What is an Algorithm ? Explain with an example

An algorithm is a step-by-step procedure to solve a given problem. It is a finite set of unambiguous instructions that, when executed, perform a task correctly. There are 3 important features,

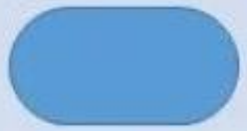
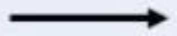
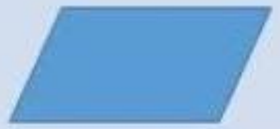
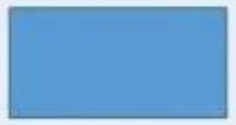
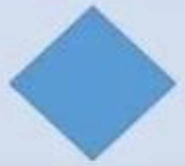
- The number of steps required to perform the task should be finite
- Each of the instructions in the algorithm should be unambiguous in nature
- Finally, the algorithm should solve the problem correctly.

Write an Algorithm to find Largest of 3 numbers.

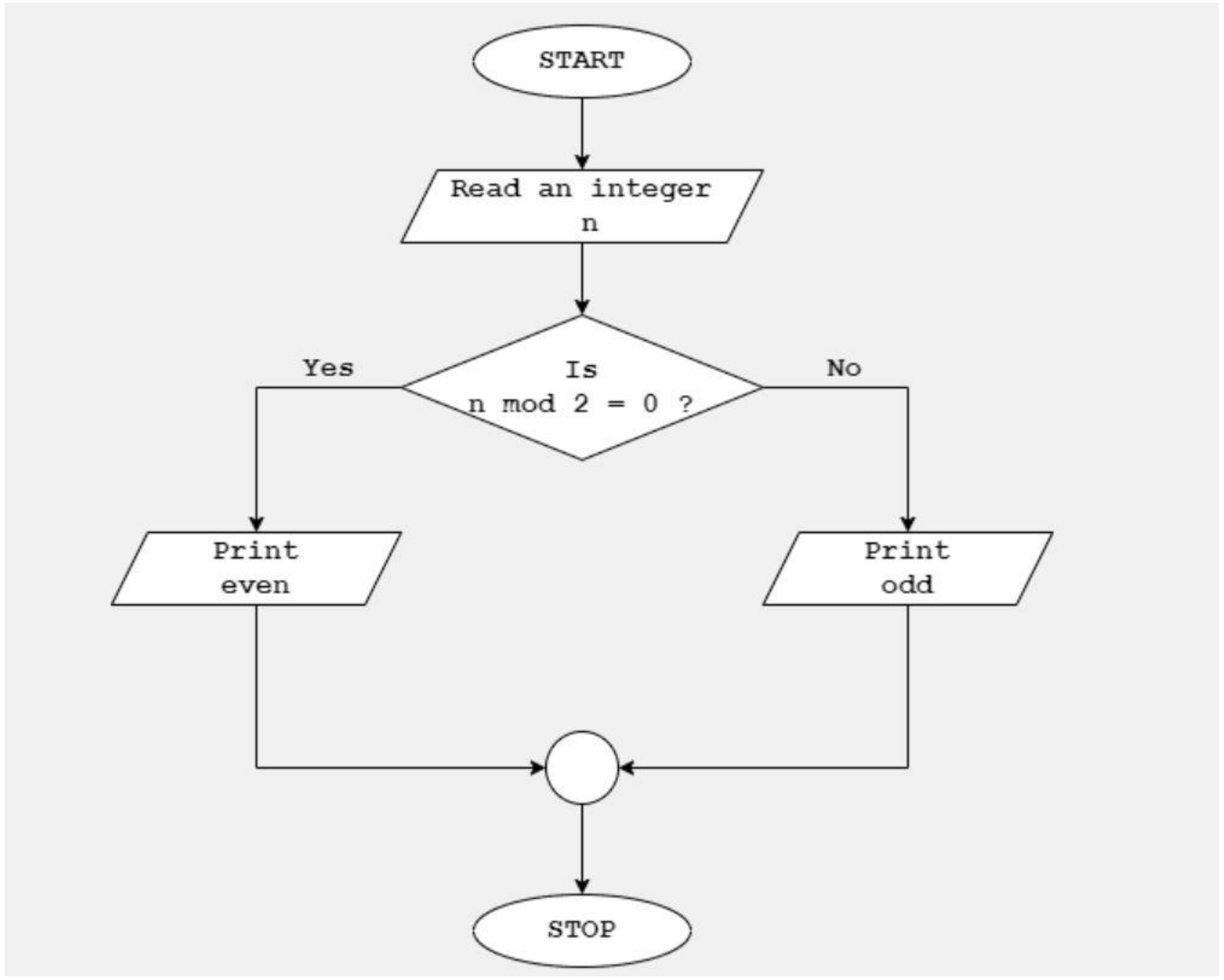
```
Step 1: Start
Step 2: Input three numbers → A, B, C
Step 3: If A > B and A > C, then
        Largest = A
Step 4: Else if B > A and B > C, then
        Largest = B
Step 5: Else
        Largest =  C
Step 6: Display Largest number
Step 7: Stop
```

What is a Flowchart ? Explain with examples.

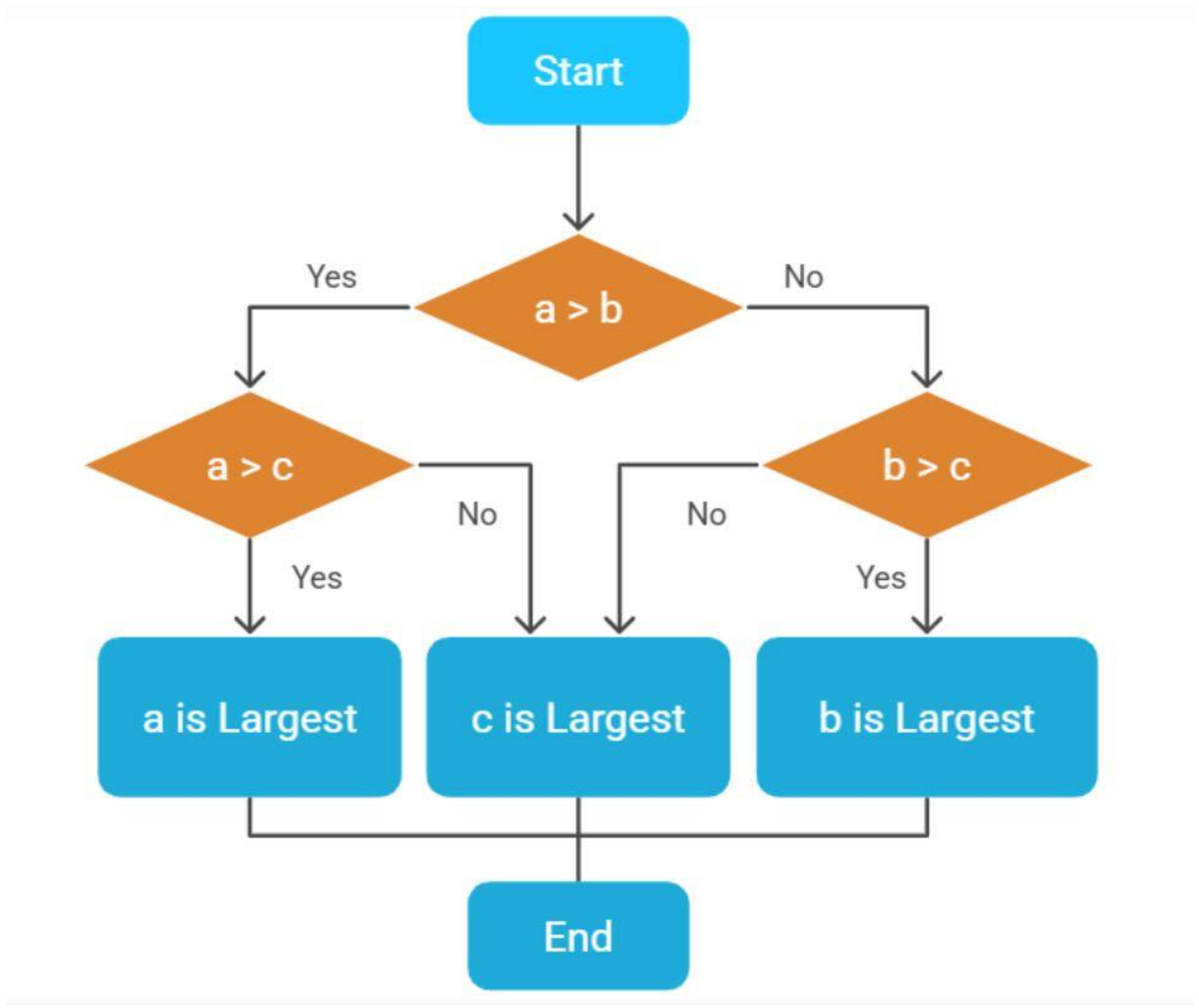
- Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem.
- It makes use of symbols which are connected among them to indicate the flow of information and processing.

Symbol	Function	Description
	Start / End	An oval represents a start or end of code
	Flow Arrow	Arrows are used to represent the code flow between the flow shapes
	Input / Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Draw the flowchart to check if a number is Even or Odd.



Flowchart to find Largest of 3 numbers



What is an identifier?

Identifiers refer to the names of variables, functions and arrays. These a user defined names and consists of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers and is usually used as a link between two words in a long identifier.

What are the rules to construct an identifier?

- ➢First character must be an alphabet (or underscore)
- ➢Must consist of only letters, digits or underscore
- ➢Cannot use a keyword
- ➢Must not contain a white space

Classify the following as valid/invalid Identifiers.

- i) num2
- ii) $num1
- iii) +add
- iv) a_2
- v) 199_space
- vi) _apple
- vii)#12

i)num2 → Valid (starts with letter, contains letters & digits)

ii) $num1 → Invalid (special symbol $ not allowed)

iii) +add → Invalid (cannot start with +)

iv) a_2 → Valid (letter + underscore + digit allowed)

v) 199_space → Invalid (cannot start with a digit)

vi) _apple → Valid (underscore at beginning allowed)

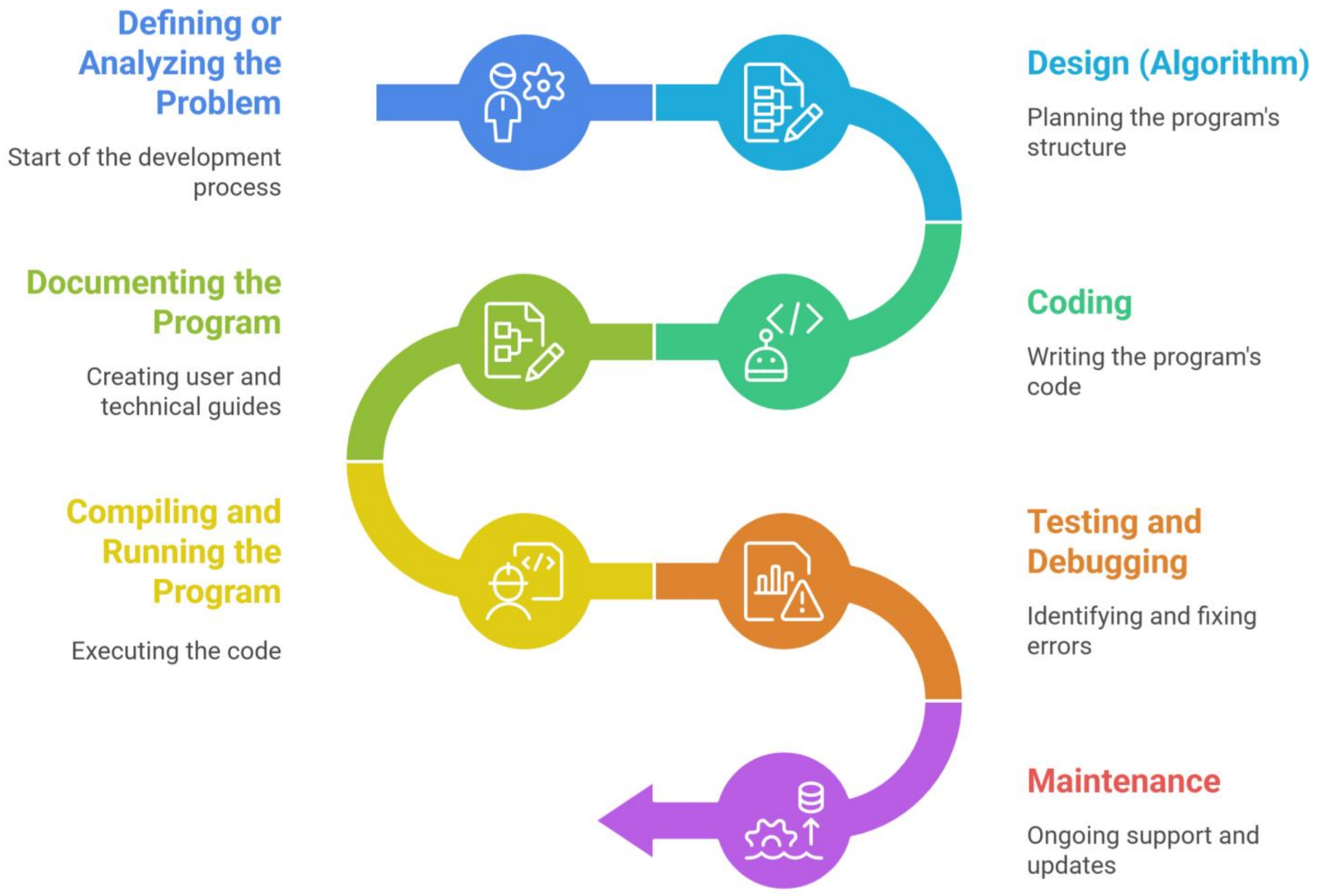
vii) #12 → Invalid (cannot start with #)

Explain the need for the following:
#include<stdio.h>
#include<math.h>

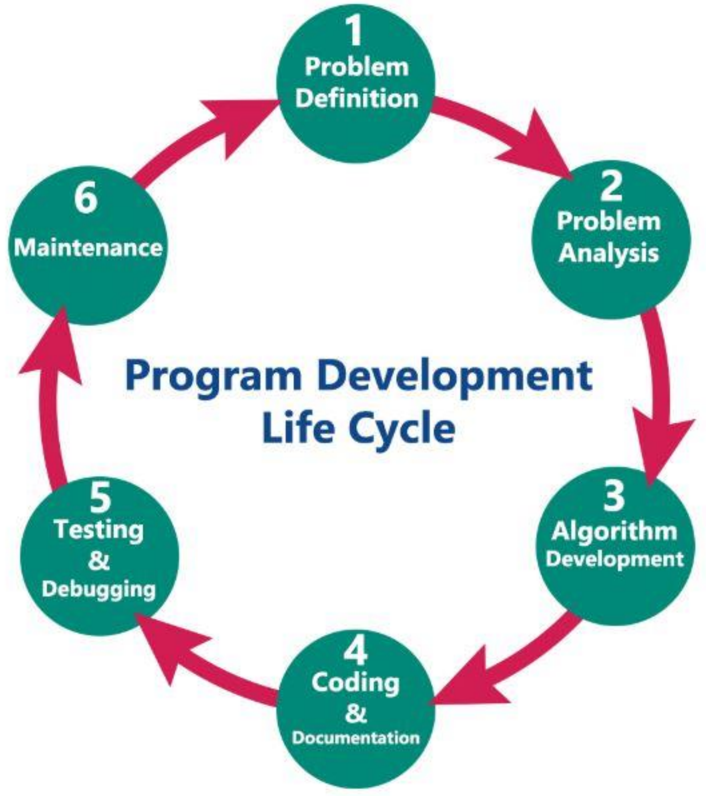
#include <stdio.h> is a Standard Input/Output header file containing standard input and output functions. Examples: **printf()** to display output and **scanf()** to take input from user. Without including < stdio.h>, functions like printf() and scanf() will not work, and the compiler will give errors.

#include<math.h> is a Math library header file in which standard mathematical functions are predefined. Examples: sqrt(x)for square root, pow(x, y) for x raised to the power y, sin(x), cos(x), tan(x) for trigonometric functions. Without including <math.h>, the above mathematical operations cannot be used.

What are Program Development Steps ? Explain with a neat diagram.


Defining or Analyzing the Problem
Start of the development process
Design (Algorithm)
Planning the program's structure
Documenting the Program
Creating user and technical guides
Coding
Writing the program's code
Compiling and Running the Program
Executing the code
Testing and Debugging
Identifying and fixing errors
Maintenance
Ongoing support and updates

OR



Program Development Steps

Defining or Analyzing the Problem:

- This is the most crucial step. It involves understanding what the problem is, identifying inputs, desired outputs, and the processing required. The goal is to gather complete and clear requirements so the developer knows exactly what needs to be done.

Design (Algorithm):

- Once the problem is understood, the next step is to design a logical plan to solve it.
- This includes:
  - ➢ Writing an algorithm (step-by-step procedure)
  - ➢ Creating flowcharts or pseudocode to visualize the logic.

Program Development Steps

Coding:

- In this phase, the designed algorithm is translated into a programming language (like C, Python, or Java). The developer writes the actual code using the selected language's syntax and rules.

Documenting the Program:

- Good programming practice involves writing clear documentation. This means:
  - ➢ Adding comments in the code
  - ➢ Writing user manuals or guides
  - ➢ Explaining the logic and structure

Program Development Steps

Compiling and Running the Program:

- After coding, the program is compiled to check for syntax errors.
- If there are no errors, the program is executed to see if it performs the desired task. This phase helps convert the human-written code into machine-understandable form.

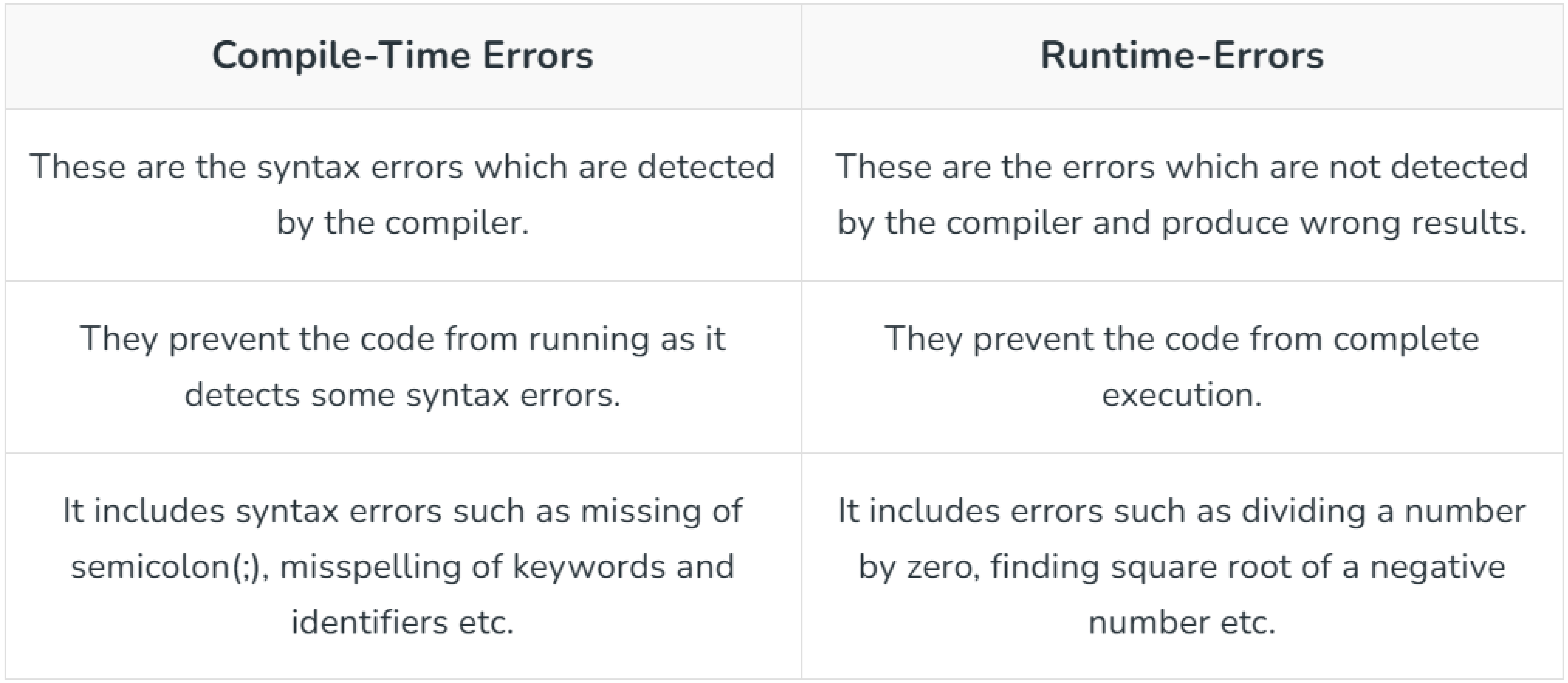
Testing and Debugging:

- The program is now tested with different sets of data to ensure it works correctly in all scenarios. If problems are found, they are identified and fixed to improve program correctness and stability.

Maintenance:

- After deployment, the program may need updates, error fixes, or feature enhancements.
- Maintenance ensures the software continues to meet user needs over time, even as requirements change or new issues arise.

Differentiate between syntax errors and run-time errors.



Compile-Time Errors	Runtime-Errors
These are the syntax errors which are detected by the compiler.	These are the errors which are not detected by the compiler and produce wrong results.
They prevent the code from running as it detects some syntax errors.	They prevent the code from complete execution.
It includes syntax errors such as missing of semicolon(;), misspelling of keywords and identifiers etc.	It includes errors such as dividing a number by zero, finding square root of a negative number etc.

X += 10 != 15 && !(10<20) || 15 > 30 where X = 9

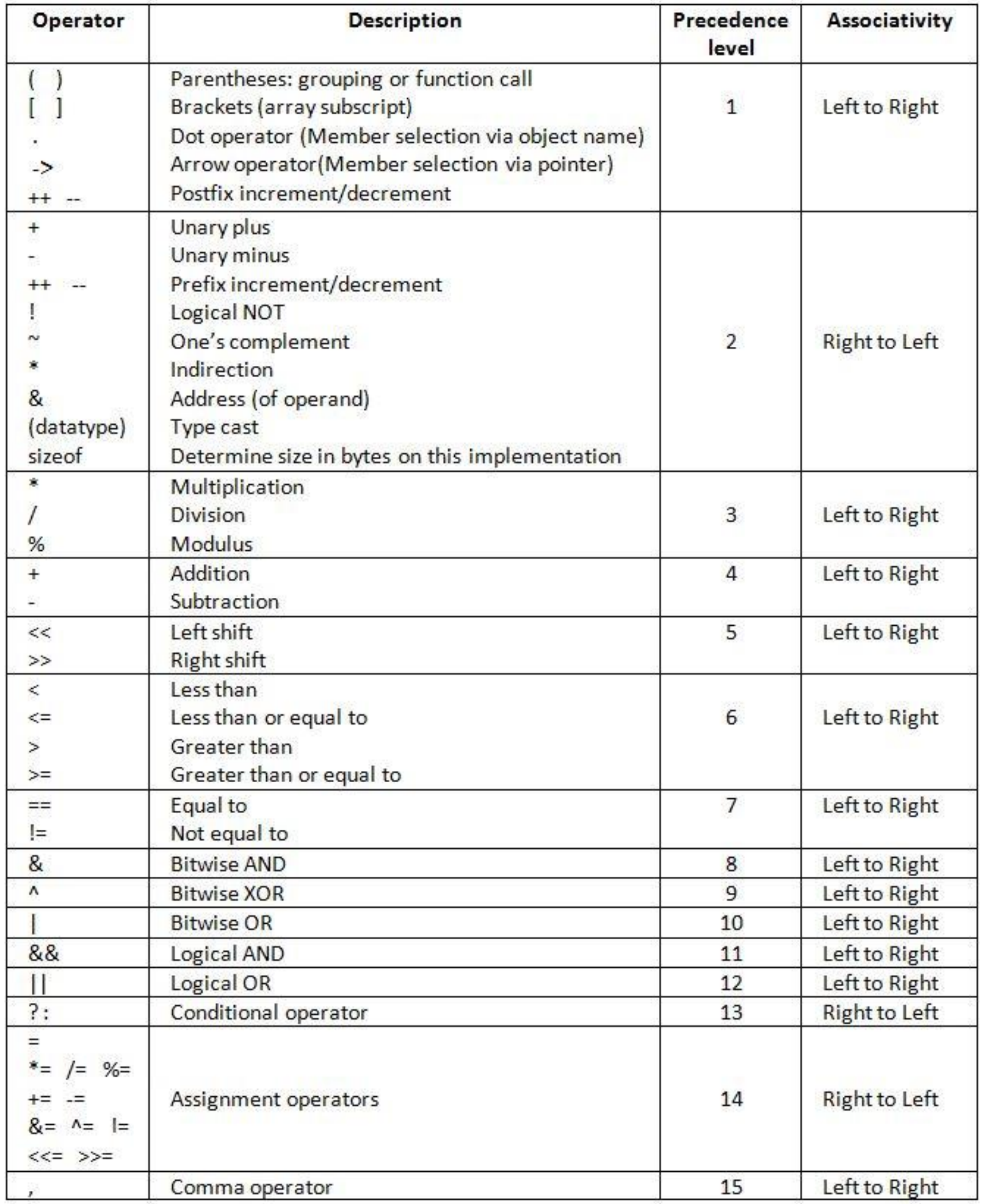
step 1: X += (10 != 15) && !(10 < 20) || (15 > 30)

10<20→ true (1) → so, (!(10 < 20)) = !(1) = 0
15>30→ false (0)
10!=15 → true (1)

step 2: X += (1 && 0) || 0

1 && 0 = 0
0||0=0

step 3: X=9+0=9



Operator	Description	Precedence level	Associativity
( )	Parentheses: grouping or function call	1	Left to Right
[ ]	Brackets (array subscript)		
.	Dot operator (Member selection via object name)		
->	Arrow operator(Member selection via pointer)		
++ --	Postfix increment/decrement		
+	Unary plus	2	Right to Left
-	Unary minus		
++ --	Prefix increment/decrement		
!	Logical NOT		
~	One's complement		
*	Indirection		
&	Address (of operand)		
(datatype)	Type cast		
sizeof	Determine size in bytes on this implementation		
*	Multiplication	3	Left to Right
/	Division		
%	Modulus		
+	Addition	4	Left to Right
-	Subtraction		
<<	Left shift	5	Left to Right
>>	Right shift		
<	Less than	6	Left to Right
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	7	Left to Right
!=	Not equal to		
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
|	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
?:	Conditional operator	13	Right to Left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

a + b / c > 2 * c || a – b < c && c << 2 where a = 5, b = 4 and c = 6.

Step 1: Division first:

`b / c = 4 / 6 = 0` (integer division).

So, a + 0 > 2 * c || a - b < c && c << 2

Step 2: Multiplication:

`2 * c = 2 * 6 = 12.`

So, a + 0 > 12 || a - b < c && c << 2

Step 3: Addition/Subtraction:

`a + 0 = 5`

`a - b = 5 - 4 = 1`

So, 5 > 12 || 1 < 6 && c << 2

Step 5: Shift operator:

`c << 2 = 6 << 2 = 24` (left shift = multiply by 2^2).

So, 5>12 || 1<6 && 24

6 (decimal) = 0000 0110=> 0001 1000 (0+0+0+2^4+2^3+0+0+0)=24

Step 4: Relational operators:

`5 > 12 → 0 (false)`

`1 < 6 → 1 (true)`

So, 0 || 1 && 24

Step 6: Logical OR , logical AND:

0 || 1= 1

`1 && 24 → true (1)` because both nonzero

Operator Precedence in C

Highest to lowest :

1. / *→ division/multiplication
2. + , – → addition / subtraction
3. << → left shift
4. >, < → relational operators
5. ||, &&→ logical OR and logical AND

Evaluate the following expression
i. X = a – b / 3 – c * 2 – 1 when a=9,b=12, c=3

Apply operator precedence

/ and * → higher precedence than - (subtraction).

- They are evaluated left to right.
- X = 9 - 12 / 3 - 3 * 2 – 1

Do division and multiplication

- 12 / 3 = 4
- 3 * 2 = 6

X = 9 - 4 - 6 – 1

Left-to-right subtraction

- 9 - 4 = 5
- 5 - 6 = -1
- -1 - 1 = -2

Answer :X=-2

Explain assignment and compound assignment operator with an example.

- An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Compound Assignment Operators: These are shorthand versions of assignment and arithmetic/bitwise operators. Instead of writing both operations separately, you can combine them.

Operator	Assignment Expression	Shorthand Assignment
+ (plus)	a = a + b	a += b
- (minus)	a = a – b	a -= b
* (asterisk)	a = a * b	a *= b
/ (slash)	a = a / b	a /= b
% (percentage)	a = a % b	a %= b
& (bitwise AND)	a = a & b	a &= b
| (bitwise OR)	a = a | b	a |= b
^ (XOR)	a = a ^ b	a ^= b
<< (left shift)	a = a << b	a <<= b
>> (right shift)	a = a >> b	a >>= b

Define type conversion in C. Explain its types with suitable examples.

- The process of **converting** one type of **data type** to another, say changing **int to float or float to int** is known as data type conversion or type casting.

The general syntax of type casting is

```
(data type)variable
```

Data type conversion can be done in two ways:

1. Implicit type conversion
2. Explicit type conversion

Implicit type conversion

- This type of conversion is usually **performed by the compiler**, when necessary, without any commands by the user. Thus, it is also called "***Automatic Type Conversion or integer promotion*** ".
- By default, in C, when more than one data types are used in an expression, the **smaller type is promoted** to the larger type.
- Because of implicit type conversion rules in C: If the operands have different types, then all smaller integer types **(char, short)** are automatically converted to type that appears highest in the hierarchy like **int** when used in an expression.

Example:

```
#include <stdio.h>

int main() {
    int i = 17;
    char c = 'c'; /* ASCII value is 99 */
    float sum;

    sum = i + c; //c (char) is promoted to int first.

    printf("Value of sum: %f\n", sum);

    return 0;
}
```

Output:

```
Value of sum : 116.000000
```

Explicit type conversion

In Explicit type conversion, the user explicitly defines within the program the datatype, to which the operands/variables of the expression need to be converted. This is **user defined.**

Syntax : (datatype)variable

```
#include <stdio.h>
int main() {
    double a = 4.5;
    double b = 4.6;
    double c = 4.9;
    int result;
    result = (int)a + (int)b + (int)c; // Explicitly casting double to int, telling the compiler
directly to convert double → int.
    printf("Result = %d\n", result);
    return 0;
}
```

```
Output: Result = 12
```

Write a C program that computes the size of int, float, double and char variables

// Use the special operator, **sizeof()** to know size of the data types

Examples: a = sizeof (int);

y = sizeof (avg);

Evaluate the following expressions: Let a = 5, b = 5
(i) c = a++ + ++a
(ii) b -= -a-- - --b

(i) Initially, a = 5.
a++ → value used = 5, then a becomes 6.
++a → increment first → a = 7, then use a = 7.
So, expression c = 5 + 7 = 12.
Final values: c = 12, a = 7.

(ii) a = 5, b = 5.
a-- → use 5, then a = 4.
--b → decrement first → b = 4, use 4.
-a-- - --b = -(5) - (4) = -5 - 4 = -9
Now b -= -9 → b = b - (-9) = b + 9.
Current b = 4, so b = 4 + 9 = 13.

List and explain various unformatted Input and output in C.

- Unformatted I/O functions are those which do not specify data types and way in which it should be read or printed.
- In C, unformatted I/O functions are those that do not allow format specifiers (like %d, %f, etc.). They work directly with characters or strings.
- **Input: getchar(), gets()**
- **Output: putchar(), puts()**

Unformatted Input:

- **getchar():** Reads a single character from standard input (keyboard). This function is declared in stdio.h(header file).
- When executed, waits until any character of the keyboard is keyed in, followed by pressing the ENTER (\n) key.
- Only after ENTER key is pressed, execution is returned to the next statement of program.

syntax: ch = getchar();

```
#include <stdio.h>

int main() {
   char ch;
ch = getchar();   // waits for user input + Enter
   printf("You entered: %c\n", ch);
   return 0;
}
```

Using formatted input statement

```
#include <stdio.h>

int main() {
   char ch;
   printf("Enter a character: ");
   scanf("%c", &ch);
printf("You entered: %c\n", ch);

   return 0;
}
```

gets() : It is a standard input function that is used to read a string(a sequence of characters) from the user. It accepts all the characters on the keyboard as input and will terminate only when the ENTER key (\n) is pressed.

Syntax: gets(string);

```
#include <stdio.h>
int main() {
    char name[20];
gets(name);   // reads full line including spaces until
Enter key is pressed

    printf("name=%s\n", name);
    return 0;
}
```

Using Formatted input

```
#include <stdio.h>

int main() {
   char name[20];
printf("Enter your name: ");
   scanf("%s", name);   // no & needed
   printf("Hello, %s\n", name);

   return 0;
}
```

Unformatted Output:

putchar(): The putchar() function is used to display a single character at a time by passing that character directly to it or by passing a variable that has already stored a character. This function is declared in stdio.h(header file).

```
#include <stdio.h>
int main()
{
   char ch;
ch = getchar(); // Reads a single character and stores in ch
   putchar(ch); // Displays that character
return 0;
}
```

Using formatted output

```
#include <stdio.h>

int main() {
   char ch = 'A';     // store character

   printf("Character = %c\n", ch);   // %c for character

   return 0;
}
```

puts(): This is a function from <stdio.h> used to print a string to the standard output (screen).
This function has a built-in next line function and unlike printf("%s", str);, you don’t need format specifiers.

```
#include <stdio.h>

int main() {
   puts("C Programming is powerful!");
   return 0;
}
```

Output:

```
C Programming is powerful!
```

Using formatted output

```
#include <stdio.h>

int main() {
   char str[] = "C programming is powerful";

   printf("%s\n", str);   // %s prints the whole string

   return 0;
}
```

Illustrate formatted input for different data type with example.

Formatted input functions are those which specifies data types and way in which it should be read. Data is entered in a structured way (the computer knows if it’s a number, word, or decimal).

scanf() function: formatted input function which reads input from the standard input device.

Syntax :

scanf(“control string”, address_list);

or

scanf(“control string”, &var1, &var2,…&varn);

→“control string” - a set of format specifiers that tell scanf what type of data to read (%d, %f, %c, %s, etc.)..

→address_list or &var1, &var2,…&varn - specify the address locations of the variables where the data is stored.

Example:

scanf(“%d”, &n); //reads integer value and stores in address of n

scanf(“%c”, &n); //reads character and stores in address of n

Example:

```
printf("Enter age (int): ");
  scanf("%d", &age);

  printf("Enter salary (float): ");
  scanf("%f", &salary);

  printf("Enter value of pi (double): ");
  scanf("%lf", &pi);

  printf("Enter grade (char): ");
  scanf(" %c", &grade);

  printf("Enter your name (string): ");
  scanf("%s", name);     // no & needed for strings
```

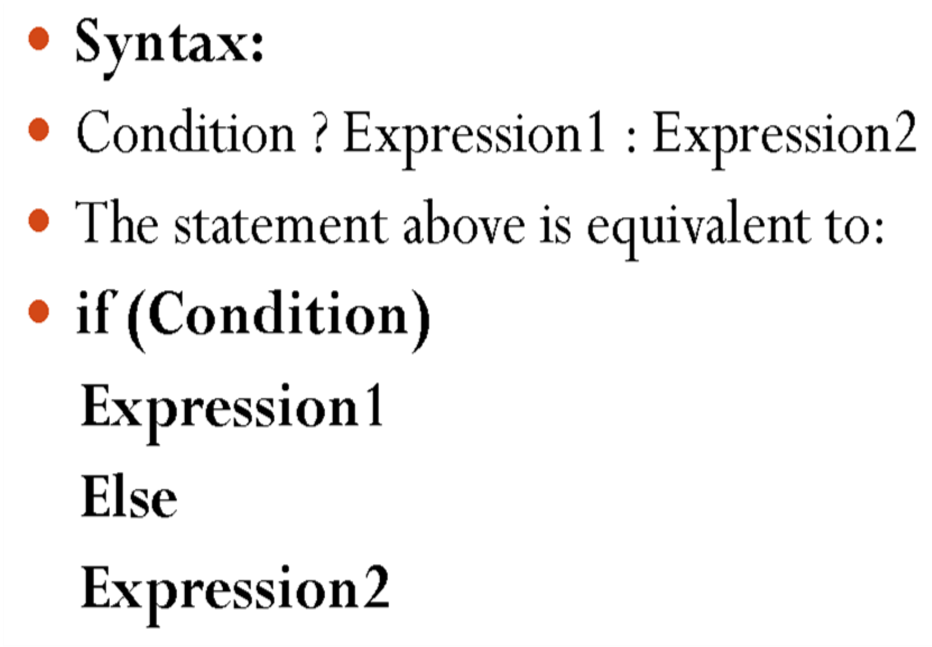
What is the use of preprocessor directive? Write any two preprocessor directives in C

The **preprocessor** is a program that processes source code before it is passed to the compiler. It works according to **preprocessor directives**, which are special instructions placed in the source code before the main() function. When the source code is examined, the preprocessor searches for these directives. If found, it performs the required actions (such as file inclusion or macro substitution) and then passes the updated code to the compiler. Preprocessor directives always begin with the symbol # and **do not require a semicolon** at the end. Examples are #define, #include.

Explain the usage of conditional operator in C.

Conditional Operator: (Ternary)

- It takes three operands.



- **Syntax:**
- Condition ? Expression1 : Expression2
- The statement above is equivalent to:
- **if (Condition)**

  Expression1

  Else

  Expression2

It uses character pair	? :
Its general syntax is	condition? expression1 : expression2

condition is a test condition, usually a relational expression like a > b, **expression 1 and expression 2** can be any valid arithmetic expressions or variables or constants.

Conditional Operator: (Ternary)

The **? :** operator pair works as follows:

- If the result of the **condition** is TRUE, exp1 is evaluated and the value of exp1 is the value of conditional operation.
- If the value of **condition** is FALSE, exp2 is evaluated and the value of exp2 is the value of the conditional operation.

Example

m = x > y ? a : b – 146;

- if x > y is TRUE, the value of ‘a’ is assigned to ‘m’,
- or if x > y is FALSE, the value of ‘b–146’ is assigned to ‘m’.

Example: total > 60 ? grade = ‘P’: grade = ‘F’;

Evaluate the following expressions: Let x = 3, y = 4, z = 4.
c = (z >= y >= x ? 100 : 200)

Break down relational part,

z >= y >= x   means   (z >= y) >= x

z >= y → 4 >= 4 → 1 (true)

Now 1 >= x → 1 >= 3 → 0 (false)

Apply conditional operator ? :

c = (0 ? 100 : 200)

Answer: c = 200

write the output for the following snippet code

```
#include <stdio.h> int main()
{ int a = 5,b = 10,c = 15;
int result = (a > b) ? a: (b > c) ? b: c;
printf("Result: %d\n",result);
return 0;
}
```

result = 15

Evaluate the following expression:
10 != 10 || 5 < 4 && 8

Step 1: Evaluate < (less than) first
5 < 4 → 0
So expression becomes:
10 != 10 || 0 && 8

Step 2: Evaluate !=
10 != 10 → 0
Now:
0 || 0 && 8

Step 3: Evaluate ||
0 || 0 → 0

Step 4: Evaluate &&
0 && 8 → 0

Final Answer = 0 (false)

100 % 20 <= 20 – 5 + 100 % 10 – 20 == 5 >= 1 != 20

Step 1: Evaluate %

100 % 20 = 0

100 % 10 = 0

Expression becomes:

0 <= 20 - 5 + 0 - 20 == 5 >= 1 != 20

Step 2: Evaluate + and – (left to right)

20 - 5 = 15

15 + 0 = 15

15 - 20 = -5

Expression becomes:

0 <= -5 == 5 >= 1 != 20

Step 3: Evaluate relational (<=, >=) (left to right)

0 <= -5 → 0 (false)

So: 0 == 5 >= 1 != 20

Next: 5 >= 1 → 1 (true)

So: **0 == 1 != 20**

Step 4: Evaluate equality (==, !=) left to right

0 == 1 → 0

Now:

0 != 20

0 != 20 → 1 (true)

From table:

1. % → level **3**
2. + , – → level **4**
3. <, <=, >, >= → level **6**
4. ==, != → level **7**

a += b *= c -= 5 where a=3, b=5 and c=8

Step 1: Evaluate c -= 5

c = c - 5

c = 8 - 5

c = 3

Step 2: Evaluate b *= c

b = b * c

b = 5 * 3

b = 15

Step 3: Evaluate a += b

a = a + b

a = 3 + 15

a = 18

Operator precedence

From precedence table:

*= , += , -= → **assignment operators → level 14, right to left associativity**

Explain simple-if, else-if , else if ladder and nested –if statements with syntax and examples for each.

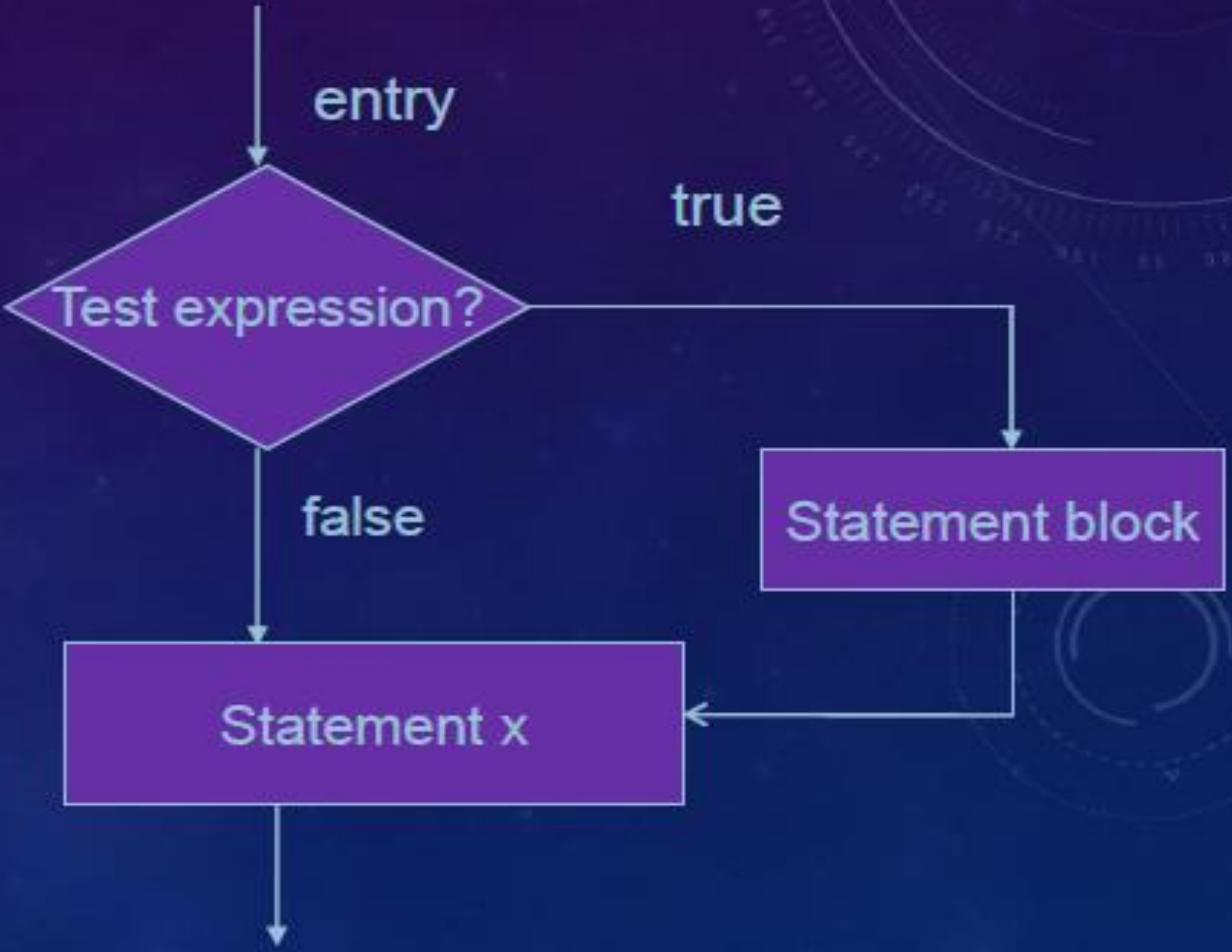
Simple if:

It is called **one way branching statement**.

It involves a test expression (a **relational** or conditional expression).

```
if (condition)
{
Statement-block; // body of if
}
statement –x; //if condition is true, statement block and statement-x are executed in sequence.
```

```
Syntax-  if ( Condition)
    {
        statement block;
    }
  statement x ;
Ex: if ( category == sports)
    {
        marks=marks+bonus;
    }
  printf ( “%f”,marks);
```



Example for if statement

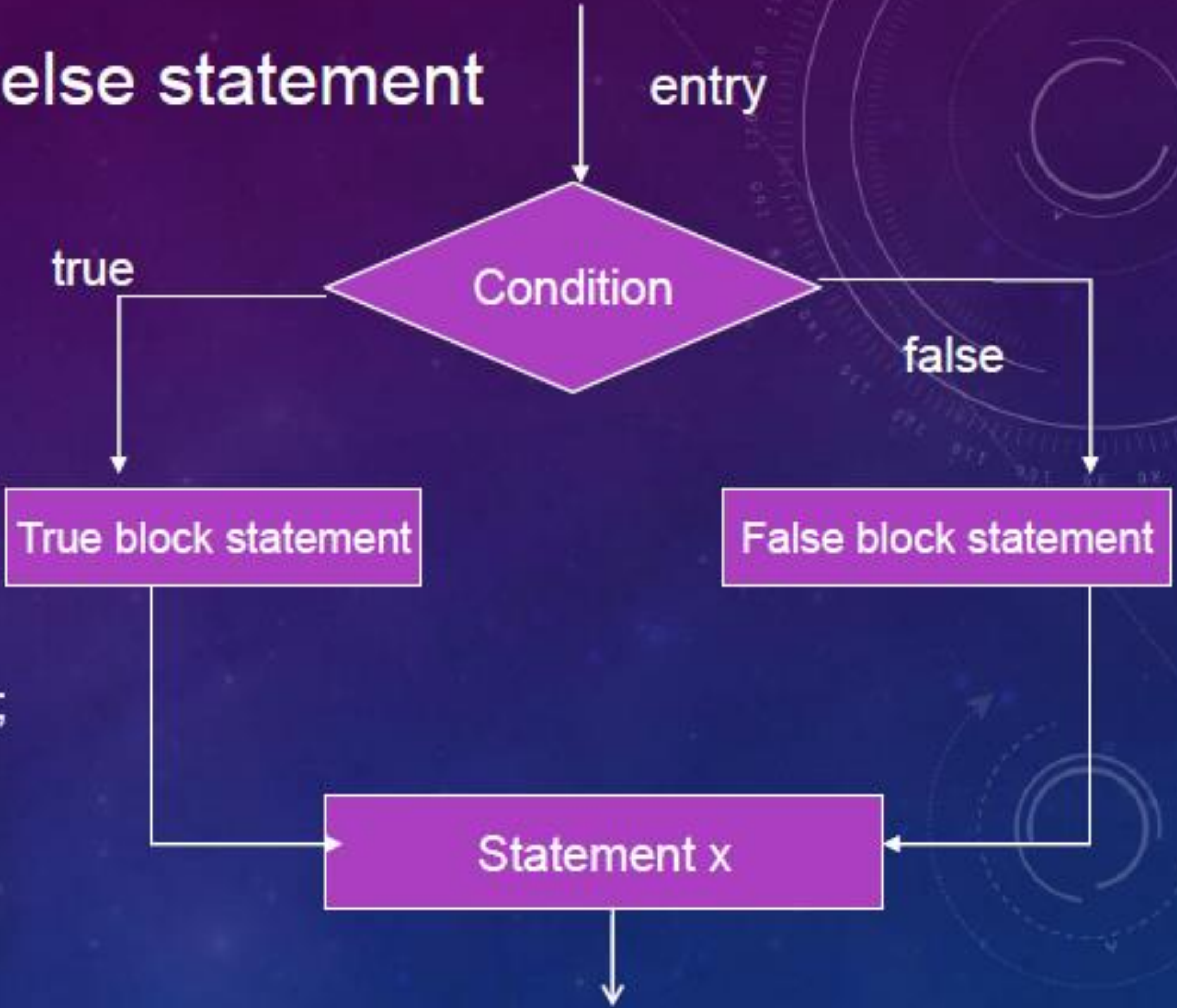
```
#include <stdio.h>
int main() {
   int number = 14;     // initialization

   if(number % 2 == 0) { // condition check
      printf("This is even number\n");
   }

   printf("The number is %d\n", number);

   return 0;
}
```

Else if statement

- The **if** statement provides one way branching, i.e., it executes the statement/s associated with **if**, only when the test expression is true, but does nothing if it is false.
- The **if-else** statement is **an extension** of the **if** statement. The **if-else** statement is a *two way branching*, i.e., it executes one set of statement/s if the test expression is true or it executes another set of statement/s if the test expression is false.

The if…else statement

```
Syntax-   if( Condition)
   {
      true block statement;
   }
  else
  {
      false block statement;
   }
  statement x;
```



Example for else if statement

Write a program to determine whether a number is even or odd using if..else

```
#include <stdio.h>

int main()
{
    int a;
    printf("Enter an integer: ");
    scanf("%d", &a);

    if (a % 2 == 0) {
        printf("%d is an even number\n", a);
    } else {
        printf("%d is an odd number\n", a);
    }

    return 0;
}
```

THE ELSE-IF LADDER

In C, **if else if ladder** is an extension of if else statement used to test a series of conditions sequentially, executing the code for the first true condition.

A condition is checked only if all previous ones are false. Once a condition is true, its code block executes, and the ladder ends

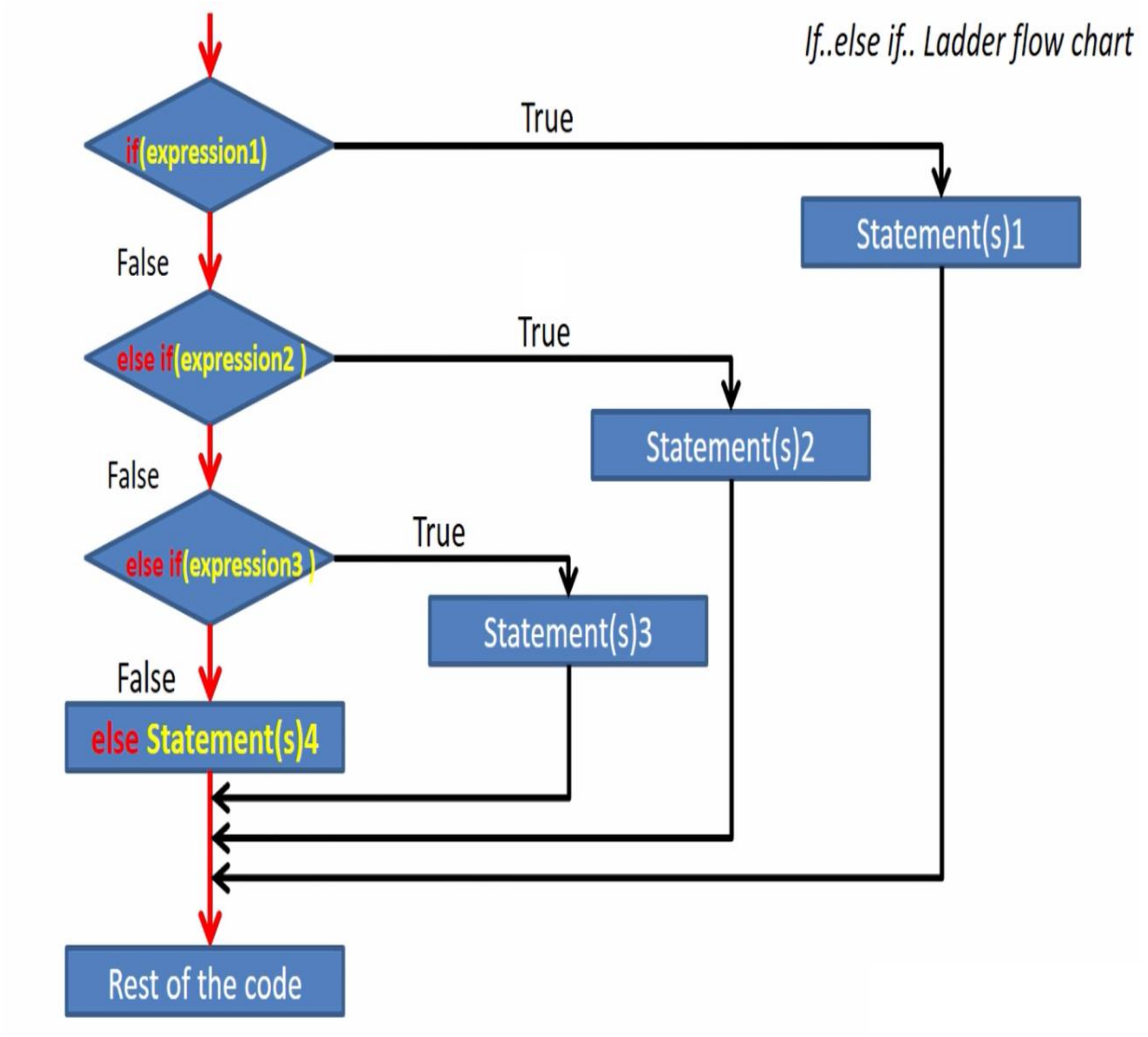
The else if ladder

syntax-:

```
if  ( condition 1)
  statement 1 ;
     else if ( condition 2)
        statement 2 ;
           else if ( condition 3)
               statement 3 ;
                   else if( condition n)
                       statement n ;
                           else
                               default statement ;
        statement x;
```

The else-if ladder

```
if (test expression1)
{
  // statement(s)
}
else if(test expression2)
{
  // statement(s)
}
else if (test expression3)
{
  // statement(s)
}
...
else
{
  // statement(s)
}
```



C program which display the grade of student using else if ladder

```
#include<stdio.h>
int main(){
  int marks;
  printf("Enter the marks of a student:");
  scanf("%d",&marks);
  if(marks <=100 && marks >= 90)
    printf("Grade=A");
  else if(marks < 90 && marks>= 80)
    printf("Grade=B");
  else if(marks < 80 && marks >= 70)
    printf("Grade=C");
  else if(marks < 70 && marks >= 60)
    printf("Grade=D");
  else if(marks < 60 && marks > 50)
    printf("Grade=E");
  else if(marks == 50)
    printf("Grade=F");
  else if(marks < 50 && marks >= 0)
    printf("Fail");
  else
    printf("Enter a valid score between 0 and 100");
  return 0;
}
```

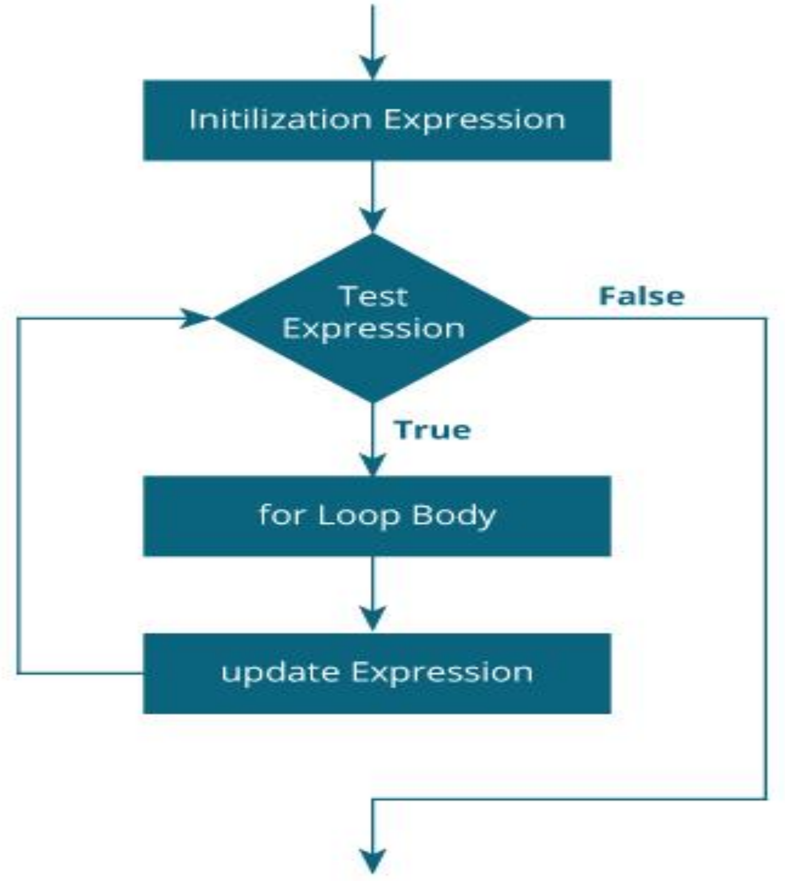
Explain for loop with a syntax and example.

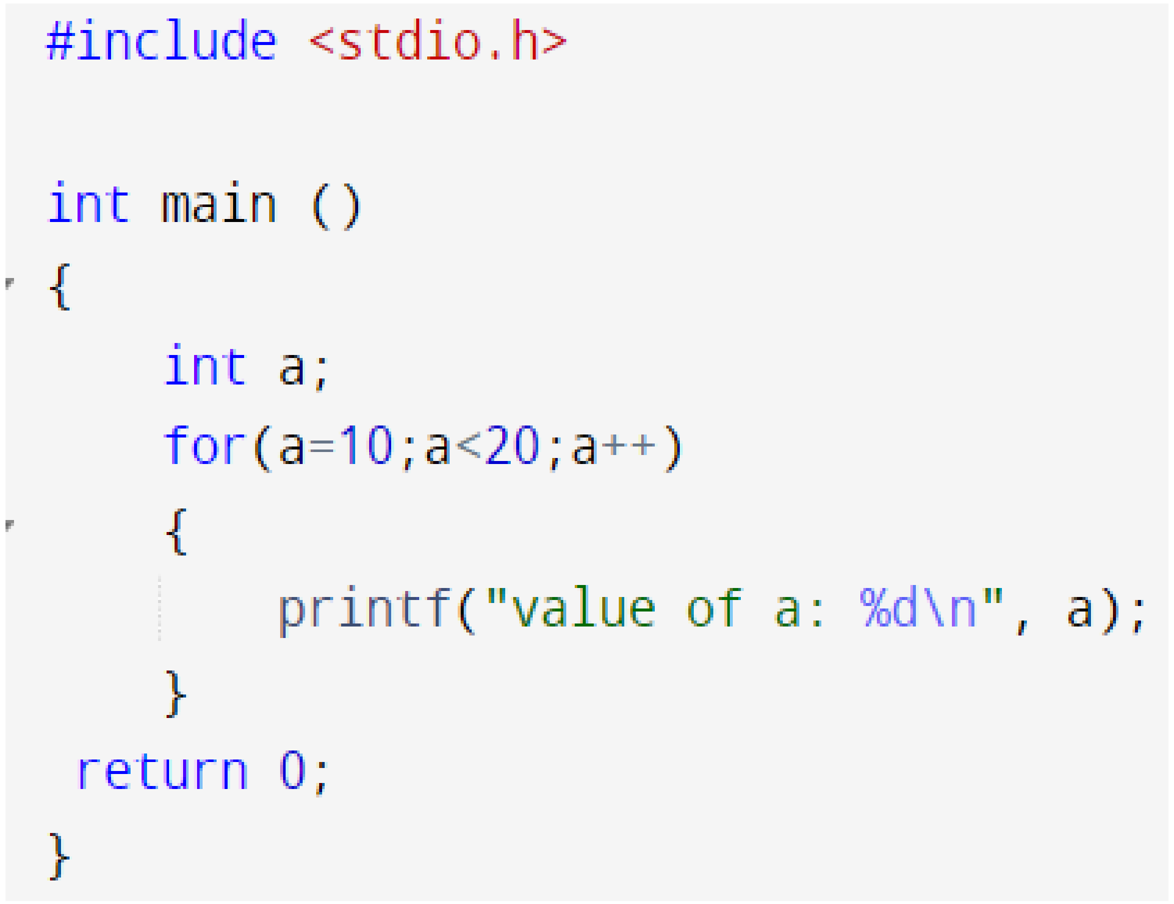
for statement

- for is an *entry – controlled loop* statement.
- This statement is used when the programmer knows how many times a set of statements are executed.

<u>Syntax of for statement:</u>

```
for (initialization; test_condition; update_expression)
{
        statement1;
        statement2;
}
Statement x;
Statement y;
```





```c
#include <stdio.h>

int main ()
{
    int a;
    for(a=10;a<20;a++)
    {
        printf("value of a: %d\n", a);
    }
 return 0;
}
```

What is pretest and post test looping? Explain them with examples

In a **pre-test loop (entry controlled),** the condition is checked before the loop body executes. If the condition is false initially, the loop body may not execute at all. **Examples: while loop and for loop in C.**

In a **post-test loop (exit controlled),** the loop body is executed first, and the condition is tested after. This guarantees that the loop body will execute at least once, even if the condition is false. **Example: do...while loop in C.**

Program to find simple interest: SI = PTR/100

```
#include <stdio.h>

int main() {
   float P, T, R, SI; // Input principal, time, and rate
   printf("Enter Principal amount: ");
   scanf("%f", &P);

   printf("Enter Time (in years): ");
   scanf("%f", &T);

   printf("Enter Rate of Interest: ");
   scanf("%f", &R);

   SI = (P * T * R) / 100;

 printf("Simple Interest = %.2f\n", SI);

   return 0;
}
```

Program to find the vowels using switch statement.

```
#include <stdio.h>
int main() {
   char ch;
printf("Enter a character: ");
   scanf("%c", &ch);
switch(ch) {
case 'a':
   printf("%c is a vowel.\n", ch);
   break;
case 'e':
   printf("%c is a vowel.\n", ch);
   break;
case 'i':
   printf("%c is a vowel.\n", ch);
   break;
case 'o':
   printf("%c is a vowel.\n", ch);
   break;
case 'u':
   printf("%c is a vowel.\n", ch);
   break;
case 'A':
         printf("%c is a vowel.\n", ch);
         break;
      case 'E':
         printf("%c is a vowel.\n", ch);
         break;
      case 'I':
         printf("%c is a vowel.\n", ch);
         break;
      case 'O':
         printf("%c is a vowel.\n", ch);
         break;
      case 'U':
         printf("%c is a vowel.\n", ch);
         break;
      default:
         printf("%c is not a vowel.\n", ch);
         break;
   }

   return 0;
}
```

Develop a C program to find the largest of three numbers using ternary operator

```
#include <stdio.h>

int main() {
   int a, b, c, largest;

printf("Enter three numbers: ");
   scanf("%d %d %d", &a, &b, &c);

   // Using ternary operator
   largest = (a > b) ?  ((a > c) ? a : c) :  ((b > c) ? b : c);

   // Output result
   printf("The largest number is: %d\n", largest);

   return 0;
}
```

Note:
(a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
First compares a and b. If a > b, then it compares a with c.
Else, it compares b with c. The final result is the largest among all three.