keyword:

```
const double PI = 3.14159265;
const int MAX_VALUE = 100;
```

Symbolic constants are especially useful when you want to define values that might be reused multiple times in your code. By assigning a meaningful name to a constant, you make the code more self-explanatory and reduce the likelihood of errors caused by manually changing the same value in multiple places.

For example, using a symbolic constant for the value of π (pi) would make the code clearer and allow you to change the value in just one place if needed:

```
#define PI 3.14159265

double area = PI * radius * radius;
```

Constants are used in various parts of a program, such as assignments, calculations, comparisons, and function calls. Here are a few examples of using constants:

```
int radius = 5;             // Integer constant
float pi = 3.14159;         // Floating-point constant
char grade = 'A';           // Character constant
char message[] = "Hello";   // String constant
```

In summary, constants in C provide a way to represent fixed values in your program. They can be either literal constants with explicit values or symbolic constants defined with meaningful names. Symbolic constants are preferred for enhancing code readability, maintainability, and reducing the likelihood of errors.

VARIABLE DECLARATIONS: -

Variable declarations in C involve specifying the characteristics of a variable before it is used in a program. Declaring a variable informs the compiler about the variable's data type and its intended use. This allows the compiler to allocate memory and perform necessary setup to accommodate the variable's storage and manipulation. Variable declarations typically include the variable's name and its data type.

Here's the general syntax for declaring a variable in C:

```
data_type variable_name;
```

- data_type: This specifies the type of data the variable will store. It can be one of the basic data types (int, float, char, etc.) or a user-defined data type like structures or enums.

- variable_name: This is the name you give to the variable. It must follow the rules for naming identifiers: starting with a letter or underscore, followed by letters, digits, or underscores.

For example, to declare an integer variable named "age," you would write:

```
int age;
```

Variables can also be initialized at the time of declaration by assigning an initial value:

```
int count = 0; // Declare and initialize an integer variable "count" with the value 0
float pi = 3.14; // Declare and initialize a floating-point variable "pi" with the value 3.14
char initial = 'A'; // Declare and initialize a character variable "initial" with the value 'A'
```

Multiple variables of the same data type can be declared in a single line by separating their names with commas:

```
int x, y, z;   // Declare three integer variables: x, y, and z
```

Variable declarations can also include other modifiers and qualifiers such as "const," "static," "extern," and more, to specify additional properties of the variable.

It's important to note that variable declarations should appear before the variables are used in the code. The scope of a variable, which determines where it can be accessed, is determined by its declaration location. Declaring variables before they are used helps in avoiding issues related to undeclared variables and ensures proper memory allocation and initialization.

STRUCTURE OF BASIC C PROGRAM: -

A basic C program consists of several parts that work together to define the structure and functionality of the program. Here's an outline of the typical structure of a basic C program:

```c
// Preprocessor Directives
#include <stdio.h> // Include necessary header files

// Function Declarations (optional)
// Declare functions used in the program

// Main Function
int main() {
    // Variable Declarations

    // Code Statements
    // Write the main logic of the program here

    // Return Statement
    return 0; // Indicates successful execution
}
```

Let's break down each part of the structure:

1. Preprocessor Directives: These lines begin with "#" and are processed by the preprocessor before the actual compilation. They typically include header files that provide essential functionality to the program, such as input/output functions ("printf" and "scanf") from the "<stdio.h>" header.

2. Function Declarations: While not always required, you can declare functions that you plan to use later in your program. Function declarations typically appear before the "main" function and help the compiler understand that these functions will be defined elsewhere.

3. Main Function: The "main" function is the entry point of the program. It's where the execution begins. The "int" before "main" indicates that the function returns an integer value (usually 0 to indicate successful execution). Inside the "main" function, you'll write the main logic of your program.

4. Variable Declarations: This is where you declare the variables you will use in your program. Variables must be declared before they are used in C. You specify the data type and name of each variable.

5. Code Statements: This is where you write the actual code of your program. This code contains the steps your program will take to achieve its desired functionality. This can include assignments, calculations, loops, conditionals, and function calls.

6. Return Statement: The “return” statement marks the end of the “main” function and indicates the program's exit status. In most cases, you return 0 to signify successful execution. A non-zero value may indicate an error condition.

Here's a very simple example of a basic C program that prints "Hello, World!" to the console:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

This structure provides a foundation for building more complex C programs by incorporating various control structures, user-defined functions, and data structures.

#INCLUDE PREPROCESSOR DIRECTIVE: -

In C programming, the “#include” preprocessor directive is used to include external files, typically header files, into your source code before compilation. Header files contain declarations for functions, macros, constants, and other elements that are needed by your program. The “#include” directive is processed by the preprocessor, a part of the compilation process that handles various text manipulations before the actual compilation of the source code begins.

The syntax of the “#include” directive is:

```c
#include <header_filename>
```

There are two main forms of including header files:

1. Including Standard Library Headers:
Standard library headers provide essential functions and macros that are part of the C standard library. These headers are enclosed in angle brackets “< > “.

For example:

```c
#include <stdio.h> // Includes the standard I/O header file
#include <stdlib.h> // Includes the standard library header file
```

2. Including User-Defined Headers:
You can create your own header files to organize your code and declarations. These headers are enclosed in double quotes " ". You typically include user-defined headers by providing the relative or absolute path to the header file.

For example:

```
#include "myheader.h" // Includes a user-defined header file named myheader.h
```

The "#include" directive allows you to access functions, macros, and other declarations defined in the included header files. This practice promotes code reusability and modular programming. When you use functions or macros from an included header, the compiler knows their definitions without needing to copy and paste the entire content of the header into your source code.

Here's a simple example demonstrating the use of the "#include" directive with the standard I/O header:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

In this example, "#include <stdio.h>" allows you to use the "printf" function from the standard I/O library without needing to define its details in your source code.

EXPRESSION STATEMENTS: -

In C programming, an expression statement is a line of code that consists of an expression followed by a semicolon (";"). An expression is a combination of values, variables, operators, and function calls that evaluates to a single value. Expression statements are a fundamental building block of C programs and play a crucial role in performing calculations, assignments, function calls, and more.

Here are a few common types of expression statements:

1. Assignment Expression:
An assignment expression assigns a value to a variable. It typically takes the form of "variable = value;", where "variable" is a variable identifier and "value" is the value being assigned to the variable.

```
int x, y;
x = 10;      // Assign 10 to variable x
y = x + 5;  // Assign the result of x + 5 to variable y
```

2. Function Call Expression:

A function call expression invokes a function and can include arguments that are passed to the function. The result of the function call can be used in an assignment or other expressions.

```
int result;
result = add(3, 5);  // Call the add function with arguments 3 and 5, and a
```

```
and assign the result to result variable
```

3. Arithmetic Expression:

An arithmetic expression involves mathematical operations such as addition, subtraction, multiplication, and division. The result of the expression is a single value.

```
int total;
total = x + y;    // Calculate the sum of x and y and assign it to total
```

4. Conditional Expression:

Conditional expressions (ternary operators) allow you to create concise if-else statements in a single line. They have the form: "condition ? value_if_true : value_if_false".

```
int max = (x > y) ? x : y;  // Assign the larger of x and y to max
```

5. Increment and Decrement Expressions:

Increment ("++") and decrement ("--") expressions increase or decrease the value of a variable by 1, respectively.

```
int count = 0;
count++;  // Increment count by 1
```

Expression statements are a way to perform operations, make decisions, and modify data within your program. These statements are executed sequentially, one after the other, following the order in which they are written in the program.

COMPOUND STATEMENTS: -

In C programming, a compound statement (also known as a block) is a group of multiple statements enclosed within curly braces "{}". A compound statement allows you to group statements together, and it is treated as a single statement by the compiler. Compound statements are often used in situations where multiple statements need to be executed together, such as in loops, if-else statements, and function definitions.

The syntax of a compound statement is as follows:

```
{
    statement1;
    statement2;
    // ...
    statementN;
}
```

In a compound statement, the statements inside the curly braces are executed sequentially, one after the other, from top to bottom. The compound statement itself can be treated as a single statement in the context of program flow.

Here are some common situations where compound statements are used:

1. Loops:
In loops, you often need to execute multiple statements repeatedly. A compound statement allows you to group these statements together within the loop's body.

```
for (int i = 0; i < 5; i++) {
    printf("Iteration %d\n", i);
    printf("Value: %d\n", i * 2);
}
```

2. Conditional Statements (if-else):
When using if-else statements, you can enclose multiple statements within a compound statement to define what should be executed in each case.

```
if (x > y) {
    printf("x is greater than y\n");
    y = x;
} else {
    printf("y is greater than or equal to x\n");
    x = y;
}
```

3. Function Definitions:

Functions are defined using compound statements. The body of the function contains multiple statements enclosed within curly braces.

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

4. Switch Statements:

Switch statements can contain multiple cases, each with its own set of statements. Compound statements are used to group the statements within each case.

```c
switch (choice) {
    case 1:
        printf("You selected option 1\n");
        break;
    case 2:
        printf("You selected option 2\n");
        break;
    // ...
    default:
        printf("Invalid choice\n");
}
```

Compound statements provide a way to logically group statements together, improve code organization, and control the scope of variables. They are essential for creating structured and readable code in C programs.

OPERATORS: -

1. Arithmetic Operators:
Arithmetic operators are used to perform mathematical operations on operands. They include:
- "+" (Addition): Adds two operands.
- "-" (Subtraction): Subtracts the right operand from the left operand.
- "*" (Multiplication): Multiplies two operands.
- "/" (Division): Divides the left operand by the right operand.
- "%" (Modulus): Returns the remainder of the division of the left operand by the right operand.

```c
#include <stdio.h>

int main() {
    int a = 10, b = 5;

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

    return 0;
}
```

2. Unary Operators:
Unary operators work with a single operand:
- "+" (Positive): Represents the value of the operand.
- "-" (Negative): Represents the negation of the operand.
- "++" (Increment): Increases the operand by 1.
- "--" (Decrement): Decreases the operand by 1.
- "!" (Logical NOT): Returns the logical NOT of the operand.

```c
#include <stdio.h>

int main() {
    int a = 5;

    printf("+a = %d\n", +a); // Positive
    printf("-a = %d\n", -a); // Negative

    int b = 10;
    printf("++b = %d\n", ++b); // Pre-increment
    printf("--b = %d\n", --b); // Pre-decrement

    int c = 7;
    printf("c++ = %d\n", c++); // Post-increment
    printf("c = %d\n", c);

    return 0;
}
```

3. Relational Operators:
Relational operators are used to compare two operands:
- “==“ (Equal to): Checks if two operands are equal.
- “!=“ (Not equal to): Checks if two operands are not equal.
- “<“ (Less than): Checks if the left operand is less than the right operand.
- “>“ (Greater than): Checks if the left operand is greater than the right operand.
- “<=“ (Less than or equal to): Checks if the left operand is less than or equal to the right operand.
- “>=“ (Greater than or equal to): Checks if the left operand is greater than or equal to the right operand.

```c
#include <stdio.h>

int main() {
    int x = 10, y = 20;

    printf("x == y: %d\n", x == y); // Equal to
    printf("x != y: %d\n", x != y); // Not equal to
    printf("x < y: %d\n", x < y);   // Less than
    printf("x > y: %d\n", x > y);   // Greater than
    printf("x <= y: %d\n", x <= y); // Less than or equal to
    printf("x >= y: %d\n", x >= y); // Greater than or equal to

    return 0;
}
```

4. Logical Operators:
Logical operators are used to combine or modify logical values:
- “&&” (Logical AND): Returns true if both operands are true.
- “||” (Logical OR): Returns true if at least one operand is true.
- “!” (Logical NOT): Returns true if the operand is false and vice versa.

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10;

    printf("a && b: %d\n", a && b); // Logical AND
    printf("a || b: %d\n", a || b); // Logical OR
    printf("!a: %d\n", !a);         // Logical NOT

    return 0;
}
```

5. Assignment Operators:
Assignment operators are used to assign values to variables:
- “=“ (Assignment): Assigns the value on the right to the variable on the left.
- “+=“, “-=“, “*=“, “/=“, “%=“ (Shorthand Assignment): Performs the operation and assigns the result to the left operand.

```c
#include <stdio.h>

int main() {
    int x = 5;
    x += 3; // Shorthand for x = x + 3

    printf("x = %d\n", x);

    return 0;
}
```

6. Conditional (Ternary) Operator:
The conditional operator is a shorthand way to write if-else statements:
- “condition ? expr1 : expr2”: If the condition is true, expr1 is evaluated; otherwise, expr2 is evaluated.

```c
#include <stdio.h>

int main() {
    int a = 10, b = 5;
    int max = (a > b) ? a : b;

    printf("Max value: %d\n", max);

    return 0;
}
```

7. Bitwise Operators:
Bitwise operators work at the bit level:
- “&” (Bitwise AND): Performs a bitwise AND operation.
- “|” (Bitwise OR): Performs a bitwise OR operation.
- “^” (Bitwise XOR): Performs a bitwise exclusive OR operation.
- “~” (Bitwise NOT): Flips the bits of the operand.
- “<<“ (Left Shift): Shifts the bits of the left operand left by the number of positions specified by the right operand.
- “>>“ (Right Shift): Shifts the bits of the left operand right by the number of positions specified by the right operand.

```c
#include <stdio.h>

int main() {
    int x = 5, y = 3;

    printf("x & y: %d\n", x & y); // Bitwise AND
    printf("x | y: %d\n", x | y); // Bitwise OR
    printf("x ^ y: %d\n", x ^ y); // Bitwise XOR
    printf("~x: %d\n", ~x);       // Bitwise NOT
    printf("x << 1: %d\n", x << 1); // Left shift
    printf("y >> 1: %d\n", y >> 1); // Right shift

    return 0;
}
```

8. Comma Operator:
The comma operator is used to separate expressions in a statement. It evaluates each expression from left to right and returns the value of the rightmost expression.
- "," (Comma): Separates expressions, such as in function calls or variable declarations.

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10, c = 15;

    int sum = (a + b, b + c, c + a); // The value of 'sum' is assigned the v
    printf("Sum: %d\n", sum); // Output: 20

    return 0;
}
```

```
// The value of 'sum' is assigned the value of the last expression (c + a)
```

These operators are fundamental tools for performing calculations, making decisions, and manipulating data in C programming. Understanding their behavior and usage is crucial for writing effective and efficient code.

C CONTROL STRUCTURES: -

Control structures in C are constructs that allow you to control the flow of your program's execution based on conditions or loops. Let's explore each of the control structures you mentioned:

1. if Statement:
The “if” statement is used to execute a block of code only if a certain condition is true.

```
if (condition) {
    // Code to execute if the condition is true
}
```

2. if...else Statement:
The “if...else” statement allows you to execute one block of code if a condition is true and another block if the condition is false.

```
if (condition) {
    // Code to execute if the condition is true
} else {
    // Code to execute if the condition is false
}
```

3. else if Ladder:
The “else if” ladder is an extension of the “if...else” statement, allowing you to test multiple conditions in a sequence.

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else if (condition3) {
    // Code to execute if condition3 is true
} else {
    // Code to execute if none of the conditions are true
}
```

4. while Loop:
The “while” loop repeatedly executes a block of code as long as a condition is true.

```
while (condition) {
    // Code to repeat while the condition is true
}
```

5. do...while Loop:
The “do...while” loop is similar to the “while” loop, but it ensures that the loop body is executed at least once before checking the condition.

```
do {
    // Code to execute
} while (condition);
```

6. for Loop:
The “for” loop is used to execute a block of code repeatedly with a defined initialization, condition, and iteration.

```
for (initialization; condition; iteration) {
    // Code to execute
}
```

7. switch Statement:
The “switch” statement allows you to execute different code blocks based on the value of a variable.

```
switch (variable) {
    case value1:
        // Code to execute for value1
        break;
    case value2:
        // Code to execute for value2
        break;
    // ...
    default:
        // Code to execute if none of the cases match
}
```

8. Nested Control Structures:
Control structures can be nested within each other to create more complex behavior. For example, you can place an “if” statement inside a “while” loop, or a “for” loop inside an “if” statement.

```
if (condition1) {
    while (condition2) {
        // Code to execute
    }
}
```

These control structures allow you to create structured and organized programs that perform various tasks based on conditions and loops. By combining these structures, you can create sophisticated algorithms and applications.

CONTROL STATEMENTS IN C: -

1. Break Statement:
The “break” statement is used to exit the nearest enclosing loop (such as “for”, “while”, or “do...while”) prematurely, regardless of whether the loop's condition is still true.

```c
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i equals 5
    }
    printf("%d\n", i);
}
```

2. Labelled break Statement:
A labelled “break” statement is used to exit a specific loop when there are nested loops. The label identifies the loop to exit.

```c
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == 1 && j == 1) {
            break; // Exits the inner loop only
        }
        printf("%d %d\n", i, j);
    }
}
```

3. Continue Statement:
The “continue” statement is used to skip the current iteration of a loop and move to the next iteration.

```c
for (int i = 0; i < 5; i++) {
    if (i == 2) {
        continue; // Skips iteration when i equals 2
    }
    printf("%d\n", i);
}
```

4. Labelled continue Statement:
Similar to labelled “break”, the labelled “continue” statement allows you to skip the current iteration of a specific loop when there are nested loops.

```c
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (i == 1 && j == 1) {
            continue; // Skips the current iteration of the inner loop only
        }
        printf("%d %d\n", i, j);
    }
}
```

5. Exit Statement:

The “exit” function from the “stdlib.h” library is used to terminate the entire program. It can be used to exit the program with a specified exit status.

```c
#include <stdlib.h>

int main() {
    printf("Exiting program\n");
    exit(0); // Exit the program with status 0
}
```

6. Goto Statement:

The “goto” statement allows you to transfer control to a labeled statement within the same function. However, its usage is discouraged in modern programming practices due to potential code complexity and difficulty in understanding.

```c
int main() {
    int count = 0;

    start:
    if (count < 5) {
        printf("%d\n", count);
        count++;
        goto start; // Jump back to the 'start' label
    }

    return 0;
}
```

While these control statements provide flexibility in programming, they can also lead to less readable and harder-to-maintain code. It's important to use them judiciously and consider alternative ways to achieve the desired behavior whenever possible.

UNIT-II

C FUNCTIONS: -

1. Function Declaration:
A function declaration informs the compiler about the function's name, return type, and the types of its parameters. It enables the compiler to perform type checking and generate correct code when the function is called.

```
return_type function_name(parameter_list);
```

Example:

```
int add(int a, int b); // Function declaration
```

2. Function Definition:
A function definition provides the actual implementation of the function. It includes the function body, which contains the statements that define what the function does.

```
return_type function_name(parameter_list) {
    // Function body
    // Statements
}
```

Example:

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

3. Function Scope:
The scope of a function refers to the area of the program where the function is accessible. Functions have local scope, meaning that variables declared within the function are only visible and accessible within that function.

```
void exampleFunction() {
    int localVar = 10; // Local variable
}

int main() {
    // localVar is not accessible here
    return 0;
}
```

4. Recursion:
Recursion is a technique in which a function calls itself to solve a problem. Recursive functions have a base case (a condition that stops the recursion) and a recursive case (where the function calls itself with modified arguments).

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

5. Call by Value:
In call by value, the function receives a copy of the actual arguments passed to it. Changes made to the parameters within the function do not affect the original arguments.

```c
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5, b = 10;
    swap(a, b); // The values of a and b remain unchanged
    return 0;
}
```

6. Call by Reference:
In call by reference, the function receives the memory addresses (pointers) of the actual arguments. Changes made to the parameters within the function affect the original arguments.

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b); // The values of a and b are swapped
    return 0;
}
```

C functions play a critical role in structuring and organizing code, allowing for code reuse, modularity, and improved readability. Understanding how to declare, define, and use functions is fundamental for creating efficient and maintainable programs.

PREPROCESSOR DIRECTIVE: -

Preprocessor directives in C are commands that are processed by the preprocessor before the actual compilation of the source code. They allow you to perform various text manipulations and conditional compilation. Let's define and explain each of the mentioned preprocessor directives:

1. #define Directive:
The "#define" directive is used to create symbolic constants and simple macros. It associates a name with a value or an expression.

```
#define PI 3.14159
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

2. Macros with Arguments:
Macros with arguments allow you to define functions-like constructs using preprocessor directives.

```
#define SQUARE(x) ((x) * (x))
```

3. Nested Macros:
Macros can be nested within each other, allowing for more complex expressions.

```
#define MAX_SQUARE(a, b) SQUARE(MAX(a, b))
```

4. # and ## Operators:
The "#" operator, called the stringizing operator, converts a macro parameter into a string literal.
The "##" operator, called the token-pasting operator, concatenates two tokens.

```
#define TO_STRING(x) #x
#define CONCAT(a, b) a ## b
```

5. Conditional Compilation:
Conditional compilation allows you to include or exclude parts of the code during compilation based on certain conditions.

```
#if defined(DEBUG)
    printf("Debug mode\n");
#else
    printf("Release mode\n");
#endif
```

6. #ifdef and #ifndef Directives:
The “#ifdef” directive checks if a macro is defined. The “#ifndef” directive checks if a macro is not defined.

```
#ifdef DEBUG
    printf("Debug mode\n");
#else
    printf("Release mode\n");
#endif
```

7. #elif Directive:
The “#elif” directive is used for alternative conditional checks within a block of conditional compilation.

```
#if defined(DEBUG)
    printf("Debug mode\n");
#elif defined(TEST)
    printf("Test mode\n");
#else
    printf("Release mode\n");
#endif
```

These preprocessor directives offer powerful tools for code organization, modularity, and conditional compilation. They allow you to create flexible and efficient code tailored to different scenarios and environments.

STORAGE CLASSES IN C: -

Storage classes in C determine the scope, lifetime, and initial value of variables. There are four main storage classes in C:

1. Automatic Storage Class:

The “auto” storage class is the default for all local variables declared within a function. Variables with the “auto” storage class are automatically allocated memory when the function is entered and released when the function exits.

```
void someFunction() {
    auto int x = 5; // Equivalent to: int x = 5;
}
```

2. External (Global) Storage Class:

The “extern” storage class is used to declare a global variable that can be accessed by multiple source files. The variable is defined elsewhere in the program, typically in another source file.

File: file1.c

```c
extern int globalVar; // Declare globalVar from another source file

void functionInFile1() {
    globalVar = 10; // Access and modify globalVar
}
```

File: file2.c

```c
int globalVar; // Define globalVar here

int main() {
    functionInFile1();
    return 0;
}
```

3. Static Storage Class:

The “static” storage class has different meanings depending on where it is used:

- When used with a local variable within a function, it preserves the value of the variable across function calls and retains memory across function calls.

- When used with a global variable, it restricts the scope of the variable to the current source file, preventing it from being accessed in other files.

```c
void someFunction() {
    static int count = 0; // The value of count persists across function calls
}
```

4. Register Storage Class:

The “register” storage class suggests that a variable should be stored in a CPU register for faster access. However, the compiler may or may not actually store the variable in a register.

```c
register int x; // Suggests that 'x' be stored in a register
```

These storage classes provide control over variable properties such as scope, lifetime, and memory allocation. Understanding them helps you manage memory efficiently and control the behavior of your program's variables.

ARRAYS IN C: -

Let's dive into each of these concepts related to arrays, strings, pointers, and dynamic memory allocation in C:

1. Arrays:
An array is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values under a single variable name.

- 1D Array:

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

- 2D Array:

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

2. Strings:
Strings are arrays of characters terminated by a null character ('\0'). They are used to represent text.

```
char greeting[] = "Hello, world!";
```

3. Pointers:
A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access and manipulate memory.

```
int x = 10;
int *ptr = &x; // Pointer to integer
```

4. Array-Pointer Relationship:
Arrays and pointers are closely related. An array name often acts as a pointer to the first element of the array.

```
int numbers[5] = {1, 2, 3, 4, 5};
int *ptr = numbers; // 'ptr' points to the first element of 'numbers'
```

5. Pointer Arithmetic:
Pointer arithmetic allows you to perform arithmetic operations on pointers to navigate through memory.

```
int numbers[5] = {1, 2, 3, 4, 5};
int *ptr = numbers;
ptr++; // Moves 'ptr' to the next element of the array
```

6. Dynamic Memory Allocation:
C allows you to allocate memory dynamically using functions like “malloc”, “calloc”, and “realloc”.

```
int *dynamicArray = (int *)malloc(5 * sizeof(int));
```

7. Pointer to Arrays:
A pointer to an array points to the first element of the array.

```
int numbers[5] = {1, 2, 3, 4, 5};
int (*ptr)[5] = &numbers; // Pointer to an array of 5 integers
```

8. Array of Pointers:
An array of pointers is an array where each element is a pointer.

```
int x = 10, y = 20, z = 30;
int *ptrArray[] = {&x, &y, &z}; // Array of pointers to integers
```

9. Pointers to Functions:
Pointers to functions allow you to store and call functions through pointers.

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

int (*funcPtr)(int, int) = add; // Pointer to a function
int result = funcPtr(5, 3); // Call function through pointer
```

10. Array of Pointers to Functions:
An array of pointers to functions allows you to store and call multiple functions through pointers.

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

int subtract(int a, int b) {
    return a - b;
}

int (*funcPtrArray[2])(int, int) = {add, subtract}; // Array of pointers to
int result = funcPtrArray[0](5, 3); // Call function through pointer
```

```
to functions
```

These concepts are fundamental in C programming, enabling you to work with data, memory, and functions efficiently and effectively.

UNIT-III

STRUCTURES IN C: -

1. Structures:
A structure is a user-defined composite data type that groups together variables of different data types under a single name. It allows you to create more complex data structures.

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

2. Unions:
A union is a data type that allows you to store different types of data in the same memory location. Unlike structures, unions share the same memory space for all their members.

```c
union Data {
    int num;
    char letter;
};
```

3. Enumeration:
Enumeration is a user-defined data type that consists of a set of named values. It provides a way to create symbolic names for integral constants.

```c
enum Days {
    Sunday,
    Monday,
    Tuesday,
    // ...
};
```

4. Passing Structure to Functions:
You can pass structures to functions by value or by pointer, allowing functions to operate on complex data.

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

5. Arrays and Structures:
Arrays of structures allow you to create collections of related data.

```c
struct Student students[3];
```

6. Types of Structures:
- Structure: Contains members of different data types.
- Nested Structure: Structure containing another structure as a member.
- Array of Structures: Array where each element is a structure.
- Pointer to Structure: Pointer pointing to a structure.

7. Difference Between Structure and Union:

Structures:
Memory Usage: Each member of a structure occupies its own memory space, and the total memory required for a structure is the sum of memory required for each member.
Member Access: You can access all members of a structure simultaneously.
Size: The size of a structure is the sum of the sizes of all its members.
Initialization: You can initialize all members of a structure individually.
Common Use: Used when you want to store related data with different data types.

Unions:
Memory Usage: All members of a union share the same memory location, and the memory required for a union is the size of its largest member.
Member Access: Only one member of a union can be accessed at a time. Reading one member might overwrite the value of another.
Size: The size of a union is the size of its largest member.
Initialization: You can initialize only the first member of a union explicitly.
Common Use: Used when you want to store different data types in the same memory location to save space.

8. Self-Referential Structure:
A structure that has a member of the same structure type as itself.

```c
struct Node {
    int data;
    struct Node *next;
};
```

9. Bit Fields:
Bit fields allow you to define the number of bits each member of a structure or union should occupy in memory.

```c
struct Flags {
    unsigned int isSet: 1;
    unsigned int isAdmin: 1;
};
```

These concepts enhance your ability to create and manage complex data structures, making your programs more organized, efficient, and readable.

FILE HANDLING IN C: -

File handling in C allows you to perform operations on files, including reading and writing data. There are two main types of file handling: text (ASCII) and binary.

File Input/Output Operations:

1. Opening a File: To open a file, you use the "fopen" function with a specified filename and access mode. The function returns a file pointer that represents the opened file.

```
FILE *filePtr = fopen("example.txt", "r"); // Open for reading
```

2. Reading from a File: You can use functions like "fgetc", "fgets", and "fscanf" to read data from a file.

```
char ch = fgetc(filePtr); // Read a character
char buffer[100];
fgets(buffer, sizeof(buffer), filePtr); // Read a line
fscanf(filePtr, "%d", &integerValue); // Read formatted data
```

3. Writing to a File: Functions like "fputc", "fputs", and "fprintf" allow you to write data to a file.

```
fputc('A', filePtr); // Write a character
fputs("Hello, World!", filePtr); // Write a string
fprintf(filePtr, "Value: %d", value); // Write formatted data
```

4. Closing a File: It's important to close the file using "fclose" when you're done with it.

```
fclose(filePtr);
```

File Access Modes:

- "r": Read mode
- "w": Write mode (creates a new file or overwrites an existing file)
- "a": Append mode (creates a new file or appends to an existing file)
- "rb": Read mode in binary
- "wb": Write mode in binary
- "ab": Append mode in binary

File Pointers:

A file pointer is used to keep track of the current position in a file. It's automatically updated when you read or write data.

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

File Positioning Functions:
- "fseek(filePtr, offset, origin)": Moves the file pointer to a specified position.
- "ftell(filePtr)": Returns the current position of the file pointer.
- "rewind(filePtr)": Moves the file pointer to the beginning of the file.

Example usage:

```
fseek(filePtr, 10, SEEK_SET); // Move 10 positions from the beginning
long position = ftell(filePtr); // Get the current position
rewind(filePtr); // Move to the beginning of the file
```

Binary File Handling:
Binary file handling involves reading and writing raw binary data. You use the same functions ("fread" and "fwrite") as text files, but with different data types.

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

struct Person person;
FILE *binaryFile = fopen("data.dat", "rb");
fread(&person, sizeof(struct Person), 1, binaryFile);
fclose(binaryFile);
```

File handling in C allows you to interact with files in various ways, facilitating data storage, retrieval, and manipulation. Understanding these operations is crucial for working with files effectively in your programs.

UNIT-II

C STANDARD LIBRARY: -

The C Standard Library provides a rich set of functions that cover a wide range of operations, from input/output to string manipulation, mathematical calculations, memory management, and more. Here's an overview of some commonly used functions from the standard library headers you mentioned:

1. stdio.h (Standard Input/Output functions):

- "printf", "fprintf", "sprintf": Formatted output functions.
- "scanf", "fscanf", "sscanf": Formatted input functions.
- "fgets", "gets": Read strings from input.
- "fputs", "puts": Write strings to output.
- "getc", "fgetc", "getchar": Read characters from input.
- "putc", "fputc", "putchar": Write characters to output.
- "fclose", "fflush": Close files or flush the output buffer.
- "fseek", "ftell", "rewind": File positioning functions.
- "remove", "rename": File manipulation functions.

2. stdlib.h (Standard Library functions):

- "malloc", "calloc", "realloc": Memory allocation functions.
- "free": Deallocate memory.
- "rand", "srand": Generate random numbers.
- "exit", "abort": Terminate program execution.
- "atoi", "atof", "strtol", "strtod": String to numeric conversion functions.
- "system": Execute a command in the system shell.

3. conio.h (Console Input/Output functions, platform-dependent - not part of the C Standard):

- "clrscr", "gotoxy": Screen manipulation functions.
- "getch", "getche": Get characters from the console without echoing.
- "cprintf", "cscanf": Formatted input/output functions.

4. ctype.h (Character functions):

- "isalpha", "isdigit", "isalnum": Check if character is a letter, digit, or alphanumeric.
- "islower", "isupper": Check if character is lowercase or uppercase.
- "tolower", "toupper": Convert character to lowercase or uppercase.

5. math.h (Mathematical functions):

- "sqrt", "pow", "exp", "log": Square root, power, exponential, and logarithmic functions.
- "sin", "cos", "tan": Trigonometric functions.
- "ceil", "floor", "round": Ceiling, floor, and round functions.
- "abs", "labs", "llabs": Absolute value functions.
- "rand", "srand": Random number generation functions.

6. string.h (String functions):

- “strlen”, “strcpy”, “strcat”, “strcmp”: String manipulation functions.
- “strchr”, “strstr”, “strtok”: Search and tokenization functions.
- “memset”, “memcpy”, “memmove”: Memory manipulation functions.

7. process.h (Platform-dependent - not part of the C Standard, mostly for DOS and Windows):

- “spawnl”, “spawnlp”: Launch a new process.

Command Line Arguments:

When you run a C program from the command line, you can pass arguments to it. The arguments are stored in the “main” function's parameters.

```c
int main(int argc, char *argv[]) {
    // argc: Number of arguments
    // argv: Array of strings representing the arguments

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

These standard library functions are crucial tools for performing various tasks in C programming. They provide ready-to-use solutions for common programming challenges and enable you to build efficient and feature-rich applications.