PROGRAMMING IN C

UNIT 1

1.Define an algorithm and a flowchart.

Algorithm:

An **algorithm** is a step-by-step set of instructions designed to perform a specific task or solve a problem. It is a logical sequence of actions that leads to a desired result. Algorithms can be written in plain language, pseudocode, or programming languages and should be finite, unambiguous, and effective.

Flowchart:

A **flowchart** is a graphical representation of an algorithm or process. It uses various shapes such as rectangles, diamonds, ovals, and arrows to visually describe the flow of steps in a process. Flowcharts are useful for understanding the sequence of operations and identifying decision points.

2. What is meant by a data types? Give its classifications.

A **data type** in programming defines the type of data that a variable can store. It specifies the type of value (like a number, a character, etc.) and the operations that can be performed on that value. Data types help in organizing and managing data efficiently within a program.

Classifications of Data Types:

Data Type	**Example of Data Type**
Primary Data Type	Integer, Floating-point, double, string.
Derived Data Type	**Union**, structure, array, etc.
Enumerated Data Type	**Enums**
Void Data Type	Empty Value
Bool Type	True or False

3. What are keywords?

Keywords in C are reserved words that have specific meanings within the language. They cannot be used as variable names, function names, or any other identifiers. Keywords are essential for defining the structure, control flow, and operations of a C program.

Data Types:

1.**int:** Integer 2.**float:** Single-precision floating-point number,3.**double**: Double-precision floating-point number,4.**char**: Character,5.**void**: Indicates no type or empty type

Control Flow:

1.**if, else, else if**: Conditional statements ,2.**for, while, do-while**: Loops,3.**switch, case, default**: Multiple-choice decisions,4.**break**: Exits a loop or switch statement,5.**continue**: Skips the current iteration of a loop

Storage Class Specifiers:

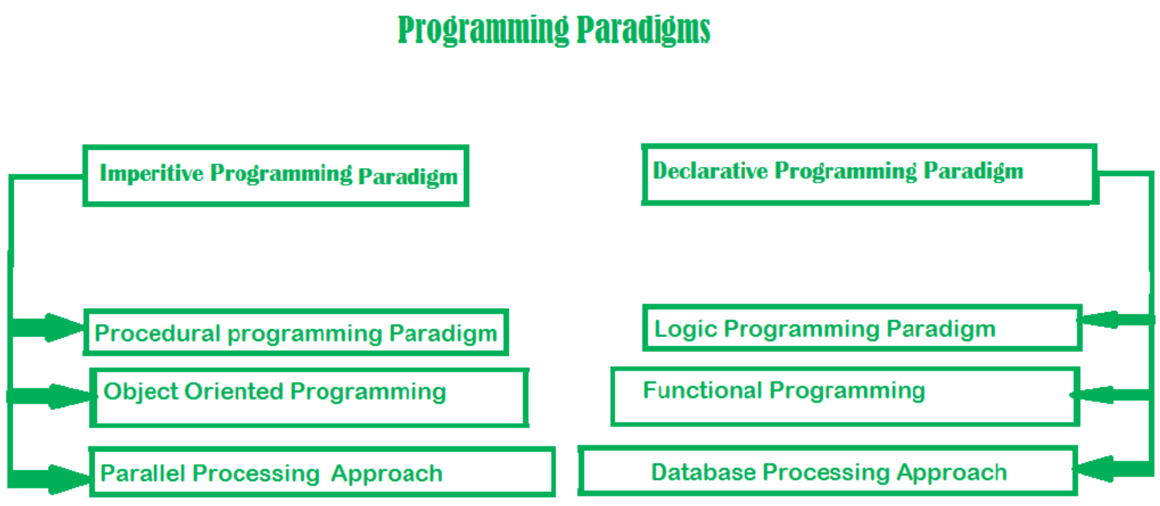
1.**auto**: Automatic storage duration,2.**register**: Suggests to the compiler to store a variable in a register for faster access,3.**static**: Static storage duration,5.**extern**: Declares a variable or function defined elsewhere

Other Keywords:

1.**return**: Returns a value from a function,2.**sizeof**: Returns the size of a data type or variable,3.**typedef**: Creates a new type name,4.**struct**: Defines a structure,5.**union**: Defines a union6.**enum**: Defines an enumeration,7.**const**: Declares a constant

4. Illustrate the concept of the programming paradigm.

Programming paradigms are fundamental approaches or styles of programming that dictate how a problem is solved and how code is organized. They provide a framework for thinking about and structuring code.


Programming Paradigms
Imperitive Programming Paradigm
Declarative Programming Paradigm
Procedural programming Paradigm
Object Oriented Programming
Parallel Processing Approach
Logic Programming Paradigm
Functional Programming
Database Processing Approach

5. What are the input and output statements in C?

In **C programming**, input and output (I/O) operations allow interaction between the user and the program. These operations can be performed using standard input/output functions, primarily found in the stdio.h header file.

Input Statements in C:

Input statements allow the user to provide data to the program during its execution.

Formatted input: scanf():

Unformatted input: getchar(), getche(), getch(),gets(),fgets()

Output Statements in C:

Output statements allow the program to display data or results to the user.

Formatted Output: printf()

Unformatted Output : putchar() , putch(),puts(),fputs()

6. Analyze Increment and Decrement Operators with an example.

Increment and decrement operators are unary operators used to modify the value of a variable by 1. They are often used to iterate through arrays or perform simple arithmetic operations.

Increment Operator (++)

- Pre-increment: When the operator is placed before the variable (e.g., ++x), the value of the variable is incremented by 1 before it is used in the expression.
- Post-increment: When the operator is placed after the variable (e.g., x++), the value of the variable is used in the expression first, and then it is incremented by 1.

Decrement Operator (--)

- Pre-decrement: When the operator is placed before the variable (e.g., --x), the value of the variable is decremented by 1 before it is used in the expression.
- Post-decrement: When the operator is placed after the variable (e.g., x--), the value of the variable is used in the expression first, and then it is decremented by 1.

7. Examine the use of ternary or conditional operator.

The ternary operator, also known as the conditional operator, is a shorthand way of writing an if-else statement in a single expression. It's a concise and efficient way to express conditional logic.

Syntax of the Ternary Operator:

```
condition ? expression_if_true : expression_if_false;
```

- **condition**: This is the expression that will be evaluated (it returns either true or false).
- **expression_if_true**: This is the value or expression that will be returned/executed if the condition is true.
- **expression_if_false**: This is the value or expression that will be returned/executed if the condition is false.

8.Differentiate between variable and identifier.

Variable and identifier are often used interchangeably in programming, but they have slightly different meanings.

Identifier:

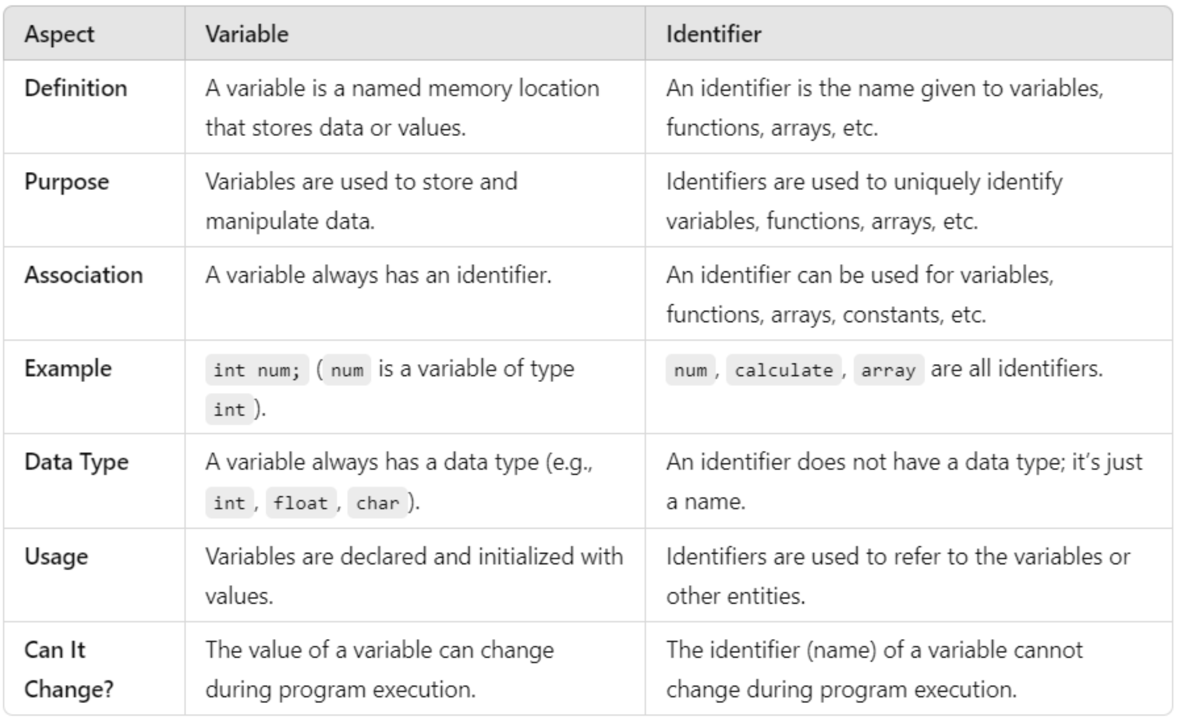
- A name given to a program entity such as a variable, function, constant, or structure.
- Used to refer to that entity within the program.
- Can be any combination of letters, digits, and underscores, but must start with a letter or underscore.

Variable:

- A specific type of identifier that represents a storage location in memory.
- Holds a value of a particular data type (e.g., integer, float, character).
- Can be assigned different values throughout the program's execution.

In essence:

- All variables are identifiers, but not all identifiers are variables.
- Functions, constants, and structures are also identifiers but not variables.



Aspect	Variable	Identifier
Definition	A variable is a named memory location that stores data or values.	An identifier is the name given to variables, functions, arrays, etc.
Purpose	Variables are used to store and manipulate data.	Identifiers are used to uniquely identify variables, functions, arrays, etc.
Association	A variable always has an identifier.	An identifier can be used for variables, functions, arrays, constants, etc.
Example	`int num;` (`num` is a variable of type `int`).	`num` , `calculate` , `array` are all identifiers.
Data Type	A variable always has a data type (e.g., `int` , `float` , `char`).	An identifier does not have a data type; it's just a name.
Usage	Variables are declared and initialized with values.	Identifiers are used to refer to the variables or other entities.
Can It Change?	The value of a variable can change during program execution.	The identifier (name) of a variable cannot change during program execution.

9.Examine the concept of preprocessor directives.

Preprocessor directives are special instructions that are processed before the actual compilation of a C program. They provide a way to control the preprocessing stage, which involves tasks like including header files, defining macros, and conditional compilation.

Common Preprocessor Directives:

1. **#include:**
    - Includes the contents of a specified header file into the current source file.
    - Used to access predefined functions, data types, and macros.
    - Example: #include <stdio.h>
2. **#define:**
    - Defines a macro, which is a textual substitution that occurs during preprocessing.
    - Used to create constants, define functions, or simplify complex expressions.
    - Example: #define PI 3.14159
3. **#ifdef, #ifndef, #else, #endif:**
    - Used for conditional compilation, allowing parts of the code to be included or excluded based on certain conditions.

10. Discuss the concept of type conversion in c

Type conversion in C refers to the process of converting a value of one data type to another. This is often necessary when performing operations that involve different data types or when storing values in variables of different types.

There are two main types of type conversion in C:

Implicit Type Conversion (Automatic Type Conversion)

- **Occurs automatically** by the compiler when it determines that the conversion is safe and reasonable.

  Example: int x = 5; float y = 2.5; float z = x + y; // Implicit conversion of x to float

Explicit Type Conversion (Casting)

- **Requires a type cast operator** ((type)) to explicitly specify the desired data type.
- **Used when** you want to control the conversion process or when the implicit conversion might not be the desired behavior.

  Example:

  int x = 5;float y = 2.5;int z = (int)(x + y); // Explicit conversion of the result to int

Descriptive Questions (16 Marks)

1.(i). Explain the characteristics and need of an algorithm. (8 Mark)

An **algorithm** is a step-by-step procedure or a set of rules for solving a specific problem or performing a task. Algorithms are essential in computer science and programming, as they provide a clear and systematic way to process data and reach a solution.

Characteristics of an Algorithm:

1. **Well-Defined Inputs**:
   - An algorithm should have clearly defined inputs, which are the values or data it will process. The inputs should be specified in a manner that is understandable and unambiguous.
2. **Well-Defined Outputs**:
   - An algorithm must produce one or more outputs based on the given inputs. The outputs should be the result of the algorithm's operations, and their meaning should be clear.
3. **Finiteness**:
   - An algorithm must always terminate after a finite number of steps. It should not go into an infinite loop and must reach a conclusion or result.
4. **Effectiveness**:
   - Each step of an algorithm should be simple and clear enough to be carried out in a finite amount of time. The operations involved should be basic enough to be performed with available resources.
5. **Generality**:
   - An algorithm should be applicable to a broad set of problems, not just a specific instance. It should be able to handle all possible valid inputs and produce correct outputs.
6. **Unambiguous**:
   - The steps in an algorithm must be clearly stated, leaving no room for ambiguity. Each instruction should be precise and easy to understand.
7. **Step-by-Step Procedure**:
   - An algorithm consists of a sequence of well-defined steps that need to be followed in order to achieve the desired result. These steps should be organized logically.

Need for Algorithms:

1. **Problem Solving**:
    - Algorithms provide a systematic way to approach and solve problems. They break down complex problems into manageable steps, making it easier to find solutions.
2. **Efficiency**:
    - Well-designed algorithms help improve the efficiency of a program or process by optimizing resource usage (time and space). An efficient algorithm can handle large inputs quickly and effectively.
3. **Consistency**:
    - Algorithms ensure that the same inputs will yield the same outputs every time they are executed. This consistency is crucial for reliable software development.
4. **Reusability**:
    - Algorithms can be reused in different programs or applications, which saves time and effort in coding. Once an algorithm is developed, it can be implemented in various contexts without needing to start from scratch.
5. **Documentation**:
    - Algorithms serve as documentation for how a problem is solved. They provide clarity on the logic and methodology used in a program, which can be helpful for future reference or for other developers.
6. **Basis for Programming**:
    - Algorithms are the foundation of programming. A programmer needs to understand the algorithm before writing code. The algorithm provides a blueprint for implementing the solution in a programming language.
7. **Facilitating Communication**:
    - Algorithms provide a common language for discussing and analyzing problems among developers. They can be expressed in pseudocode or flowcharts, making it easier for teams to collaborate on solutions.
8. **Adaptability**:
    - Algorithms can often be modified or extended to accommodate changes in requirements or improvements in technology, making them versatile for evolving needs.

Example:

Here’s a simple algorithm for finding the maximum of two numbers:

Algorithm:

1. Start
2. Input two numbers, a and b
3. If a is greater than b, then:
    - Output a as the maximum
4. Else:
    - Output b as the maximum
5. End

1.(ii). Write an algorithm to find the first N natural numbers.(8 Mark)

Algorithm:

1. **Start**
2. **Read the value of N (the number of natural numbers to be found)**
3. **Initialize a variable i to 1 (to represent the first natural number)**
4. **Repeat steps 5-7 until i becomes greater than N 5. Print the value of i 6. Increment i by 1**
5. **End**

Explanation:

This algorithm follows a simple iterative approach:

1. **Start: The algorithm begins execution.**
2. **Read the value of N: The user is prompted to enter the value of N, which specifies the number of natural numbers to be printed.**
3. **Initialize a variable i to 1: A variable i is created and assigned the value 1. This variable will be used to keep track of the current natural number being printed.**
4. **Repeat steps 5-7 until i becomes greater than N: This step sets up a loop that will continue to execute as long as the value of i is less than or equal to N.**
   - **Print the value of i: The current value of i is printed to the console, representing the current natural number.**
   - **Increment i by 1: The value of i is increased by 1, moving to the next natural number.**
5. **End: The algorithm terminates after the loop has finished executing, having printed the first N natural numbers.**

This algorithm effectively prints the first N natural numbers by starting with 1 and incrementing a counter variable until it reaches N.

Example:

If N is 5, the algorithm will print the first 5 natural numbers: 1, 2, 3, 4, 5.

2. Describe the structure of a C program with an example.(16 Mark)

The structure of a C program consists of several key components that are arranged in a specific order. Understanding this structure is essential for writing correct and efficient C programs.
Simple C Program:

```
/* Simple C program to display the
text */
#include<stdio.h>:
```

```
int main()
{
printf("good morning\n");
return(0);
}
```

/*……..*/

A **comment** starts with star and asterisk **/* and ends with** asterisk and start ***/** .
comment lines used to give description and it is ignored by compiler during execution.

#include<stdio.h>

This <stdio.h> is called header file that has functions used for reading input and generating output.

In order to use **printf function,** we need to include the required file in our program using #include preprocessor.

int main()

C program consists of functions with required main() function as a entry point of program. It is where, **actual execution starts.**

A program can have **only one main function** and any number of sub functions.

Curly brackets { …. }

Indicates **block of code**.

we can have statements inside the brackets to determine what the function does when executed.

printf("good morning\n");

printf is a build-in functions which is used to display output in the screen.

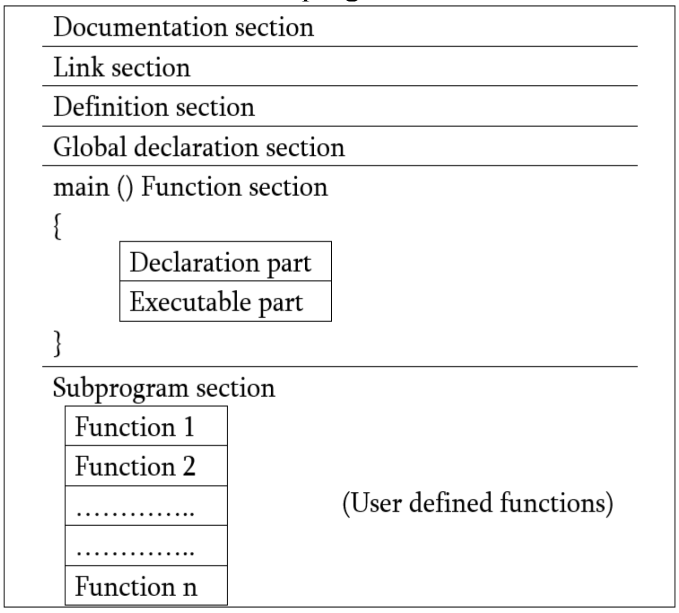
\n is called escape sequence to go to next line It always begin with \ (back slash).

Semicolon(;)

It is used to terminate the statement.

return(0);

- This statement terminates the main function.
- Returns the value 0 to the operating system indicating Successful completion of program.
- Any other number indicates that the program is failed.



Fig, Structure of C program

Documentation Section

- This section consists of set of **comment lines** which gives the name of programmer and other details like time and date of writing the program.
- **Comment lines used to give description and it is ignored by compiler during execution.**
- Documentation section helps anyone to get an overview of the program.

There are two types of comments.

Single line comment (// program for addition)

Multiple line comment(/* ……..*/)

Link Section

- The link section consists of the **header files of the functions** that are used in the program.
- It provides instructions to the compiler to **link functions from the system library.**
- Some of the header files are,

#include<stdio.h>
#include<conio.h>
#include<math.h>
Here, **#include** is the **preprocessor directive.**

Definition Section

All the **symbolic constants** are written in definition section.
Macros are known as symbolic constants.
Ex: #define PI 3.14

Global Declaration Section

The variables that are used in more than one function throughout the program are called global variables. Global variables declared outside of all the functions.

main() Function Section

Every C program must have one main() function, which specifies the starting point of 'C' program.
Main function contains two parts,
i.Declaration part
ii.Executable part

- The declaration part contains all the variables that are used in executable part.
- These two parts must be written in between the opening and closing braces.
- Each statement in the declaration and executable part must end with a semicolon (;).
- The execution of program starts at opening braces"{"and ends at closing braces"}".

Subfunction(Sub program) Section

Subprogram section contains all the user defined functions that are placed after the main function.

3.Explain the different types of operators used in 'C' with an example.(16 Mark).

Operators are symbols used to perform operations on operands (values or variables). C supports a wide range of operators, categorized as follows:

Arithmetic Operators
Used for mathematical calculations.

- **Addition:** + (e.g., x + y)
- **Subtraction:** - (e.g., x - y)
- **Multiplication:** * (e.g., x * y)
- **Division:** / (e.g., x / y)
- **Modulo:** % (returns the remainder of division) (e.g., x % y)

Relational Operators
Used to compare values.

- **Equal to:** == (e.g., x == y)
- **Not equal to:** != (e.g., x != y)
- **Greater than:** > (e.g., x > y)
- **Less than:** < (e.g., x < y)
- **Greater than or equal to:** >= (e.g., x >= y)
- **Less than or equal to:** <= (e.g., x <= y)

Logical Operators
Used to combine logical expressions.

- **Logical AND:** && (e.g., x > 0 && y < 10)
- **Logical OR:** || (e.g., x == 0 || y == 0)
- **Logical NOT:** ! (e.g., !x)

Assignment Operators
Used to assign values to variables.

- **Simple assignment:** = (e.g., x = 5)
- **Compound assignment:** +=, -=, *=, /=, %= (e.g., x += 2)

Increment/Decrement Operators
Used to increase or decrease the value of a variable by 1.

- **Increment:** ++ (e.g., ++x, x++)
- **Decrement:** -- (e.g., --x, x--)

Bitwise Operators
Used to perform operations on individual bits of a value.

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

Conditional Operator (Ternary Operator)
A shorthand way to write an if-else statement.

- **Syntax:** condition ? expression1 : expression2

Comma Operator
Used to evaluate multiple expressions and return the value of the last expression.
Example program

```
#include <stdio.h>


int main() {

   int a = 10, b = 5;

   float x = 3.14, y = 2.71;


   // Arithmetic operators

   int sum = a + b;

   int difference = a - b;

   int product = a * b;

   int quotient = a / b;

   int remainder = a % b;


   // Relational operators

   int is_equal = a == b;

   int is_not_equal = a != b;

   int is_greater = a > b;

   int is_less = a < b;

   int is_greater_equal = a >= b;

   int is_less_equal = a <= b;


   // Logical operators
```

```
int logical_and = (a > 0) && (b > 0);
int logical_or = (a == 0) || (b == 0);
int logical_not = !a;

// Assignment operators
a += 2;
b -= 3;
x *= 2;
y /= 3;
remainder %= 2;

// Increment/decrement operators
int pre_increment = ++a;
int post_increment = b++;
int pre_decrement = --x;
int post_decrement = y--;

// Bitwise operators (assuming a and b are unsigned)
int bitwise_and = a & b;
int bitwise_or = a | b;
int bitwise_xor = a ^ b;
int bitwise_not = ~a;
int left_shift = a << 2;
int right_shift = b >> 1;

// Conditional operator
int max = (a > b) ? a : b;

// Comma operator
int result = (a++, b--, x *= 2, y /= 3);

// Print the results
printf("Arithmetic:\n");
printf("Sum: %d\n", sum);
printf("Difference: %d\n", difference);
printf("Product: %d\n", product);
```

```
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);


    // ... (continue printing for other operators)


    return 0;
}
```

Output

```
Arithmetic:
Sum: 15
Difference: 5
Product: 50
Quotient: 2
Remainder: 0

Relational:
is_equal: 0
is_not_equal: 1
is_greater: 1
is_less: 0
is_greater_equal: 1
is_less_equal: 0

Logical:
logical_and: 1
logical_or: 0
logical_not: 0

Assignment:
a: 12
b: 2
x: 6.28
y: 0.903333

Increment/Decrement:
pre_increment: 13
post_increment: 2
pre_decrement: 5.28
post_decrement: 0.903333

Bitwise:
bitwise_and: 0
bitwise_or: 15
bitwise_xor: 15
bitwise_not: -11
left_shift: 40
right_shift: 2

Conditional:
max: 10

Comma:
result: 0.903333
```

4. Discuss the various Conditional Statements used in C with its syntax and program.

Conditional Statements in C

- Conditional statements are used to control the flow of execution in a C program based on certain conditions. They allow you to make decisions and execute different code blocks depending on the outcome of these conditions.
- **1. if Statement**
- **Syntax:**

```
if (condition) {
   // Statements to be executed if the condition is true
}
```

- **Example:**

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

2. if-else Statement

- **Syntax:**

```
if (condition) {
   // Statements to be executed if the condition is true
} else {
   // Statements to be executed if the condition is false

}
```

- **Example:**

```
int number = 10;
if (number % 2 == 0) {
   printf("The number is even.\n");
} else {
   printf("The number is odd.\n");
}
```

3. if-else if-else Statement

- **Syntax:**

```
if (condition1) {
   // Statements to be executed if condition1 is true
} else if (condition2) {
   // Statements to be executed if condition2 is true
} else
 {
   // Statements to be executed if none of the conditions are true
}
```

- **Example:**

```
int grade = 85;
if (grade >= 90) {
   printf("Grade: A\n");
} else if (grade >= 80) {
   printf("Grade: B\n");
} else if (grade >= 70) {
   printf("Grade: C\n");
} else {
   printf("Grade: F\n");
}
```

4. switch Statement

- **Syntax:**

```
switch (expression) {
   case value1:
      // Statements to be executed if expression == value1
      break;
   case value2:
      // Statements to be executed if expression == value2
      break;
   // ...
   default:
      // Statements to be executed if none of the cases match
}
```

- **Example:**

```
int day = 3;
switch (day) {
   case 1:
      printf("Monday\n");
      break;
   case 2:
      printf("Tuesday\n");
      break;
   // ...
   default:
      printf("Invalid day\n");
}
```

5. Classify various looping statements in c with an example program.(16 Marks)

Looping statements in C allow the execution of a block of code multiple times based on a specified condition. The primary looping statements in C are:

1. **for Loop**
2. **while Loop**
3. **do-while Loop**

1. for Loop

The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and iteration expression.

Syntax:

```
for (initialization; condition; iteration) {
   // code to be executed
}
```

Example:

```
#include <stdio.h>

int main() {
   int i;

   printf("Counting from 1 to 5:\n");
```

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

    return 0;
}
```
2. while Loop
The while loop is used when the number of iterations is not known in advance, and the loop continues as long as the condition is true.
Syntax:

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

```
#include <stdio.h>

int main() {
    int i = 1;

    printf("Counting from 1 to 5 using while loop:\n");
    while (i <= 5) {
        printf("%d\n", i);
        i++; // incrementing i
    }

    return 0;
}
```
3. do-while Loop
The do-while loop is similar to the while loop, but it guarantees that the code block will execute at least once, as the condition is checked after the code execution.
Syntax:

```
do {
    // code to be executed
} while (condition);
```
Example:

```
#include <stdio.h>

int main() {
    int i = 1;

    printf("Counting from 1 to 5 using do-while loop:\n");
    do {
        printf("%d\n", i);
        i++; // incrementing i
    } while (i <= 5);

    return 0;
}
```

C provides three primary looping statements:

- **for Loop**: Ideal when the number of iterations is known beforehand.
- **while Loop**: Suitable when the number of iterations is unknown, and it checks the condition before each iteration.

- **do-while Loop**: Similar to the while loop but ensures that the code block executes at least once since it checks the condition after the loop's body.

These looping constructs help implement repetitive tasks efficiently in C programs.

6.i)Develop a C program to check whether a number is prime or not. (8 Marks)

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. In other words, a prime number is a number that cannot be divided evenly by any other natural number except 1 and itself.

Here are some examples of prime numbers:

- 2
- 3
- 5
- 7

Program:

```
#include <stdio.h>

int main() {
   int num, i, isPrime = 1;

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

   if (num <= 1) {
      isPrime = 0;
   } else {
      for (i = 2; i <= num / 2; i++) {
         if (num % i == 0) {
            isPrime = 0;
            break;
         }
      }
   }

   if (isPrime) {
      printf("%d is a prime number.\n", num);
   } else {
      printf("%d is not a prime number.\n", num);
   }

   return 0;
}
```

Output:

For example, if the user enters 7, the output will be:

```
7 is a prime number.
```

.

For example, if the user enters 10, the output will be:

```
10 is not a prime number.
```

This program works as follows:

1. **Input:** The user is prompted to enter a positive integer.
2. **Initialization:** A flag variable isPrime is initialized to 1. This indicates that the number is initially assumed to be prime.

3. **Special Case:** If the input number is less than or equal to 1, it's not prime, so the isPrime flag is set to 0.
4. **Loop:** The program iterates from 2 to the square root of the number. If the number is divisible by any of these numbers, it's not prime, so the isPrime flag is set to 0 and the loop is terminated.
5. **Output:** Finally, the program checks the value of isPrime. If it's still 1, the number is prime; otherwise, it's not.

This program efficiently checks if a number is prime by iterating only up to the square root of the number.

6.ii)To convert the temperature given in Fahrenheit to Celsius and its vice versa.(8 marks)

The program prompts the user to choose the conversion type (Celsius to Fahrenheit or Fahrenheit to Celsius) and then performs the conversion based on the user's input.
Program:

```
#include <stdio.h>

int main() {
   float temperature, convertedTemperature;
   int choice;

   // Display conversion options
   printf("Temperature Conversion Menu:\n");
   printf("1. Celsius to Fahrenheit\n");
   printf("2. Fahrenheit to Celsius\n");
   printf("Enter your choice (1 or 2): ");
   scanf("%d", &choice);

   // Perform conversion based on user's choice
   if (choice == 1) {
      // Celsius to Fahrenheit
      printf("Enter temperature in Celsius: ");
      scanf("%f", &temperature);
      convertedTemperature = (temperature * 9/5) + 32; // Conversion formula
      printf("%.2f Celsius is equal to %.2f Fahrenheit\n", temperature,
convertedTemperature);
   } else if (choice == 2) {
      // Fahrenheit to Celsius
      printf("Enter temperature in Fahrenheit: ");
      scanf("%f", &temperature);
      convertedTemperature = (temperature - 32) * 5/9; // Conversion formula
      printf("%.2f Fahrenheit is equal to %.2f Celsius\n", temperature,
convertedTemperature);
   } else {
      // Invalid choice
      printf("Invalid choice! Please enter 1 or 2.\n");
   }

   return 0;
}
```

Output:

```
Temperature Conversion Menu:
1. Celsius to Fahrenheit
2. Fahrenheit to Celsius
Enter your choice (1 or 2): 1
Enter temperature in Celsius: 25
25.00 Celsius is equal to 77.00 Fahrenheit
```

How the Program Works

1. **User Input**: The program presents a menu to the user, allowing them to choose the conversion direction.
2. **Conversion**:
   - If the user chooses to convert Celsius to Fahrenheit, the program reads the temperature in Celsius and applies the formula: F=(C×95)+32F = \left(C \times \frac{9}{5} \right) + 32F=(C×59)+32
   - If the user chooses to convert Fahrenheit to Celsius, the program reads the temperature in Fahrenheit and applies the formula: C=(F−32)×59C = \left(F - 32 \right) \times \frac{5}{9}C=(F−32)×95
3. **Output**: The converted temperature is displayed with two decimal precision.
4. **Validation**: If the user enters an invalid choice, an error message is displayed.

7.Analyze various conditional control Statements with an example program.(16 Marks)

Conditional control statements in C are used to make decisions in your code. They allow the program to execute different blocks of code based on whether certain conditions are true or false. The primary conditional control statements in C are:

1. **if statement**
2. **if-else statement**
3. **else if statement**
4. **nested if statement**
5. **switch statement**
6. **conditional (ternary) operator**

1. if Statement

The if statement executes a block of code if a specified condition is true.

Syntax:

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

Example:

```
#include <stdio.h>

int main() {
   int a = 10;

   if (a > 0) {
      printf("a is positive.\n");
   }

   return 0;
}
```

2. if-else Statement

The if-else statement allows you to execute one block of code if the condition is true and another block if it is false.

Syntax:

```
if (condition) {
    // code if condition is true
} else {
    // code if condition is false
}
```
Example:

```
#include <stdio.h>

int main() {
    int a = -5;

    if (a > 0) {
        printf("a is positive.\n");
    } else {
        printf("a is not positive.\n");
    }

    return 0;
}
```
3. else if Statement
You can chain multiple conditions using else if statements.
Syntax:
```
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition2 is true
} else {
    // code if both conditions are false
}
```
Example:
```
#include <stdio.h>

int main() {
    int a = 0;

    if (a > 0) {
        printf("a is positive.\n");
    } else if (a < 0) {
        printf("a is negative.\n");
    } else {
        printf("a is zero.\n");
    }

    return 0;
}
```
4. Nested if Statement
You can use if statements inside other if statements, which is called nesting.
Syntax:
```
if (condition1) {
    if (condition2) {
        // code if both condition1 and condition2 are true
    }
}
```
Example:
```
#include <stdio.h>

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

    if (a > 0) {
        if (b > 0) {
            printf("Both a and b are positive.\n");
        }
    }

    return 0;
}
```

5. switch Statement

The switch statement allows a variable to be tested for equality against a list of values. Each value is called a case.

Syntax:

```
switch (expression) {
    case constant1:
        // code to be executed if expression == constant1
        break;
    case constant2:
        // code to be executed if expression == constant2
        break;
    default:
        // code to be executed if none of the cases are true
}
```

Example:

```
#include <stdio.h>

int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        default:
            printf("Weekend\n");
            break;
    }

    return 0;
}
```

6. Conditional (Ternary) Operator

The ternary operator is a shorthand way of expressing if-else statements.

Syntax:

```
condition ? expression1 : expression2;
```

Example:
```
#include <stdio.h>

int main() {
   int a = 10, b = 20;
   int max = (a > b) ? a : b; // max gets the greater value
   printf("The maximum value is: %d\n", max); // Output: 20

   return 0;
}
```
Combined Example Program

Here is a combined example that incorporates various conditional control statements to determine the grade based on a student's score.
```
#include <stdio.h>

int main() {
   int score;
   char grade;

   // Input the student's score
   printf("Enter the student's score (0-100): ");
   scanf("%d", &score);

   // Check for valid score
   if (score < 0 || score > 100) {
      printf("Invalid score! Please enter a score between 0 and 100.\n");
   } else {
      // Determine the grade
      if (score >= 90) {
         grade = 'A';
      } else if (score >= 80) {
         grade = 'B';
      } else if (score >= 70) {
         grade = 'C';
      } else if (score >= 60) {
         grade = 'D';
      } else {
         grade = 'F';
      }

      // Output the grade
      printf("The grade is: %c\n", grade);
   }

   return 0;
}
```

8. Examine the concept of storage classes in c with an example program.(16 Marks)

storage classes define the scope, visibility, and lifetime of variables and functions. There are four primary storage classes in C:

1. **Automatic Storage Class (auto)**
2. **Register Storage Class (register)**
3. **Static Storage Class (static)**
4. **External Storage Class (extern)**

1. Automatic Storage Class (auto)

- **Scope**: Local to the block in which it is defined.
- **Lifetime**: Exists only during the execution of the block.
- **Default Storage Class**: Variables declared inside a function are automatic by default.

Example:

```
#include <stdio.h>

void func() {
   auto int x = 10; // 'auto' is optional here
   printf("Inside func, x = %d\n", x);
}

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

2. Register Storage Class (register)

- **Scope**: Local to the block in which it is defined.
- **Lifetime**: Exists only during the execution of the block.
- **Special Feature**: Suggests to the compiler to store the variable in a CPU register for faster access. However, this is not guaranteed.

Example:

```
#include <stdio.h>

int main() {
   register int counter;

   for (counter = 0; counter < 5; counter++) {
      printf("Counter: %d\n", counter);
   }

   return 0;
}
```

3. Static Storage Class (static)

- **Scope**: Local to the block in which it is defined but retains its value between function calls.
- **Lifetime**: Exists for the duration of the program.
- **Visibility**: The variable is not visible outside the block.

Example:

```
#include <stdio.h>

void func() {
   static int count = 0; // Static variable
   count++;
   printf("Function called %d times.\n", count);
}

int main() {
   func();
   func();
   func(); // Output will show the count of function calls
   return 0;
}
```

4. External Storage Class (extern)

- **Scope**: Global visibility across multiple files.
- **Lifetime**: Exists for the duration of the program.

- **Purpose**: Used to declare a global variable or function that is defined in another file or elsewhere in the same file.

Example:
```
#include <stdio.h>

extern int globalVar; // Declaration of an external variable

void func() {
   printf("Value of globalVar in func: %d\n", globalVar);
}

int globalVar = 100; // Definition of the external variable

int main() {
   printf("Value of globalVar in main: %d\n", globalVar);
   func();
   return 0;
}
```

9. Distinguish between branching and looping statements used in c programming.(16 Marks)

- In C programming, **branching statements** and **looping statements** are two fundamental types of control statements used to control the flow of execution in a program. Here's a detailed distinction between the two:

Branching Statements
- Branching statements allow the program to make decisions and execute different blocks of code based on certain conditions. They direct the flow of execution to different parts of the program, depending on the evaluation of conditions.

Characteristics:
- **Purpose**: To choose between different paths of execution.
- **Execution**: Only one block of code is executed based on the condition(s).
- **Examples**: if, if-else, else if, switch, goto.

Examples:
1. **if Statement**:
```
int a = 10;
if (a > 0) {
   printf("a is positive.\n");
}
```
2. **if-else Statement**:
```
int a = -5;
if (a > 0) {
   printf("a is positive.\n");
} else {
   printf("a is not positive.\n");
}
```
3. **switch Statement**:
```
int day = 3;
switch (day) {
   case 1: printf("Monday\n"); break;
   case 2: printf("Tuesday\n"); break;
   case 3: printf("Wednesday\n"); break;
   default: printf("Invalid day\n"); break;
}
```
Looping Statements

- Looping statements are used to execute a block of code multiple times, based on a condition. They allow repeated execution of code as long as a certain condition is true.

Characteristics:

- **Purpose**: To perform repetitive tasks until a condition is met.
- **Execution**: A block of code is executed multiple times.
- **Examples**: for, while, do-while.

Examples:

1. **for Loop**:

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

2. **while Loop**:

```
int i = 0;
while (i < 5) {
   printf("%d\n", i);
   i++;
}
```

3. **do-while Loop**:

```
int i = 0;
do {
   printf("%d\n", i);
   i++;
} while (i < 5);
```

Comparison

Aspect	**Branching Statements**	**Looping Statements**
Purpose	To make decisions and choose paths	To repeat code execution
Execution	Executes one block of code based on a condition	Executes a block of code multiple times based on a condition
Examples	if, if-else, switch, goto	for, while, do-while
Control Flow	Can alter the flow to various parts of the code	Continuously runs until a condition is false

10.i) Develop a C program To check whether the given number is palindrome or not.(8 Marks)

A number is considered a palindrome if it remains the same when its digits are reversed.
C Program to Check Palindrome

```
#include <stdio.h>

int main() {
   int num, reversed = 0, original, remainder;

   // Input the number
   printf("Enter an integer: ");
   scanf("%d", &num);

   original = num; // Store the original number for comparison

   // Reverse the number
   while (num != 0) {
      remainder = num % 10; // Get the last digit
      reversed = reversed * 10 + remainder; // Build the reversed number
```

```
        num /= 10; // Remove the last digit
    }

    // Check if the original number and reversed number are the same
    if (original == reversed) {
        printf("%d is a palindrome.\n", original);
    } else {
        printf("%d is not a palindrome.\n", original);
    }

    return 0;
}
```

Output:

```
Enter an integer: 121
121 is a palindrome.

Enter an integer: 123
123 is not a palindrome.
```

Explanation of the Program

1. Input the Number: The program prompts the user to enter an integer.
2. Store Original Number: The original number is stored in a variable original for later comparison.
3. Reverse the Number: A while loop is used to reverse the digits of the number:
   - The last digit is extracted using num % 10.
   - The reversed number is constructed by multiplying the current reversed value by 10 and adding the last digit.
   - The last digit is removed from num using integer division (num /= 10).
4. Check for Palindrome: Finally, the program compares the original number with the reversed number:
   - If they are the same, it prints that the number is a palindrome.
   - Otherwise, it indicates that the number is not a palindrome.

10.ii) To check whether the given number is an armstrong or not.

An Armstrong number is a number that is equal to the sum of the cubes of its digits. For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

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

int main() {
    int num, originalNum, remainder, result = 0, count = 0;

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

    if (num <= 0) {
        printf("Please enter a positive integer.\n");
    } else {
        originalNum = num;
        // Count the number of digits
        while (originalNum != 0) {
            originalNum /= 10;
            count++;
        }

        originalNum = num;

        // Calculate the sum of the cubes of digits
        while (originalNum != 0) {
            remainder = originalNum % 10;
            result += pow(remainder, count);
            originalNum /= 10;
        }

        // Check if the number is an Armstrong number
        if (num == result) {
            printf("%d is an Armstrong number.\n", num);
        } else {
            printf("%d is not an Armstrong number.\n", num);
        }
    }

    return 0;
}
```

This program works as follows:

1. Input: The user is prompted to enter a positive integer.
2. Counting Digits: The number of digits in the input number is calculated.
3. Calculating Sum of Cubes: Each digit is extracted, raised to the power of the number of digits, and added to a sum.
4. Comparison: If the sum is equal to the original number, it's an Armstrong number; otherwise, it's not.

Output:
For example, if the user enters 153, the output will be:

```
153 is an Armstrong number.
```

For example, if the user enters 10, the output will be:

```
10 is not an Armstrong number.
```