Ques 1. What is computational thinking? Describe various steps involve in it?

Ans. Computational Thinking is a **problem-solving process** that involves breaking down complex problems into manageable steps and using logical reasoning.

Computational thinking helps in understanding and **developing efficient solutions** for problems by following systematic methods.

1) **Decomposition:** Decomposition is the process of breaking down a complex problem or system into smaller, more manageable parts. By focusing on individual components, it becomes easier to understand and solve.
   Example: Imagine you are asked to organize a school event. Instead of handling everything at once, break it down:
   - **Venue Selection**
   - **Invitations**
   - **Food Arrangements**
   - **Decoration**
   - **Entertainment**

2) **Pattern Recognition:** Pattern recognition involves identifying similarities or patterns within a problem. Allows for understanding of repetitive elements, which can be used for optimization.
   Example:
   - If you're solving puzzles, like a Rubik's cube, you may notice patterns in the colors and their positions. Once you've identified a pattern, you can apply a set of moves that often work to solve the puzzle.
   - In coding, **loops** are used because repetitive tasks can be automated by recognizing that the same block of code needs to run multiple times.

3) **Abstraction:** Abstraction is the process of focusing only on the relevant details and ignoring unnecessary information.
   Example:
   - When you're driving, you don’t need to think about how the engine works. You only care about **steering, speed control, and direction**—the relevant information needed to get you to your destination.

4) **Algorithm Design: Algorithm Design** is the process of defining a clear, step-by-step procedure or set of instructions to solve a specific problem. The goal of an algorithm is to take inputs (data) and transform them into the desired outputs (solutions) in an efficient and organized way.
   Algorithms can be applied to a wide range of problems, from simple tasks like adding two numbers, to complex tasks like sorting a list or even processing huge amounts of data.
   Real-life example: In coding, a simple algorithm to sort a list of numbers from lowest to highest could be:
   1. Compare the first two numbers.
   2. Swap them if they are in the wrong order.
   3. Move to the next number and repeat the process.
   4. Continue until the list is sorted.

Ques 2. Describe the key difference between modular programming language and object-oriented programming language?

Ans.

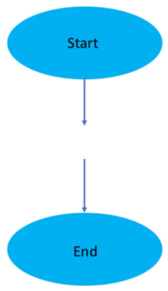
S.No.	Modular Programming Language	Object-Oriented Programming Language
1.	Program is divided into **modules** (functions or procedures).	Program is divided into **objects** (real-world entities).
2.	Focus is on **functions** and procedures.	Focus is on **data** and objects.
3.	Data is **shared** among functions.	Data is **encapsulated** inside objects and accessed through methods.
4.	Does not support features like inheritance and polymorphism.	Supports **inheritance**, **polymorphism**, and **encapsulation**.
5.	Example languages: C, Modula-2	Example languages: Java, C++, Python
6.	Less reusable and harder to maintain for large projects.	More reusable and easier to maintain through objects and classes.
7.	No concept of classes and objects.	Uses **classes** and **objects** as the main building blocks.

Ques 3. What are the various flowchart symbols? Explain with diagram? Draw flowchart to calculate and print the sum of First N prime numbers?

Ans. A **flowchart** is a diagram that shows the **step-by-step flow of a process** or **algorithm** using different **symbols**. Each symbol represents a specific type of action or step.

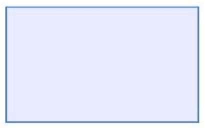
1. Start / End Symbol (Oval)

- This symbol is used to indicate **where the flowchart begins and ends**.
- The word **"Start"** is written inside the oval at the beginning and **"End"** at the termination.
- Every flowchart must have one **start** and at least one **end**.
- **Example:** Start of a program or when the output is displayed and the program finishes.



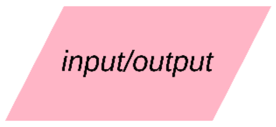
2. Process Symbol (Rectangle)

- This symbol shows a **process**, **instruction**, or **operation**.
- It is used to represent steps like calculations, data processing, or assignments.
- **Example:** SUM = A + B or any arithmetic operation.


Process Symbol

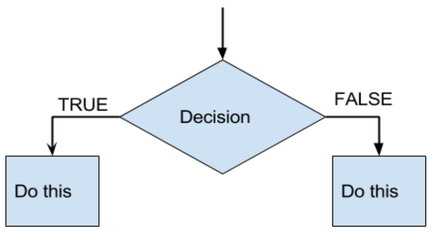
3. Input / Output Symbol (Parallelogram)

- This symbol is used to represent **input** or **output** operations.
- **Input** means taking data from the user, and **output** means displaying data to the user.
- **Example:** "Enter two numbers" or "Display the result".

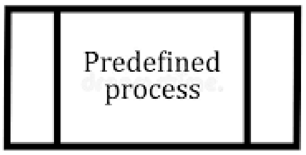


4. Decision Symbol (Diamond)

- It represents a **decision-making step** in the flowchart.
- This symbol is always followed by **two or more branches** depending on the answer (e.g., YES/NO or TRUE/FALSE).
- **Example:** "Is A > B?" If yes, follow one path; if no, follow another.

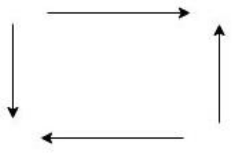


5. Predefined Process Symbol (Rectangle with Double Lines)



- This symbol indicates a **sub-process** or **subroutine** that is defined elsewhere.
- It helps to avoid repeating the same set of steps multiple times.
- **Example:** calling a function like Sort() or Calculate().

7. Flow Lines (Arrows)



- Arrows are used to **show the direction of flow** from one step to another.
- They connect all the symbols together and guide the order in which steps are executed.
- **Example:** From “Start” → “Input” → “Process” → “Output” → “End”.

Draw flowchart to calculate and print the sum of First N prime numbers?

Ques 4. What do you mean by the scope of the variable?

Ans. In **C**, the **scope of a variable** means **the part of the program where the variable can be accessed or used**. Scope tells you where in the program a variable is “visible” or “alive.”

There are mainly **two types of variable scope** in C—

a) **Local Scope:**

- A variable declared inside a function or a block (like { }) has local scope.
- It can only be used inside that function or block.
- It cannot be accessed outside it.
- It is created when the block starts and destroyed when the block ends.
- **Example:**

```
#include <stdio.h>
int main() {
   int x = 10;     // local variable
   printf("%d", x);
   return 0;
}
```

b) Global Scope:

- A variable declared **outside all functions** has **global scope**.
- It can be **used in any function** of the program.
- It exists **for the entire program**.
- **Example:**

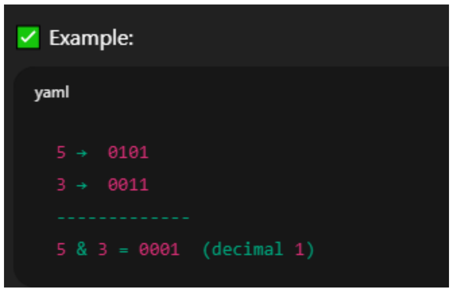
```
#include <stdio.h>
int x = 10;   // global variable

int main() {
   printf("%d", x);  // accessible here
   return 0;
}
void display() {
   printf("%d", x);  // accessible here too
}
```

Ques 5. What are the bitwise operators and how bitwise operators being different from other operators? Explain with a proper example to demonstrate the working of all the bitwise operators?

Ans.

- Bitwise operators work **directly on binary representations** of integers.
- They can only be applied to **integer types** (`int`, `short`, `long`, `char`), not floating-point types.
- Bitwise operations are **very fast** because they are performed at the hardware level.
- These operators are often used in **system programming**, **embedded systems**, **networking**, **encryption**, and **bit masking**.
- The **result of a bitwise operation** is always an integer.
- Bitwise operators do **not perform logical comparison**; they only check or change bits.
- Here are the **main bitwise operators**:

1) **Bitwise AND (&):**
   - Compares each bit of two numbers.
   - The result bit is **1 only if both bits are 1**, otherwise 0.

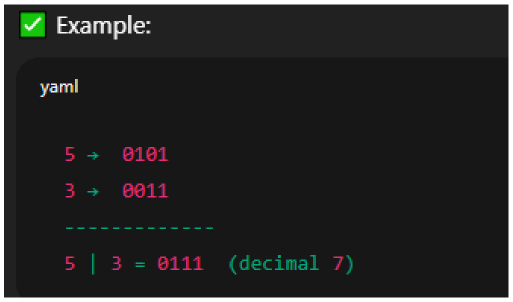


✓ Example:

yaml

```
5 → 0101
3 → 0011
-------------
5 & 3 = 0001  (decimal 1)
```

2) **Bitwise OR (|):**
   - Compares each bit of two numbers.
   - The result bit is **1 if at least one bit is 1**, otherwise 0

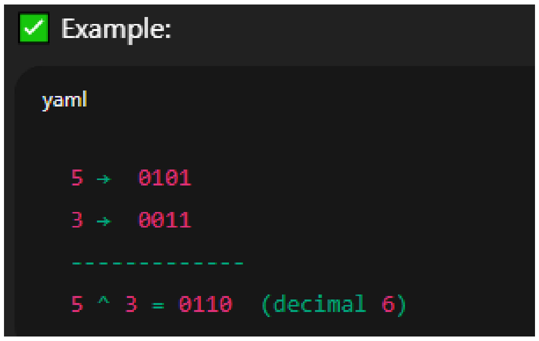


✓ Example:

yaml

```
5 → 0101
3 → 0011
-------------
5 | 3 = 0111  (decimal 7)
```

3) **Bitwise XOR (^)** ***(Exclusive OR):***
   - Compares each bit of two numbers.
   - The result bit is **1 if the bits are different**, and 0 if they are the same.

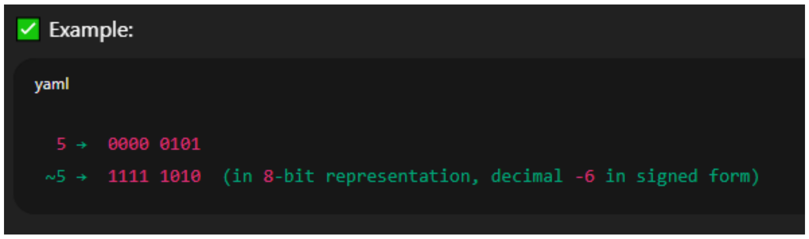


✓ Example:

yaml

```
5 → 0101
3 → 0011
-------------
5 ^ 3 = 0110  (decimal 6)
```

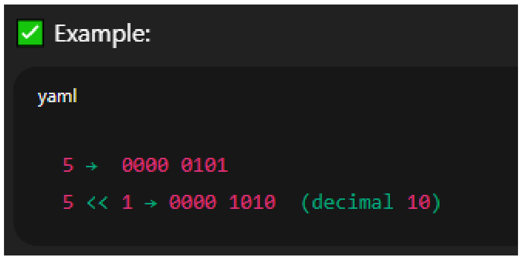
4) **Bitwise NOT (~) *(One's Complement):***
   - Inverts all the bits of the number.
   - 1 becomes 0 and 0 becomes 1.



✅ Example:

```yaml
 5 →  0000 0101
~5 →  1111 1010  (in 8-bit representation, decimal -6 in signed form)
```

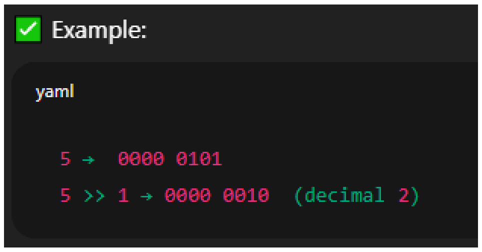
5) **Left Shift (<<):**
   - Shifts all bits to the **left** by a specified number of positions.
   - Each left shift multiplies the number by 2.



✅ Example:

```yaml
5 →  0000 0101
5 << 1 → 0000 1010  (decimal 10)
```

6) **Right Shift (>>):**
   - Shifts all bits to the **right** by a specified number of positions.
   - Each right shift divides the number by 2.



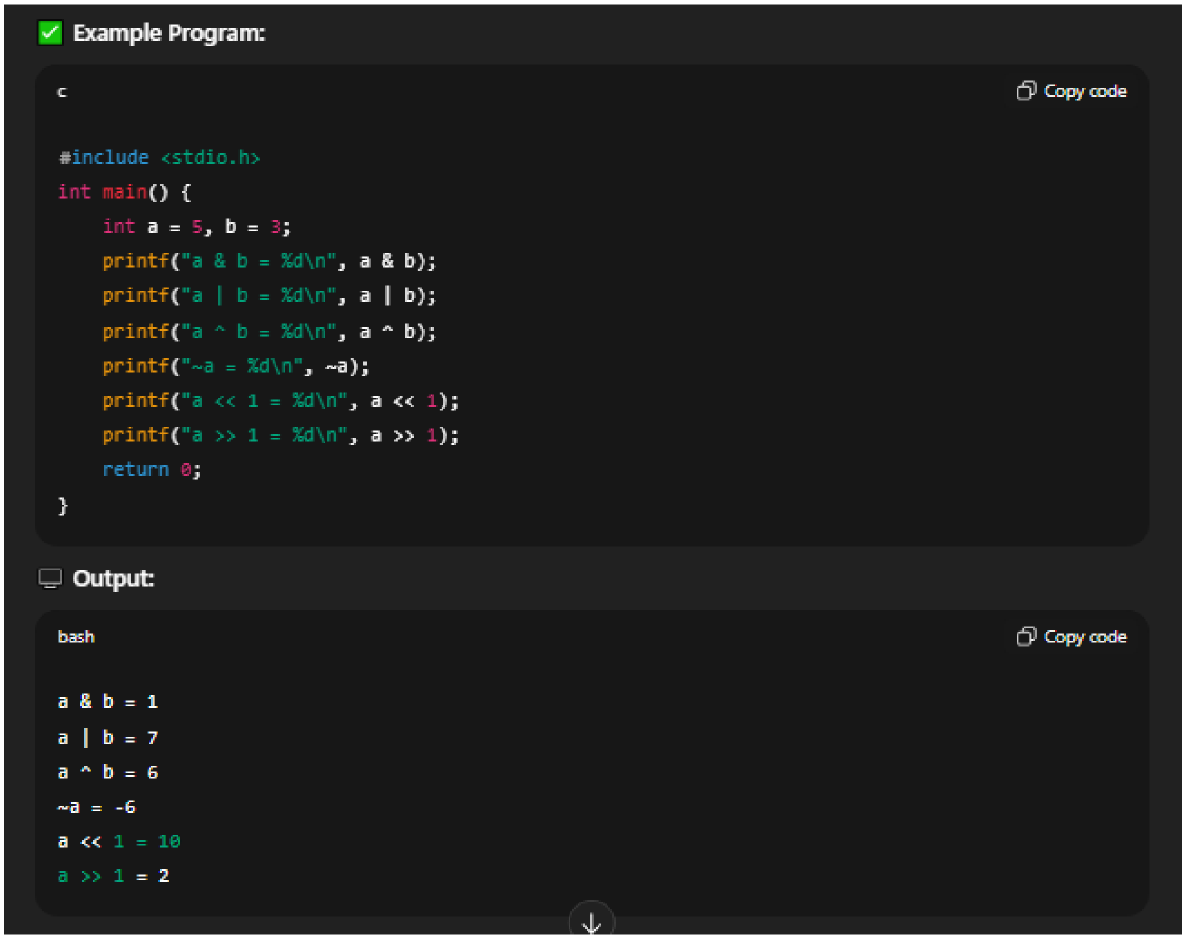
✅ Example:

```yaml
5 →  0000 0101
5 >> 1 → 0000 0010  (decimal 2)
```

➢ **how bitwise operators being different from other operators?**

- Bitwise operators are **different from other operators** in C because they work **directly on the binary bits** of numbers, not on their mathematical values like arithmetic or logical operators do.
- Bitwise operators operate **bit by bit** (0s and 1s) of integer data.
  Other operators (like `+`, `-`, `*`, `/`) work on the **entire value** of the variable.
- Bitwise operators work only on **integer data types** (`int`, `char`, `long`, etc.).
  Other operators can work on integers, floats, doubles, etc.
- Bitwise operations are **very fast** because they are done at the **processor level**.
  Arithmetic operations may take more time comparatively.
- Bitwise operators don't check if a number is "true" or "false"; they just manipulate **each bit**.
  Logical operators (`&&`, `||`, `!`) work with **boolean (true/false)** values.

Example:


✅ **Example Program:**

c

```c
#include <stdio.h>
int main() {
    int a = 5, b = 3;
    printf("a & b = %d\n", a & b);
    printf("a | b = %d\n", a | b);
    printf("a ^ b = %d\n", a ^ b);
    printf("~a = %d\n", ~a);
    printf("a << 1 = %d\n", a << 1);
    printf("a >> 1 = %d\n", a >> 1);
    return 0;
}
```

Output:

bash

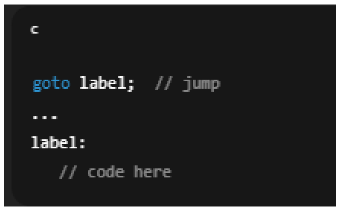
```
a & b = 1
a | b = 7
a ^ b = 6
~a = -6
a << 1 = 10
a >> 1 = 2
```

Ques 6. Explain goto, break and continue statements in C? WAP to print factorial of a number using goto statement?

Ans. goto, **break**, and **continue** are **control statements** used to **change the normal flow** of a program.

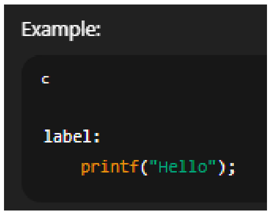
A) goto Statement: It transfers the control of the program **directly** to a **labeled statement** inside the same function.

- **Unconditional Jump:**
    - Unlike loops or conditions, `goto` **does not check any condition**.
    - It jumps directly to the label as soon as it is executed.



```c
goto label;  // jump
...
label:
   // code here
```

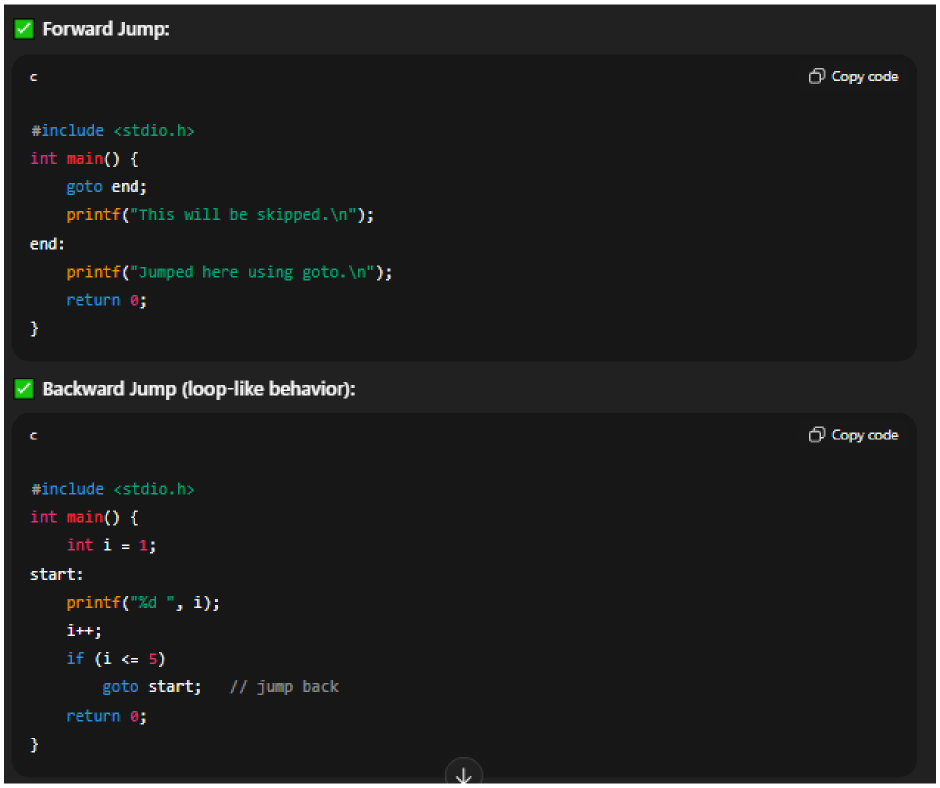
- **Label Declaration:**
    - The label is **an identifier followed by a colon (:)**.
    - The label can be written **before or after** the `goto` statement.
    - Labels must be **unique** inside a function.
    - Label name can be anything.



Example:

```c
label:
    printf("Hello");
```

- **Forward and Backward Jump:**
    - `goto` can jump **forward** (down the code) or **backward** (up the code).



✅ Forward Jump:

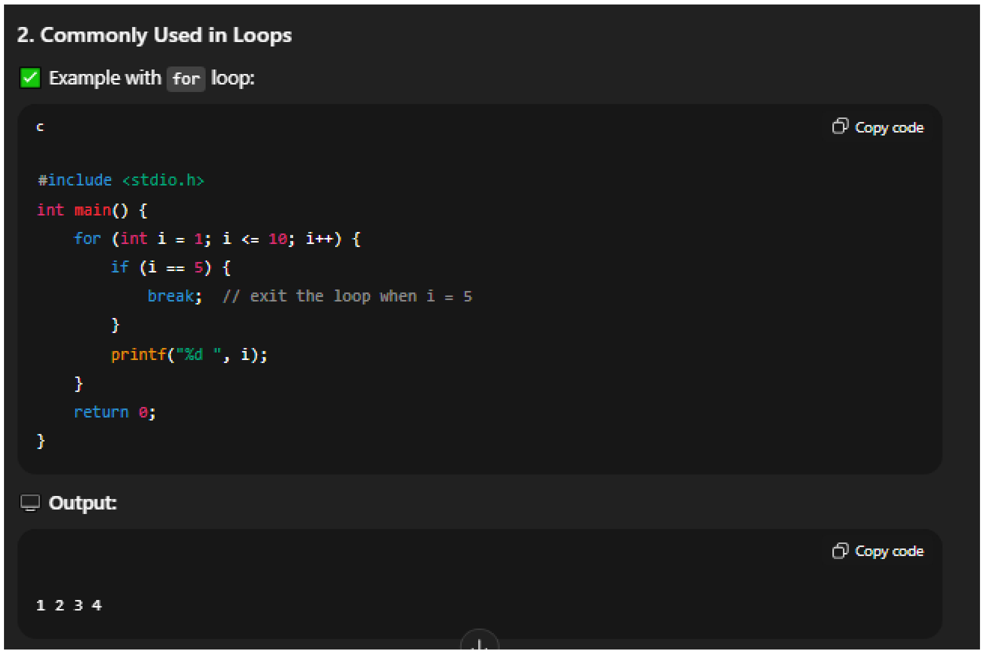
```c
#include <stdio.h>
int main() {
    goto end;
    printf("This will be skipped.\n");
end:
    printf("Jumped here using goto.\n");
    return 0;
}
```

✅ Backward Jump (loop-like behavior):

```c
#include <stdio.h>
int main() {
    int i = 1;
start:
    printf("%d ", i);
    i++;
    if (i <= 5)
        goto start;   // jump back
    return 0;
}
```

- **`goto` Works Only Within the Same Function:**
    - You cannot jump to a label **outside the current function**.
    - Labels are **local to the function** in which they are defined.

B) `Break` Statement: The **`break` statement** is used to **immediately terminate** the **nearest enclosing loop** (`for`, `while`, `do-while`) or a **`switch` statement**, and **transfer program control to the statement next to it**.



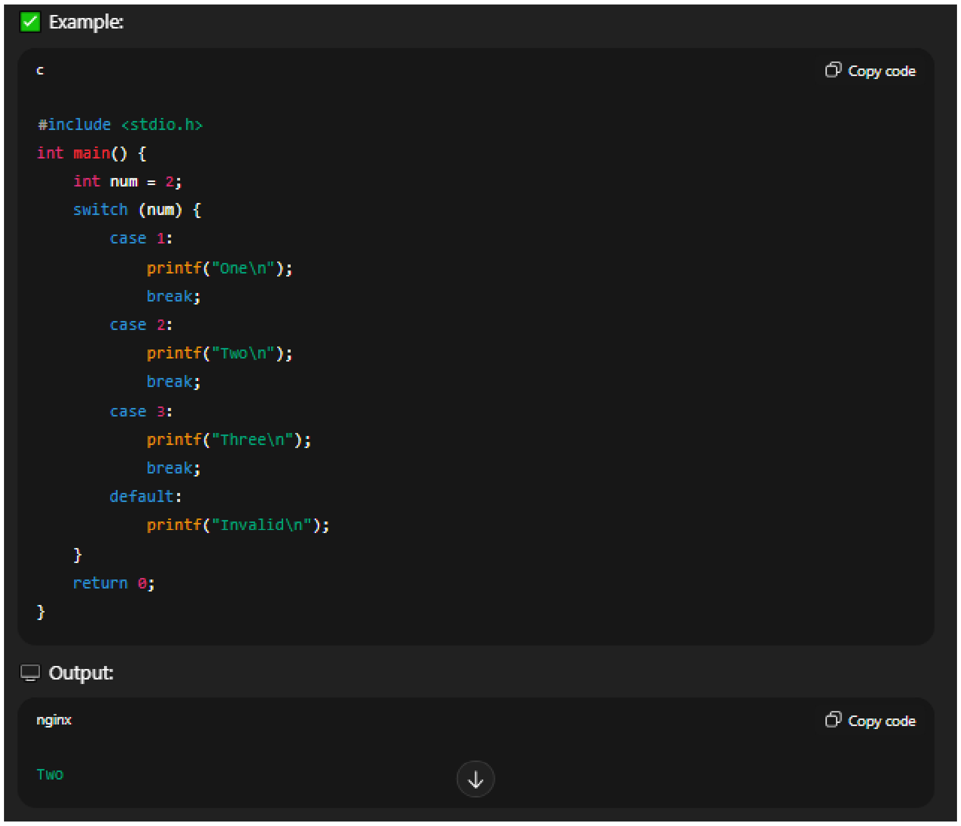
2. Commonly Used in Loops

✅ Example with `for` loop:

```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;  // exit the loop when i = 5
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2 3 4
```



✅ Example:

```c
#include <stdio.h>
int main() {
    int num = 2;
    switch (num) {
        case 1:
            printf("One\n");
            break;
        case 2:
            printf("Two\n");
            break;
        case 3:
            printf("Three\n");
            break;
        default:
            printf("Invalid\n");
    }
    return 0;
}
```

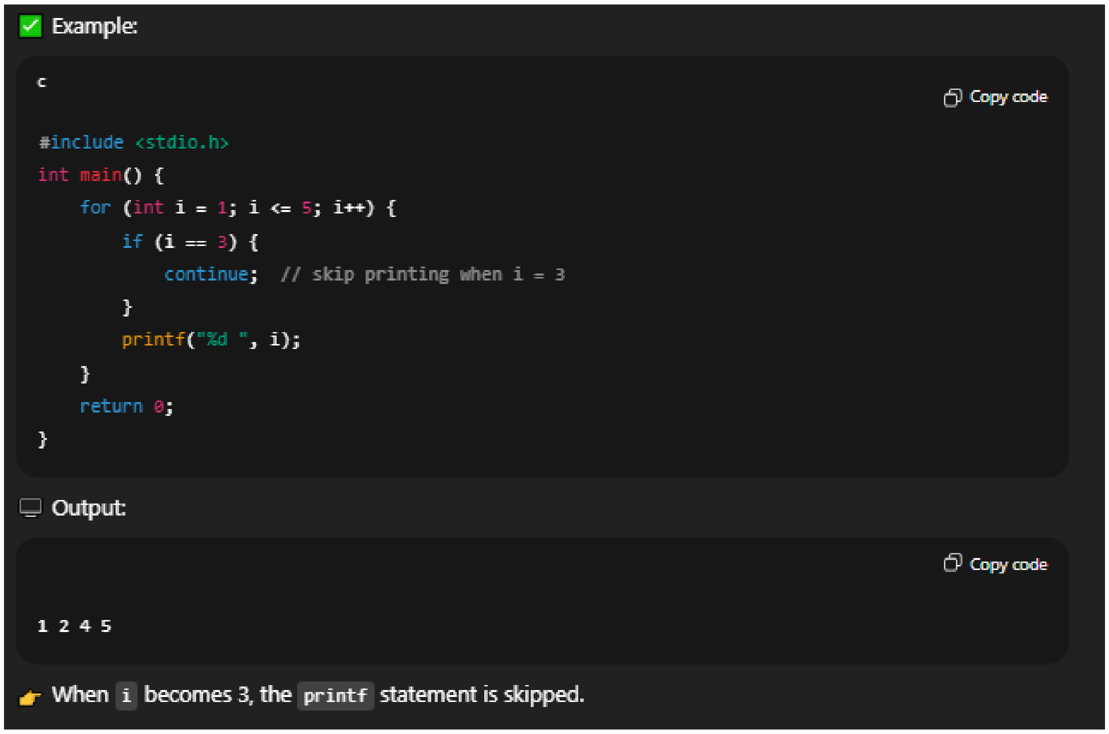
Output:

```
Two
```

C) `continue` Statement: The **`continue` statement** is used to **skip the remaining statements** in the **current iteration** of a loop and **jump directly to the next iteration** of that loop.

- `continue` can only be used inside loops (`for`, `while`, `do-while`).
- Unlike `break`, it **cannot be used in `switch` statements** directly.

- It can be Used in Nested Loops.



✅ Example:

```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // skip printing when i = 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2 4 5
```

👉 When `i` becomes 3, the `printf` statement is skipped.

Ques 7. What are top-down and bottom-up design approaches in programming?

Ans. In programming, **Top-Down** and **Bottom-Up** are two important **design and problem-solving approaches** used to build programs or systems.

1) **Top-Down Design Approach:** Top-Down approach is a method where we **start with the main problem** (big picture) and then **break it down into smaller, simpler sub-problems or modules** until each part can be easily coded.
    - Start by **understanding the big problem first**.
    - Then **divide the big problem into smaller and smaller parts**.
    - Finally, write the program for each small part and combine them together.

   Advantages of Top-Down:
    - You always **know the full structure** of the program.
    - Easier to **organize big projects**.
    - Easier to **understand and explain** to others.
    - Good for beginners because it follows **step-by-step planning**.

   Disadvantages of Top-Down:
    - You can't start coding immediately.
    - If the **top plan is wrong**, everything below it will also be wrong.
    - Changes in one top part can affect many lower parts

2) **Bottom-Up Design Approach:** Start by building the smallest parts first. Then join these small parts together to make the full program.

   How It Works (Step by Step):
   a) **Identify the small tasks** that the program will need.
      Example: For a result program:
      a. Function to input marks
      b. Function to add marks
      c. Function to calculate percentage
      d. Function to display result
   b) **Write code for these small parts** first.
   c) **Combine these small parts** to build the big program.

   Advantages of Bottom-Up:
    - You can **start coding immediately**.
    - Small parts are **easier to test** and debug.
    - Promotes **code reusability** — you can use the same function in different programs.
    - More flexible if you want to **change something** later.

   Disadvantages of Bottom-Up:
    - At the beginning, it's **not clear what the final structure** will look like.
    - Without a plan, it can become **confusing**.
    - Harder to manage very big projects.

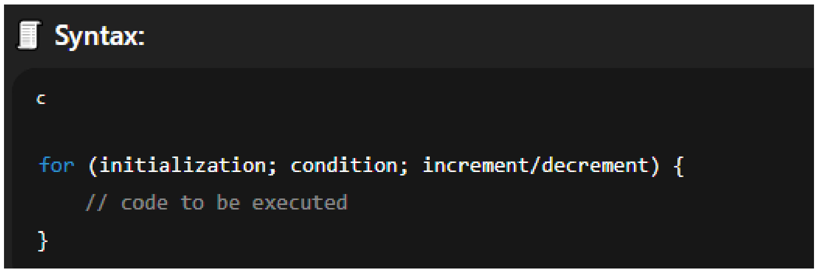
Ques 8. Explain the different kinds of loops available in C with suitable examples?

Ans. In C, **loops** are used to **repeat a block of code** multiple times which saves time, reduces code length, and makes programs more efficient.

There are **three main types of loops** in C—

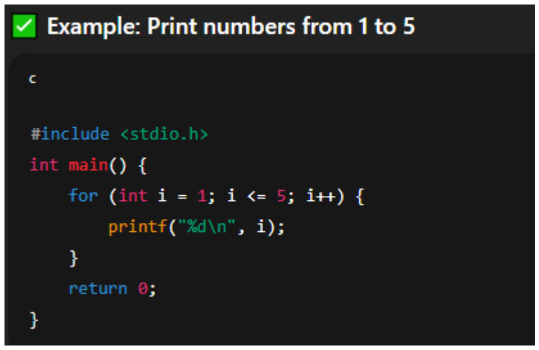
1) **for loop**
2) **while loop**
3) **do…while loop**

1) for Loop: The `for` loop is used when we **know in advance** how many times we want to repeat the code. It is an **entry-controlled loop**, meaning the **condition is checked first**, and if it's true, the loop runs.



Syntax:

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

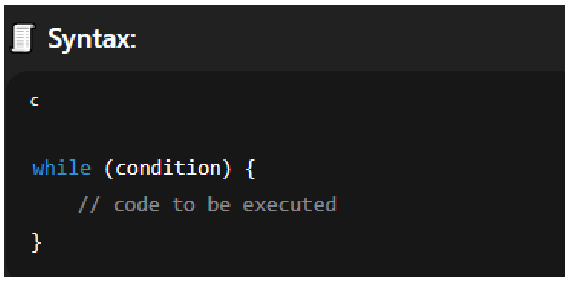


✓ Example: Print numbers from 1 to 5

```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

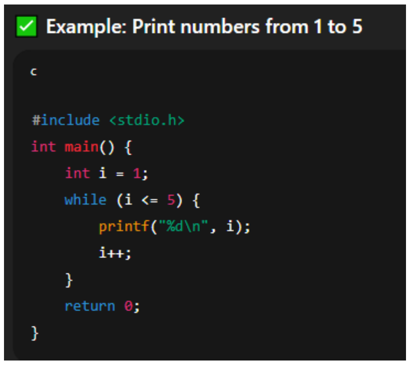
- **Initialization:** sets the starting value of the loop variable.
- **Condition:** checked before each iteration; if false, loop stops.
- **Increment/Decrement:** updates the loop variable after each iteration.

2) while Loop: The `while` loop is used when **we don't know in advance** how many times the loop should run. The **condition is checked before the loop body**, and if it's true, the loop runs.



Syntax:

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

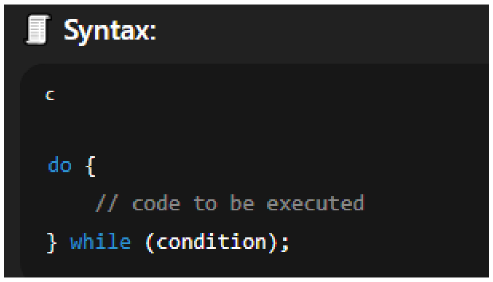


✓ Example: Print numbers from 1 to 5

```c
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

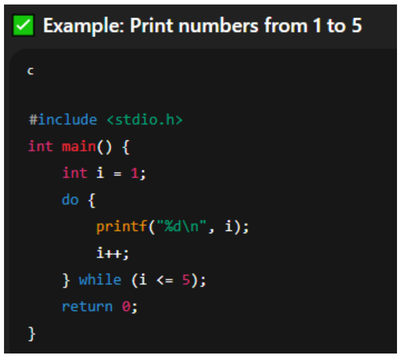
3) do…while Loop: The `do…while` loop is used when we **want the loop to run at least once**, no matter whether the condition is true or false.

The **condition is checked after the loop body**.



Syntax:

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



✓ Example: Print numbers from 1 to 5

```c
#include <stdio.h>
int main() {
    int i = 1;
    do {
        printf("%d\n", i);
        i++;
    } while (i <= 5);
    return 0;
}
```

Ques 9. What are the various control statements in C language?

Ans. In **C**, **control statements** are used to **control the flow of execution** in a program. They help the program **decide**, **repeat**, or **jump** to different parts of the code depending on certain conditions.

There are mainly **three types of control statements** in C:

A) Decision Control Statements: These statements allow the program to **make decisions** and execute different code depending on the condition.

- **`if` statement:** Executes a block of code if a condition is true. E.g.,
```
if (a > b) {
   printf("a is greater");
}
```
- **`if-else` statement:** Executes one block if the condition is true, otherwise executes another block. E.g.,
```
if (a > b) {
   printf("a is greater");
} else {
   printf("b is greater");
}
```
- **nested-if statement:** If contains another if is called nested-if.
- **`else if` ladder:** Used when multiple conditions need to be checked. E.g.,
```
if (a > b) {
   printf("a is greater");
} else if (a == b) {
   printf("both are equal");
} else {
   printf("b is greater");
}
```
- **`switch` statement:** Used when you have multiple values to compare with the same variable. E.g.,
```
switch (ch) {
   case 1: printf("One"); break;
   case 2: printf("Two"); break;
   default: printf("Invalid");
}
```

B) Loop Control Statements: These statements are used to **repeat a block of code** multiple times.

- **`for` loop:** Used when the number of iterations is known.
- **`while` loop:** Used when the number of iterations is not fixed and depends on a condition.
- **`do-while` loop:** Executes the block at least once, then checks the condition.

C) Jump Control Statements: These are used to **change the normal flow** of program execution.

- **`break` statement:** Exits from the loop or `switch` immediately.
- **`continue` statement:** Skips the current iteration and moves to the next iteration of the loop.
- **`goto` statement:** Transfers control to a labeled statement. (Use carefully!)

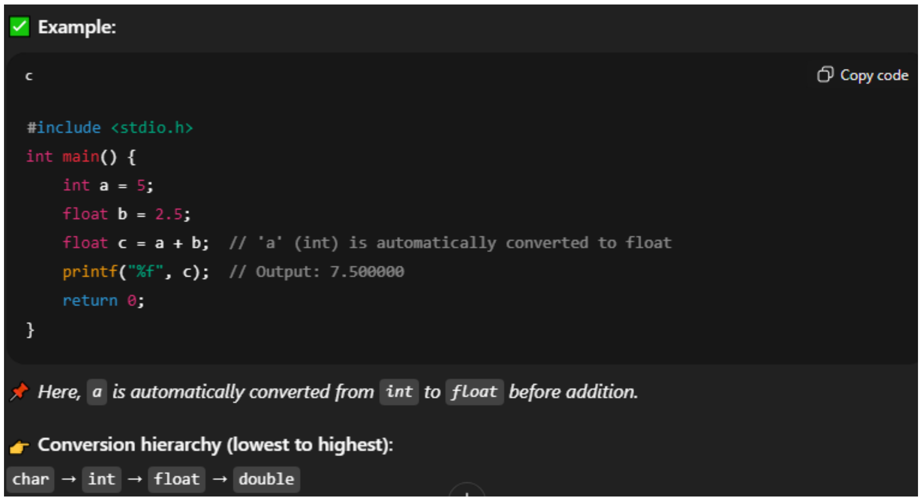
Ques 10. What is type conversion in C?

Ans. In **C**, **type conversion** means **changing the data type of a variable or value from one type to another**. For example, converting an `int` value into a `float` value or vice versa.

There are **two types of type conversion** in C—

A) Implicit Type Conversion (Type Promotion): Implicit Type Conversion a**lso** called **automatic type conversion**.

- It happens **automatically by the compiler** when two different data types are used in an expression.
- The compiler **converts the smaller data type to the larger data type** to avoid data loss.
- No special syntax is required.


✅ **Example:**

c

Copy code

```c
#include <stdio.h>
int main() {
    int a = 5;
    float b = 2.5;
    float c = a + b;  // 'a' (int) is automatically converted to float
    printf("%f", c);  // Output: 7.500000
    return 0;
}
```

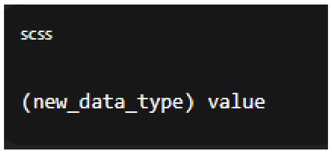
📌 *Here, `a` is automatically converted from `int` to `float` before addition.*

👉 **Conversion hierarchy (lowest to highest):**

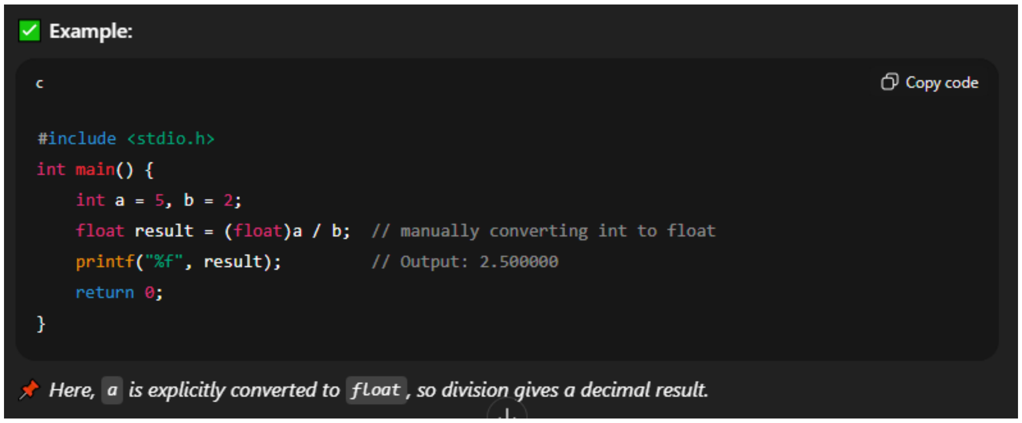
`char` → `int` → `float` → `double`

B) Explicit Type Conversion (Type Casting): It is also called **manual type conversion**.

- It happens **when the programmer forces the conversion** from one type to another.
- Explicit conversion is done **manually** by the programmer.
- Converting a larger data type to a smaller one can cause **data loss** (e.g., `float` to `int`).
- This is done using **casting operator**:


scss

```
(new_data_type) value
```


✅ **Example:**

c

Copy code

```c
#include <stdio.h>
int main() {
    int a = 5, b = 2;
    float result = (float)a / b;  // manually converting int to float
    printf("%f", result);         // Output: 2.500000
    return 0;
}
```

📌 *Here, `a` is explicitly converted to `float`, so division gives a decimal result.*