SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM, CHENNAI

COURSE CODE: 21CSS101J

COURSE NAME: PROGRAMMING FOR PROBLEM SOLVING

UNIT -1

SYLLABUS:

Evolution of Programming & Languages - Problem solving through programming - Writing algorithms & Pseudo code - Single line and multiline comments - Introduction to C: Structure of the C program - Input and output statements. Variables and identifiers, Constants, Keywords - Values, Names, Scope, Binding, Storage Classes - Numeric Data types: integer, floating point Non-Numeric Data types: char and string - L value and R value in expression, Increment and decrement operator - Comma, Arrow and Assignment operator, Bitwise and Size-of operator - Arithmetic, Relational and logical Operators - Condition Operators, Operator Precedence - Expressions with pre / post increment operator

1.1 Evolution of Programming & Languages :

Programming languages have evolved to meet the changing needs of developers, technology advancements, and shifts in programming paradigms. Here's a brief overview of the major stages in the evolution of programming languages:

- **Machine Language and Assembly Language (1940s-1950s):** The earliest programming languages were essentially machine languages, which consisted of binary code that directly controlled the computer's hardware. Assembly languages were developed as a step up from machine languages, providing symbolic representations of machine instructions to make programming more readable and manageable.

- **Fortran (1957):** Developed by IBM, Fortran (short for "Formula Translation") was one of the first high-level programming languages. It aimed to make scientific and engineering computations easier by using English-like commands.

- **COBOL (1959):** COBOL (Common Business-Oriented Language) was designed for business data processing. It focused on being human-readable and business-oriented, making it easier to create programs for business applications.

- **LISP (1958):** LISP (LISt Processing) was designed for artificial intelligence research. It introduced the concept of symbolic processing and is still used in AI and symbolic • computing.

- **ALGOL (1958):** ALGOL (ALGOrithmic Language) aimed to be a universal language for scientific computation. It influenced the development of subsequent languages and introduced concepts like block structure and nested functions.

- **BASIC (1964):** Beginner's All-purpose Symbolic Instruction Code (BASIC) was designed to make programming more accessible to beginners. It had a simple syntax and was often used for educational purposes.

- **C (1972):** C was developed at Bell Labs and became a widely-used programming language due to its efficiency and portability. It influenced the development of many later languages and is still popular today.

- **Pascal (1970s):** Designed by Niklaus Wirth, Pascal aimed to teach structured programming concepts. It emphasized clarity and modularity in code.

- **C++ (1983):** C++ was an extension of C that introduced object-oriented programming (OOP) features. It combined procedural programming with OOP, allowing for better code organization and reuse.

- **Python (1991):** Python was designed to be readable and emphasize code simplicity. Its clear syntax and extensive standard library made it a popular choice for various applications, from scripting to web development and data analysis.

- **Java (1995):** Developed by Sun Microsystems, Java aimed to be platform-independent through the "write once, run anywhere" philosophy. It introduced the concept of bytecode and the Java Virtual Machine (JVM).

- **C# (2000):** Developed by Microsoft, C# was designed as a modern programming language for building Windows applications and services. It combined features from C++, Java, and other languages.

- **JavaScript (1995):** Despite the name similarity to Java, JavaScript is a completely different language designed for web development. It's used for adding interactivity and dynamic behavior to web pages.

- **Ruby (1995):** Ruby focused on simplicity and programmer productivity. It gained popularity with the Ruby on Rails web framework, known for rapid web application development.

- **Swift (2014):** Created by Apple, Swift aimed to be a safer and more modern alternative to Objective-C for iOS and macOS app development.

- **Rust (2010):** Rust emphasized memory safety and system-level programming without sacrificing performance. It was designed to prevent common programming errors like null pointer dereferences.

1.2 Problem solving through programming:

Programming languages and computational thinking to analyze, design, and implement solutions to various challenges. Here's a step-by-step approach to problem-solving through programming:

1. **Understand the Problem:**

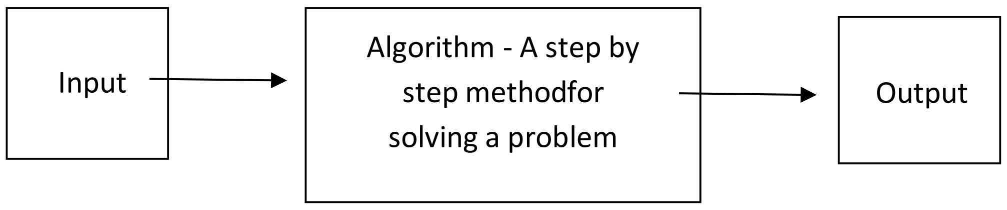
    - Clearly define the problem you need to solve. Break it down into smaller, manageable components.
    - Gather any requirements, constraints, or specifications related to the problem.

2. **Analyze the Problem:**
   - Examine the problem thoroughly to understand its context and the desired outcome.
   - Identify the input data or variables needed and the expected output.
3. **Plan Your Approach:**
   - Decide on the appropriate programming language for the task.
   - Choose the appropriate algorithms, data structures, and programming paradigms (e.g., procedural, object-oriented, functional) based on the problem's nature.
4. **Design Your Solution:**
   - Create a high-level design of your program's structure, including the main steps, functions, and classes if applicable.
   - Design data structures to store and manipulate the necessary data.
5. **Break Down the Problem:**
   - Divide the problem into smaller sub-problems or tasks that can be tackled individually.
   - Identify dependencies between these sub-problems.
6. **Implement the Solution:**
   - Write the code according to your design, addressing each sub-problem one at a time.
   - Use meaningful variable and function names, and follow coding standards for readability.
7. **Debug and Test:**
   - Test your code thoroughly with different input data, including edge cases and corner cases.
   - Debug and fix any issues that arise during testing.
8. **Validate Your Solution:**
   - Verify that your program produces the expected output for various input scenarios.
   - Check if your program meets the requirements and constraints provided.

1.3 (A)Writing Algorithms:

An algorithm is defined as a step by step procedure for solving a problem. It is a ordered set of rules to solve a problem. An algorithm is a representation of a solution to a problem. It is a well-defined computational procedure consisting of a set of instructions that takes some value or set of values, as input, and produces some value or set of values, as output

The following are the steps to create any algorithm.



1.3.1 Properties of Algorithms

Every algorithm must have five essential properties:

(1) **Inputs specified**: An algorithm must have zero or more inputs, We must specify the type of the data, the amount of data, and the form that the data will take.

(2) **Outputs specified**: An algorithm has one or more outputs, which have a specified relation to the inputs.

(3) **Definiteness**: Every detail of each step must be clearly specified.

(4) **Effectiveness**: All operations to be performed must be sufficiently basic that they can bedone exactly and in finite length.

(5) **Finiteness**: An algorithm must always terminate after a finite number of steps.

1.3.2 Characteristics of Algorithm

Finiteness

Algorithm must terminate after a finite number of steps and further each steps must be executable in finite amount of time.

Definiteness

Instruction must be clear, user defined & Precise. There should not be any ambiguity.

Input

An algorithm has zero or more, but only finite number of inputs, zero inputs. Eg: ASCII character of 0-225.

Output

An algorithm has one or more output.

Effectiveness

An algorithm should be effective that means each of the operation to be preformed in algorithm should be computer programming language independent.

Uniqueness

Result or each steps are uniquely defined and only depend on the input and result of the preceding steps.

Generality

Algorithm should apply to set of input, rather than single input.

Feasibility

It must be possible to perform each instructions.

Output

An algorithm has one or more output.

Effectiveness

An algorithm should be effective that means each of the operation to be preformed in algorithm should be computer programming language independent.

Uniqueness

Result or each steps are uniquely defined and only depend on the input and result of the preceding steps.

Generality

Algorithm should apply to set of input, rather than single input.

Feasibility

It must be possible to perform each instructions.

1.3.3 Method for Developing an Algorithm

1.3.3.1 Define the problem: State the problem to be solved in clear and concise manner.

1.3.3.2 List the inputs and outputs

1.3.3.3 Describe the steps needed to convert input to output

1.3.3.4 Test the algorithm: Choose input data and verify that the algorithm works.

EXAMPLE ALGORITHMS:

A) Develop an algorithm for addition of two numbers

```
Algorithm: AddTwoNumbers
Input: Two integers, num1 and num2
Output: The sum of num1 and num2

1. Start
2. Read num1 and num2 from the user.
3. Set sum = num1 + num2.
4. Display the value of sum.
5. End.
```

When we compile and run the program,

Enter the Input data as follows:

num1= 10

num2= 20

then, the output is displayed as 20

B) Develop an algorithm to find out whether the given number is Even or Odd

Algorithm: CheckEvenOdd
Input: An integer number, num
Output: A message indicating whether num is even or odd

```
1. Start
2. Read the integer num from the user.
3. If num % 2 equals 0, then:
   1. Display "The number num is even."
   2. Go to step 4.
4. Else:
   1. Display "The number num is odd."
5. End.
```

When we compile and run the program,

Enter the Input data as follows:

num = 26

Output is displayed as “Even”

Num=91

Output is displayed as “Odd”

(B)Writing Pseudocode:

Pseudocode (pronounced SOO-doh-kohd) is a kind of structured English for representing algorithms and is a “text-based” detail design tool for human understanding rather than machine understanding. It allows the designer to concentrate on the logic of the algorithm without worrying about the details of language syntax.

Once the pseudo code is prepared, it is rewritten using the syntax and semantic rules of a programming language.

Rules of Pseudo code

- Start with an algorithm and phrase it using words that are easily transcribed intocomputer instructions.
- Indent when you are enclosing instructions within a loop or a conditional clause.
- Avoid words associated with a certain kind of computer language.
- Do not include data declarations in pseudo code.

Three basic constructs for flow of control

1. Sequence
    - Default mode.
    - Used for the sequential execution of statements, one line after another.
2. Selection
    - Used for decisions to choose between two or more alternative paths.
3. Repetition
    - Used for looping to repeat a piece of code several times.

Sequence

It is indicated by writing one statement after another, each statement on a line by itself, and all statements aligned with the same indent. The actions are performed in the order in which they are written, from top to bottom.

Common Keywords

Input: READ, INPUT, OBTAIN, GET

Output: PRINT, OUTPUT, DISPLAY, SHOW

Compute: COMPUTE, CALCULATE, DETERMINE

Initialize: SET, INIT

Add one: INCREMENT, BUMP

Examples

(i) Pseudo code for computing the area of a rectangle.

```
READ height of rectangle
READ width of rectangle
COMPUTE area as height times width
PRINT area
```

(ii) *Pseudo code for computing the average of five numbers.*

```
PRINT  "Enter 5 numbers"
READ n1, n2, n3, n4, n5
PRINT  "The average is"
SET avg to (n1+n2+n3+n4+n5)/5
PRINT  avg
```

Selection

It is a decision (selection) in which a choice is made between two alternative courses of action and it comprises of the following constructs:

- IF-THEN-ELSE
- **CASE...ENDCASE**

IF-THEN-ELSE

Binary choice on a given Boolean condition is indicated by the use of four keywords:

- IF, THEN, ELSE, and ENDIF.

The general form is:

```
IF condition THEN
    sequence 1
ELSE
    sequence 2
ENDIF
```

The ELSE keyword and “sequence 2” are optional. If the condition is true, sequence 1 is performed, otherwise sequence 2 is performed.

Examples

(i) Pseudo code to check whether the number is odd or even.

```
READ number
IF number MOD 2 = 0 THEN DISPLAY
    “Number is Even”
ELSE
    DISPLAY “Number is Odd”
ENDIF
```

(ii) *Pseudo code to check whether the given non-zero number is positive or negative.*

```
READ number
IF num is less than 0 THEN
PRINT num is negative ELSE
PRINT num is positive
ENDIF
```

CASE

CASE is a multiway branch (decision) based on the value of an expression. CASE is a generalization of IF-THEN-ELSE. Four keywords, CASE, OF, OTHERS, and ENDCASE, and conditions are used to indicate the various alternatives.

The general form is:

```
CASE expression OF
    condition 1 : sequence 1
    condition 2 : sequence 2
    ...
    condition n : sequence n
      OTHERS:
    default sequence
ENDCASE
```

The OTHERS clause with its default sequence is optional. Conditions are normally numbers or characters indicating the value of “expression”, but they can be English statements or some other notation that specifies the condition under which the given sequence is to be performed. A certain sequence may be associated with more than one condition.

Examples

(i) Pseudo code for simple calculator

```
  READ n1, n2
  READ choice
CASE choice OF
    +    : PRINT n1+n2
    –    : PRINT n1-n2
    *    : PRINT n1*n2
    /    : PRINT n1/n2
ENDCASE
```

(ii) *Pseudo code for determining gradepoints from grades.*

```
READ grade
CASE  grade  OF
    S    : gradepoint = 10
    A    : gradepoint = 9
```

```
        B   : gradepoint = 8C
        : gradepoint = 7D   :
        gradepoint = 6 E    :
        gradepoint = 5 U    :
        gradepoint = 0
ENDCASE
      DISPLAY gradepoint
```

Repetition

It is a loop (iteration) based on the satisfaction of some condition(s). It comprises of the following constructs:

- WHILE...ENDWHILE
- REPEAT...UNTIL
- FOR...ENDFOR

WHILE...ENDWHILE

It is a loop (repetition) with a simple condition at its beginning. The beginning and ending of the loop are indicated by two keywords WHILE and ENDWHILE.

The general form is:

```
WHILE condition
    sequence
  ENDWHILE
```

The loop is entered only if the condition is true. The "sequence" is performed for each iteration. At the conclusion of each iteration, the condition is evaluated and the loop continues as long as the condition is true.

Examples

(i) Pseudo code to print the numbers from 1 to 100.

```
      n=1
      WHILE n is less than or equal to 100
          DISPLAY n
          INCREMENT n by 1
ENDWHILE
```

(ii) Pseudo code to print the sum of the digits of a given number

```
INPUT a Number INITIALIZE
Sum to zero WHILE Number is
not zero
    COMPUTE Remainder by Number Mod 10
    ADD Remainder to Sum
    DIVIDE Number by 10
ENDWHILE
PRINT Sum
```

REPEAT...UNTIL

It is a loop with a simple condition at the bottom. This loop is similar to the WHILE loop except that the test is performed at the bottom of the loop instead of at the top. Two keywords, REPEAT and UNTIL are used.

The general form is:

```
REPEAT
    sequence
UNTIL condition
```

The "sequence" in this type of loop is always performed at least once, because the test is peformed after the sequence is executed. At the conclusion of each iteration, the condition is evaluated, and the loop repeats if the condition is false. The loop terminates when the condition becomes true.

Examples

(i) Pseudo code to print the numbers from 1 to 100.

```
n=1
REPEAT
    DISPLAY n
    INCREMENT n by 1
UNTIL n is greater than 100
```

(ii) Pseudo code to print the sum of the digits of a given number

```
INPUT a Number
INITIALIZE Sum to zero
```

```
REPEAT
    COMPUTE Remainder by Number Mod 10
    ADD Remainder to Sum
    DIVIDE Number by 10
UNTIL Number is zero
PRINT Sum
```

FOR...ENDFOR

FOR is a “counting” loop. This loop is a specialized construct for iterating a specific number of times, often called a “counting” loop. Two keywords, FOR and ENDFOR are used.

The general form is:

```
FOR iteration bounds
    sequence
ENDFOR
```

Examples

(i) Pseudo code to print the numbers from 1 to 100.

```
FOR n=1 to 100
    DISPLAY n
ENDFOR
```

(ii) *Pseudo code to input ten numbers and print the sum.*

```
INITIALIZE sum to 0
FOR n=1 to 10
    INPUT number
    COMPUTE sum as sum+number
ENDFOR
DISPLAY sum
```

Invoking Subprocedures

To call subprocedures (functions), CALL keyword is used with the following structure:

CALL subprocedurename [**WITH** argumentslist] [**RETURNING** returnvalue]

Here, “**WITH** argumentslist” and “**RETURNING** returnvalue” are optional. “**WITH** argumentslist” is used to call a function with arguments. “**RETURNING** returnvalue” is used when a called function returns a value.

Examples:

CALL SumAndAvg **WITH** NumberList

- Calls a function SumAndAvg with NumberList as an argument. This function does not return any value.

CALL SwapItems **WITH** CurrentItem and TargetItem

- ◦ Calls a function SwapItems with CurrentItem and TargetItem as arguments. This function does not return any value.

CALL getBalance **RETURNING** BalanceAmt

- ◦ Calls a function getBalance which returns BalanceAmt as its return value. This function does not take any argument.

Advantages of Pseudo code

(1) Can be read and understood easily.

(2) Can be done easily on a word processor.

(3) Can be modified easily.

(4) Implements structured concepts well.

(5) Clarifies algorithms in many cases.

(6) Imposes increased discipline on the process of documenting detailed design.

(7) Provides additional level at which inspection can be performed.

(8) Helps to trap defects before they become code.

(9) Increases product reliability.

(10) Converting a pseudocode to a program is simple.

Disadvantages of Pseudo code

(1) Creates an additional level of documentation to maintain.

(2) Introduces error possibilities in translating to code.

(3) May require tool to extract pseudocode and facilitate drawing flowcharts.

(4) There is no standardized format, so one pseudocode may be different from another.

(5) For a beginner, it is more difficult to follow the logic and write pseudocode as comparedto flowchart.

(6) We do not get a picture of the design.

Single line comments and Multiline comments:

SINGLE LINE AND MULTILINE COMMENTS

comments are the explanatory notes within the source code that are ignored by the compiler during compilation. Comments help improve code readability, provide context, and make it easier for others (or even yourself) to understand the purpose and functionality of different parts of the code.

Single-Line Comments:

Single-line comments are used to add comments on a single line of code. Anything following the double forward slash **//** is treated as a comment until the end of the line.

Single-line comments are useful for adding brief explanations or notes.

This is a single-line comment

int x = 10; // Initialize variable x with the value 10

Multi-Line Comments:

Multi-line comments (also known as block comments) are used to enclose longer comments that span multiple lines. These comments are enclosed between **/*** and ***/**.

Everything between these symbols is treated as a comment, and it can span multiple lines.

Example:

/*This is a multi-line comment.

It can span multiple lines and is useful

for more detailed explanations. */

Introduction to C:

The C programming language is a widely used and influential programming language that has significantly impacted the field of computer science and software development. Here's a brief overview of the evolution of the C language and its history:

1960s - BCPL and B Language: The precursor to C was the BCPL (Basic Combined Programming Language), which was developed by Martin Richards in the mid-1960s. Later, in the early 1970s, Ken Thompson at Bell Labs developed the B language, a simplified version of BCPL, which was used for early Unix operating system development.

Early 1970s - Development of C: Dennis Ritchie, also at Bell Labs, further refined the B language and created the C programming language between 1969 and 1973. C was designed with a focus on efficiency, portability, and low-level programming capabilities. It became the programming language for developing the Unix operating system.

1978 - "The C Programming Language": Dennis Ritchie and Brian Kernighan published "The C Programming Language," often referred to as "K&R C" after their initials. This book became the definitive reference for the language and played a crucial role in popularizing C.

1980s - ANSI C: As C gained popularity, efforts were made to standardize the language. In 1983, the American National Standards Institute (ANSI) established a committee to develop a standardized version of C. The result was ANSI C, also known as C89, which formalized many aspects of the language.

C++ and Object-Oriented Programming: In the 1980s, Bjarne Stroustrup extended C to create C++, which introduced object-oriented programming features. C++ retained C's capabilities while adding object-oriented concepts like classes and inheritance.

Modern Developments: While C++ gained popularity, the original C language continued to be widely used for system programming, embedded systems, and high-performance applications. Various programming languages and tools have been developed to build on C's foundation, such as C#, Java, and Rust.

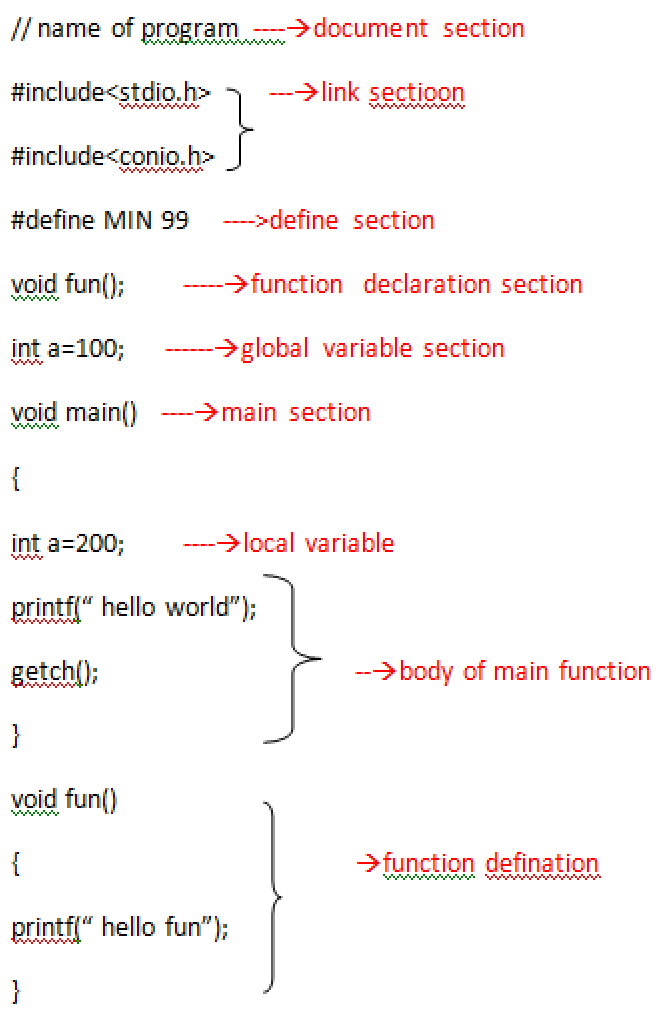
Structure of the C Program:

The basic structure of a C program is divided into 6 parts which makes it easy to read, modify, document, and understand in a particular format.

Sections of the C Program

There are 6 basic sections responsible for the proper execution of a program. Sections are mentioned below:

1. Documentation
2. Preprocessor Section
3. Definition
4. Global Declaration
5. Main() Function
6. Sub Programs



```
// name of program ---->document section
#include<stdio.h>  }  --->link sectioon
#include<conio.h>  }
#define MIN 99    ---->define section
void fun();      ----->function  declaration section
int a=100;      ------>global  variable section
void main()   ---->main section
{
int a=200;      ---->local variable
printf(" hello world");  }
getch();                 }  -->body of main function
}                        }
void fun()               }
{                        }  >function defination
printf(" hello fun");    }
}                        }
```

Preprocessor Directives: These are lines that begin with a '#' symbol and are processed by the preprocessor before the actual compilation. They include commands to include header files, define macros, and perform other preprocessing tasks.

Function Declarations: These are prototypes of functions used in the program. They provide information about the function's name, return type, and parameter list.

Global Variables: These are variables declared outside of any function and are accessible from anywhere in the program.

Main Function: Every C program must have a main() function. It serves as the entry point of the program and is where the execution begins.

Local Variables: These are variables declared within functions and have scope limited to that function. They are not accessible outside the function.

Statements and Expressions: The main body of the program consists of statements that perform actions and expressions that calculate values. Statements are terminated with a semicolon.

Function Definition: A function is a group of statements that together perform a task. Every C program has at least one function, which is main().

We can divide the entire code into separate function and each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Input and Output Statements:

In C programming, input and output functions are essential for interacting with users and manipulating data through the console or terminal. C provides a set of standard library functions for performing input and output operations. These functions are declared in the <stdio.h> header file.

Here are some of the commonly used input and output functions in C:

1. **Input Functions:**

   a.) scanf():
   This function is used to read formatted input from the user. It takes format specifiers as arguments to determine how to interpret the input data.

   For example:

```
int total;

printf("Enter the total maks: ");

scanf("%d", &total);
```

   b.) **getchar():**
   This function reads a single character from the standard input (usually the keyboard).

```
char ch;

printf("Enter a character: ");

ch = getchar();
```

2. **Output Functions**

a)printf():

This function is used for formatted output. It allows you to print text and formatted values to the console. The syntax is: Here are some commonly used input and output functions in C:

```
int num = 100;

printf("The answer is: %d\n", num);
```

b) putchar():

This function writes a single character to the standard output (usually the screen).

```
char ch = 'A';

putchar(ch);
```

c) **puts():**

This function is used to write a string (sequence of characters) followed by a newline character to the standard output.

```
 char message[] = "Hello, World!";

 puts(message);
```

Here's a simple example that demonstrates using input and output functions:

```
#include <stdio.h>

int main()

{

int num;

printf("Enter a number: ");

scanf("%d", &num);

printf("You entered: %d\n", num);

return 0;

}
```

In this example, the program prompts the user to enter a number, reads the input using **scanf()**, and then prints the entered number using **printf()**.

VARIABLES AND IDENTIFIERS:

In C programming, variables and identifiers are fundamental concepts that play a crucial role in storing and manipulating data within a program.

Variables:

A variable is a name that points to a memory location. It is the basic unit of storage in a program. The value of a variable can be changed during program execution. All the operations are done on the variable effects that memory location. In C, all the variables must be declared before use and in C we can declare anywhere in the program at our convenience.

The way to declare any variable:

data_type variable_name;

For example:

int number; // Declares an integer variable named "number"

char ch; // Declares a character variable named "ch"

float price; // Declares a floating-point variable named "price"

Identifiers:

An identifier is a name given to a variable, function, or any other user-defined entity in your program. Identifiers are used to uniquely identify these entities in your code. They have certain rules and conventions to follow:

- An identifier can consist of letters (both uppercase and lowercase), digits, and underscores (**_**).
- The first character of an identifier must be a letter or an underscore.
- Identifiers are case-sensitive, meaning **variableName** and **variablename** are treated as different identifiers.
- Keywords (reserved words) used in the C language cannot be used as identifiers (e.g., **int**, **float**, **for**, etc.)

  Examples:

```
int studentAge;      // A variable representing the age of a student
float averageScore;  // A variable representng the average score
void calculateSum(); // A function that calculates the sum
```

C provides certain naming conventions that are commonly followed:

- Use lowercase letters for variable and function names (e.g., calculate, findaverage).
- Use uppercase letters for constant identifiers (e.g., PI, MAX_VALUE).
- Use mixed case for improved readability (e.g., averageScore, calculateSum).

Difference Between Identifiers and Variables:

S.NO	VARIABLES	IDENTIFIERS
1	A Variable is a name that is assigned to a memory location, which is used to contain the corresponding value in it.	It is a unique name which is given to an entity to distinctly identify it
2	A variable is a specific type of identifier that refers to a named memory location used to store data. Variables have a data type that determines the type of data they can hold (e.g., integers, floating-point numbers, characters).	They are general names used to uniquely identify various entities in your program,
3	e.g., int, for, while	e.g, int age; float temperature;

EXPRESSIONS:

Expressions are the combination of variables, operands, and operators. The result would be stored in the variable once the expressions are processed based on the operator's precedence.

There are three kinds of expressions:

➔ An arithmetic expression evaluates to a single arithmetic value.
➔ A character expression evaluates to a single value of type character.
➔ A logical or relational expression evaluates to a single logical value.

Here are some key elements of expressions:

1. **Operators:**

Operators are symbols that perform operations on operands. They can be used to perform arithmetic, logical, bitwise, and other operations. For example, the + and - operators perform addition and subtraction, respectively.

2. **Operands:**

Operands are values or variables that an operator acts upon. For instance, in the expression **a + 2**, both **a** and **2** are operands.

3. **Unary, Binary, and Ternary Operators:**

    - Unary operators work with a single operand, like the **-** operator in **-a**.
    - Binary operators work with two operands, like the **+** operator in **a + 2**.
    - Ternary operators, specifically the conditional operator (**? :**), work with three operands and can be used for conditional expressions.

4. **Precedence and Associativity:**

Operators have precedence and associativity rules that determine the order in which they are evaluated. For example, multiplication is typically performed before addition due to higher precedence.

5. **Function Calls:**

Functions can also be part of expressions. A function call is an expression that invokes a specific function and can yield a value. For example, **sqrt(25)** is an expression that calls the square root function on the operand **25**.

6. **Literals:**

Numeric or character constants are referred to as literals and can be used directly in expressions. For example, **100** and **'A'** are literals.

Here are a few examples of expressions:

1. Arithmetic Expression:

```
int x = 10, y = 5;

int result = x + y * 2; // The expression calculates the result as 20
```

2. Logical Expression:

```
int a = 5, b = 10;

int isTrue = (a > 3) && (b < 15); // The expression evaluates to 1 (true)
```

3. Function Call Expression:

double radius = 3.0;

double area = 3.14159 * pow(radius, 2); // The expression calculates the area of a circle

4. Conditional Expression:

int value = 8;

char result = (value % 2 == 0) ? 'E' : 'O'; // The expression evaluates to 'E'

Expressions are used extensively in control structures (if-else statements, loops, etc.), assignments, and more complex computations within C programs.

CONSTANTS, KEYWORDS:

1. Constants:

Constants are fixed values that do not change during the execution of a program. They are used to represent fixed numerical values, characters, or strings. C supports several types of constants:

- Numeric Constants: These are used to represent numeric values like integers and floating-point numbers. For example: 42, 3.14.
- Character Constants: These represent individual characters. They are enclosed in single quotes. For example: 'A', '5'.
- String Constants: These represent sequences of characters (strings). They are enclosed in double quotes. For example: "Hello, World!".
- Symbolic Constants: Also known as "macros" or "named constants," these are identifiers that are replaced with their associated values during compilation using the #define preprocessor directive. For example: #define PI 3.14159.

2. Keywords:

Keywords are reserved words that have predefined meanings in the C programming language. They are used to define the structure, control flow, and behavior of programs. Since keywords have specific meanings, they cannot be used as identifiers (variable names, function names, etc.).

Examples of C keywords include:

- int: Defines an integer data type.
- float: Defines a floating-point data type.
- if: Introduces a conditional statement.

- for: Introduces a loop construct.
- while: Introduces a loop construct.
- return: Exits a function and returns a value.

VALUES, NAMES, SCOPE, BINDING, STORAGE CLASSES:

1. Values:

Values are the actual data that variables can hold. They can be numbers, characters, strings, or any other data type supported by the programming language.

For example,

in C, 10, 'x', and "Good Morning" are values of different types.

2. Names (Identifiers):

Names, also known as identifiers, are used to uniquely identify variables, functions, and other user-defined entities within a program.

They are used to reference and manipulate values.

In C, identifiers must follow specific rules and conventions, such as starting with a letter or underscore, being case-sensitive, and avoiding reserved keywords.

3. Scope:

The scope of a variable refers to the region of the program where the variable is accessible. Variables can have different scopes depending on where they are defined.

In C, variables can have local scope (limited to a specific block or function) or global scope (accessible throughout the entire program).

Scope determines where a variable can be used and accessed.

4. Binding:

Binding is the association between a name (identifier) and the memory location or value it represents.

It's the process of connecting a variable name with the data it references.

This can be done during compile-time (static binding) or runtime (dynamic binding).

5.Storage Classes:

Storage classes in C determine how and where variables are stored in memory, as well as their lifetimes.

C provides several storage classes:

- auto: The default storage class for local variables. The variable's memory is allocated when the block containing its declaration is entered, and it is deallocated when the block is exited.
- register: Suggests that a variable be stored in a CPU register for faster access.
- static: Extends the lifetime of a variable to the entire program execution. A static local variable retains its value between function calls.
- extern: Indicates that a variable is defined externally, often used for variables declared in other files.
- typedef: Creates an alias for a data type, improving code readability.

The choice of storage class affects variables' lifetime, scope, and memory usage.

Numeric Data Types:

Integer:

In C programming, the int data type is used to represent integers, which are whole numbers without any fractional or decimal parts. The int data type typically has a fixed size on a particular system, which can vary depending on the architecture and compiler being used.

Size and Range: The size of an int data type depends on the system architecture. On most modern systems, an int is usually represented using 2 bytes (16bits) or 4 bytes (32 bits). The range of values an int can represent depends on its size.

Declaration: To declare a variable of type **int**, you can use the following syntax:

Eg. Int age;

Initialization: You can also initialize an **int** variable during declaration:

Eg. Int age =20;

Float:

In C programming, the float data type is used to represent floating-point numbers, which include both integer and fractional parts.

Size and Precision: The float data type typically uses 4 bytes (32 bits) of memory to store a floating-point number. It allows us to represent a wide range of values, but the precision (number of significant digits) is limited.

Declaration: To declare a variable of type **float**, you can use the following syntax:

Eg. float weight;

Initialization: You can also initialize an **float** variable during declaration:

Eg. Float weight =56.5;

Non-Numeric Data Types:

Character :

The character data type is used to hold character data and is divided into two types one is signed data type and the second one is unsigned data type. Both Signed data type and unsigned data type occupy only **one byte** of memory.

Declaration:

Char c;

Initialization of character data:

Char c='h';

String:

In C programming, there is no built-in "string" data type like in some higher-level languages such as C++, Python, or Java. Instead, C uses character arrays to represent strings. A string in C is essentially an array of characters terminated by a null character ('\0'), which marks the end of the string.

Declaration and Initialization:

char myString[] = "Hello, World!";

This creates an array named myString that holds the characters of the string "Hello, World!" including the null terminator.

String Length: The length of a string is the number of characters in the string excluding the null terminator. You can find the length of a string using the strlen function from the <string.h> header:

```
#include <string.h>

char myString[] = "Hello, World!";

int length = strlen(myString);
```

Accessing Characters: You can access individual characters of a string using array indexing

```
char myString[] = "Hello";

char firstChar = myString[0]; // 'H'
```

Modifying Strings: You can modify individual characters of a string, but Changing the length of a string often requires careful management of memory and reallocation.

Input and Output: To print a string, you can use the %s format specifier with printf:

```
char myString[] = "Hello, World!";

printf("%s\n", myString);
```

To read a string from the user, you can use the scanf function with the %s format specifier, but note that scanf might not be suitable for reading strings with spaces due to its whitespace-based nature:

```
char userInput[100];

printf("Enter a string: ");

scanf("%s", userInput); // This may have issues with spaces
```

The following are the various string functions that we can perform on strings:

C has many useful string functions, which can be used to perform certain operations on strings.

To use them, we must include the <string.h> header file in our program:

String Length:

Strlen(): This function calculates the length of a string (excluding the null terminator).

```
#include <string.h>

char myString[] = "Hello, World!";

int length = strlen(myString); // Returns 13
```

Concatenate Strings:

To concatenate (combine) two strings, you can use the strcat() function:

```
char str1[20] = "Hello ";
char str2[] = "World!";

// Concatenate str2 to str1 (result is stored in str1)
strcat(str1, str2);

// Print str1
printf("%s", str1);
```

The size of str1 should be large enough to store the result of the two strings combined.

Copy Strings:

strcpy(): It is used to copy the character array pointed by the source to the location pointed by the destination. Or it copies the source string(character array) to the destination string(character array).

```
#include <string.h>

char source[] = "Copy this";

char destination[20];

strcpy(destination, source); // Copies "Copy this"
```

Compare Strings:

To compare two strings, **strmp()** function is used.

It returns 0 if two strings are equal, otherwise a value that is not 0:

This function compares two strings and returns an integer value:

- Negative if the first string is lexicographically less than the second.
- Zero if the strings are equal.
- Positive if the first string is lexicographically greater than the second.

```
#include <string.h>

char str1[] = "apple";
```

char str2[] = "banana";

int result = strcmp(str1, str2); // Returns a negative value

sprint():

This function works similarly to **printf**, but instead of printing to the console, it formats and stores the output in a string buffer.

#include <stdio.h>

char buffer[50];

int num = 42;

sprintf(buffer, "The number is %d.", num); // Stores "The number is 42."

gets() and puts() functions:

gets() : Reads characters from the standard input and stores them as a string.

puts() : It is used to print the string on the console which is previously read by using gets() or scanf() function.

Example:

```
  #include<stdio.h>
  #include <string.h>
  int main(){
  char name[50];
      printf("Enter your name: ");
      gets(name); //reads string from user
      printf("Your name is: ");
      puts(name);  //displays string
   return 0;
  }
```

Strlwr() and Strupr() functions in C:

Strlwr(): It converts the given string into lowercase

Strupr(): It converts the given string into uppercase

Example :

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

```
int main(){
char str[20];
printf("Enter string: ");
gets(str);//reads string from console.
printf("String is: %s",str);
printf("\nLower String is: %s",strlwr(str));
printf("\nUpper String is: %s",strupr(str));
```

L Value and R Value in Expression:

L Value:

"lvalue" and "rvalue" are terms used to describe the two fundamental categories that expressions and objects can fall into. These terms are closely related to the concept of assignment and how values are manipulated within the language.

Lvalue (Left Value): An lvalue refers to an expression or an object that can appear on the left-hand side of an assignment operation. In other words, an lvalue represents a memory location or an object with an identifiable address.

Example:

```
int x = 10;    // 'x' is an lvalue
int arr[5];    // 'arr' is an lvalue (array)
```

R Value:

An rvalue refers to an expression or a value that can appear on the right-hand side of an assignment operation. An rvalue represents the actual value itself, rather than a memory location. Rvalues are typically used in calculations or as operands for various operations. You cannot assign a new value to an rvalue directly.

Example:

```
int y = 5 + 3;   // The expression '5 + 3' is an rvalue

int z = x + y;   // The expressions 'x' and 'y' are both rvalues
```

Increment and Decrement Operator:

The increment (++) and decrement (--) operators are used to modify the value of a variable by either increasing or decreasing it by 1, respectively. These operators are commonly used for looping, counting, and other situations where you need to manipulate numerical values. The increment and decrement operators can be applied to both integer and floating-point types.

1. **Increment Operator (++):** The increment operator (++) increases the value of a variable by 1. It can be used in two forms: as a prefix or a postfix operator.

- **Prefix Increment (++variable):** In this form, the variable is incremented before its value is used in the expression.

int x = 5; int y = ++x; // 'y' will be 6, and 'x' will be 6

- **Postfix Increment (variable++):** In this form, the variable's current value is used in the expression, and then the variable is incremented afterward.

int a = 5;

int b = a++; // 'b' will be 5, and 'a' will be 6

2. **Decrement Operator (--):** The decrement operator (**--**) decreases the value of a variable by 1. Similar to the increment operator, it can be used as a prefix or a postfix operator.

- **Prefix Decrement (--variable):** The variable is decremented before its value is used in the expression.

int p = 8;

int q = --p; // 'q' will be 7, and 'p' will be 7

- **Postfix Decrement (variable--):** The variable's current value is used in the expression, and then the variable is decremented afterward.

int m = 8;

int n = m--; // 'n' will be 8, and 'm' will be 7

Comma, Arrow and Assignment operator:

Comma Operator (,): The comma operator in C is used to separate expressions within a larger expression or statement. It evaluates the expressions from left to right and returns the value of the rightmost expression. It's often used in situations where multiple expressions need to be executed sequentially, such as in a for loop initialization or in function arguments.

int x = 5, y = 10; int z = (x++, y++); // The comma operator is used to evaluate 'x++' and 'y++', and the value of 'y++' is assigned to 'z'

Here, the value assigned to z will be 11, as x and y are both incremented.

Arrow Operator (->): The arrow operator (->) is used to access a member of a structure or union through a pointer to that structure or union. It combines pointer dereferencing and member access into a single operation.

```
struct Student
{
int rollNumber;
char name[50]; };
struct Student student1;
struct Student *ptr = &student1;
ptr->rollNumber = 101; // Using the arrow operator to access the 'rollNumber' member through the pointer
}
```

In this example, ptr->rollNumber is equivalent to (*ptr).rollNumber. The arrow operator is more concise and easier to read when dealing with pointers to structures or unions.

Assignment Operator (=): The assignment operator (=) is used to assign a value to a variable. It assigns the value on the right-hand side to the variable on the left-hand side. The assignment operator is fundamental for storing values in variables and updating their contents.

```
int a = 5; // The assignment operator assigns the value 5 to the variable 'a'
int b = a + 3; // The assignment operator assigns the value of the expression 'a + 3' to the variable 'b'
```

The assignment operator can also be combined with other operators to create compound assignment operators, such as +=, -=, *=, /=, and so on.

Bitwise and Sizeof() Operator :

Bitwise Operators: Bitwise operators in C are used to manipulate individual bits of integral data types, such as integers and characters. They operate on the binary representation of the values and are particularly useful for low-level programming, dealing with hardware registers, and working with data compression or encryption algorithms.

Bitwise AND (&): Performs a bitwise AND operation between the corresponding bits of two operands.

- Bitwise OR (|): Performs a bitwise OR operation between the corresponding bits of two operands.
- Bitwise XOR (^): Performs a bitwise exclusive OR (XOR) operation between the corresponding bits of two operands.
- Bitwise NOT (~): Flips the individual bits of its operand, turning each 0 into a 1 and each 1 into a 0.
- Left Shift (<<): Shifts the bits of the left operand to the left by a specified number of positions.
- Right Shift (>>): Shifts the bits of the left operand to the right by a specified number of positions.

```
unsigned int a = 5; // Binary: 0101

unsigned int b = 3; // Binary: 0011

unsigned int result_and = a & b; // Binary AND: 0001 (Decimal: 1)

unsigned int result_or = a | b; // Binary OR: 0111 (Decimal: 7)

unsigned int result_xor = a ^ b; // Binary XOR: 0110 (Decimal: 6)

unsigned int result_shift = a << 2; // Left Shift: 10100 (Decimal: 20)
```

sizeof() Operator: The sizeof() operator in C is used to determine the size, in bytes, of a data type or a variable. It's a compile-time operator that helps you write portable and platform-independent code by providing information about the memory requirements of your program's data structures.

```
int integerTypeSize = sizeof(int);

double doubleTypeSize = sizeof(double);

char charTypeSize = sizeof(char);

struct Student
```

```
{

int rollNumber;

char name[50];

 };

size_t structSize = sizeof(struct Student);
```

The sizeof() operator returns a value of type size_t, which is an unsigned integer type specifically designed to represent sizes in bytes.

Arithmetic, Relational and Logical Operators:

Arithmetic Operators: Arithmetic operators are used to perform mathematical operations on numeric data types (integers and floating-point numbers). These operators allow you to perform basic arithmetic calculations:

- **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 (%):** Computes the remainder when the left operand is divided by the right operand.

```
int x = 10, y = 3;

int sum = x + y; // 10 + 3 = 13

int difference = x - y; // 10 - 3 = 7

int product = x * y; // 10 * 3 = 30

int quotient = x / y; // 10 / 3 = 3 (integer division)

int remainder = x % y; // 10 % 3 = 1 (remainder)
```

Relational Operators: Relational operators are used to compare values and determine the relationship between them. These operators return a Boolean value (**true** or **false**) indicating whether the condition is met or not:

- **Equal to (==):** Checks if two operands are equal.
- **Not equal to (!=):** Checks if two operands are not equal.
- **Greater than (>):** Checks if the left operand is greater than the right operand.
- **Less than (<):** Checks if the left operand is less than the right operand.
- **Greater than or equal to (>=):** Checks if the left operand is greater than or equal to the right operand.
- **Less than or equal to (<=):** Checks if the left operand is less than or equal to the right operand.

```
int a = 5, b = 8;

bool isEqual = (a == b); // false

bool isNotEqual = (a != b); // true

bool isGreater = (a > b); // false

bool isLess = (a < b); // true

bool isGreaterOrEqual = (a >= b); // false

bool isLessOrEqual = (a <= b); // true
```

Logical Operators: Logical operators are used to combine and manipulate Boolean values (**true** or **false**). They allow us to create complex conditions by combining simpler conditions:

- **Logical AND (&&):** Returns **true** if both operands are **true**.
- **Logical OR (||):** Returns **true** if at least one operand is **true**.
- **Logical NOT (!):** Returns the opposite Boolean value of the operand.

```
bool p = true,

q = false;

bool logicalAndResult = p && q; // false

bool logicalOrResult = p || q; // true

bool logicalNotResult = !p; // false
```

Condition Operators:

It is also known as the "ternary operator". The conditional operator (? :) is used to express simple conditional statements, often used for assigning values based on a condition. It's a shorthand form of an if-else statement and has the following syntax:

Syntax:

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

The condition is evaluated first. If the condition is true, the expression_if_true is evaluated and becomes the result of the entire expression. If the condition is false, the expression_if_false is evaluated and becomes the result.

The result of the conditional operator can be used in assignments, calculations, function calls, and more.

Example:

```
#include <stdio.h>

int main()

{

int x = 10;

int y = 20;

int max = (x > y) ? x : y; // If x > y, max = x; otherwise, max = y

printf("The maximum value is: %d\n", max);
return 0;

}
```

In this example, the condition x > y is evaluated. If it's true, the value of x becomes the value of max; otherwise, the value of y becomes the value of max.

Operator Precedence:

Operator precedence determines the order in which different operators are evaluated within an expression. When an expression contains multiple operators, operator precedence dictates which operators are evaluated first and which ones are evaluated later. This is important to ensure that expressions are evaluated correctly and consistently.

C follows a specific set of rules for operator precedence, which determines how expressions are evaluated. Here's a general overview of some key operator precedence rules in C:

1. **Higher Precedence Operators are Evaluated First:** Operators with higher precedence are evaluated before operators with lower precedence. For example, multiplication (*) has higher precedence than addition (+), so 2 + 3 * 4 is evaluated as 2 + (3 * 4).
2. **Parentheses Override Precedence:** Expressions within parentheses are evaluated first, regardless of operator precedence. Parentheses can be used to explicitly specify the order of evaluation.
3. **Associativity:** If multiple operators of the same precedence level are used in an expression, their associativity determines the order of evaluation. Most binary operators in C are left-associative, meaning they are evaluated from left to right. The assignment operator (=) is right-associative, meaning it's evaluated from right to left.

Here's a simplified breakdown of operator precedence for some common operators in C:

1. Highest Precedence:
    - Parentheses ( )

   - Function call ()
   - Array subscripting []
   - Structure/Union member access ., ->
   - Postfix increment and decrement ++, --

2. Medium Precedence:
   - Unary operators +, -, !, ~, *, &
   - Type cast (type)
3. Lower Precedence:
   - Multiplication *, Division /, Modulus %
4. Lower Precedence:
   - Addition +, Subtraction -
5. Lower Precedence:
   - Relational operators <, <=, >, >=
6. Lower Precedence:
   - Equality operators ==, !=
7. Lower Precedence:
   - Logical AND &&
8. Lower Precedence:
   - Logical OR ||
9. Lowest Precedence:
   - Assignment operators =, +=, -= and so on

Operator precedence can become more complex when expressions contain mixed operators. To avoid confusion, it's a good practice to use parentheses to explicitly indicate the desired order of evaluation, especially when dealing with complex expressions.

Here's an example that demonstrates the importance of operator precedence:

```
#include <stdio.h>

int main()

{

int result = 2 + 3 * 4; // This is evaluated as 2 + (3 * 4)

printf("Result: %d\n", result); // Output: Result: 14

return 0;

}
```

In this example, without parentheses, the multiplication is performed before the addition due to the higher precedence of the multiplication operator.

Expressions with pre/post increment operator:

Expressions involving pre-increment and post-increment operators (++) can sometimes lead to confusion if not understood correctly. These operators are used to increase the value of a variable by 1, but they have different effects depending on whether they are used before or after the variable in the expression.

Pre-Increment Operator (++variable): The pre-increment operator increments the value of the variable before its value is used in the expression. After incrementing, the updated value is then used in calculations.

```
 int x = 5;

 int y = ++x;  // 'x' is incremented to 6, and 'y' is assigned the value 6
```

Post-Increment Operator (variable++): The post-increment operator evaluates the expression using the current value of the variable and then increments the variable afterward. In other words, the variable's value is used first, and then it is incremented.

```
int a = 5;
int b = a++; // 'b' is assigned the value 5 (current value of 'a'), then 'a' is incremented   by 1
```