Shakuntala Krishna Institute Of Technology KD - 64

Subject : - C Programming (BCA 1st)

Top – 15 Questions

1. What are operators in C? Explain different types of operators with examples.
2. Explain operator precedence and associativity in C with an example program.
3. Explain different types of control structures in C with syntax and examples.
4. What is an array? Explain declaration, initialization, and processing of one dimensional array with examples.
5. Explain two-dimensional arrays with example programs (matrix addition/multiplication).
6. What is a string? Explain different string operations with examples.
7. Explain the concept of functions in C. Write the advantages of functions with examples.
8. Differentiate between actual and formal arguments with suitable examples.
9. Explain bubble sort and selection sort with proper algorithms and example programs.
10. Explain insertion sort and quick sort algorithms with examples.
11. What is merge sort? Explain its working principle with an example.
12. Explain linear search and binary search with algorithms and example programs.
13. What is a structure in C? Explain its declaration, initialization, and accessing of members with examples.
14. What are pointers? Explain pointer declaration, initialization, and pointer arithmetic with examples.
15. Write short notes on:
    - Dynamic Memory Allocation in C

Answer No. – 1

Operators:- An **operator** is a symbol that tells the compiler to perform specific mathematical or logical functions. By definition, an **operator** performs a certain operation on operands. An operator needs one or more operands for the operation to be performed.

Depending on how many operands are required to perform the operation, operands are called as unary, binary or ternary operators. They need one, two or three operands respectively.

- **Unary operators** − ++ (increment), -- (decrement), ! (NOT), ~ (compliment), & (address of), * (dereference)
- **Binary operators** − arithmetic, logical and relational operators except !
- **Ternary operators** − The ? operator

C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment/Decrement Operators
- Ternary Operators

We will, in this chapter, look into the way each operator works. Here, you will get an overview of all these chapters. Thereafter, we have provided independent chapters on each of these operators that contain plenty of examples to show how these operators work in C Programming.

Arithmetic Operators

We are most familiar with the arithmetic operators. These operators are used to perform arithmetic operations on operands. The most common arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/).

In addition, the modulo (%) is an important arithmetic operator that computes the remainder of a division operation. Arithmetic operators are used in forming an arithmetic expression. These

operators are binary in nature in the sense they need two operands, and they operate on numeric operands, which may be numeric literals, variables or expressions.

For example, take a look at this simple expression −

a + b

Here "+" is an arithmetic operator. We shall learn more about arithmetic operators in C in a subsequent chapter.

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20 then −

Show Examples

Operator	**Description**	**Example**
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Program

```c
#include <stdio.h>

int main(){

   int op1 = 10;
   int op2 = 3;

   printf("Operand1: %d Operand2: %d \n\n", op1, op2);
   printf("Addition of op1 and op2: %d\n", op1 + op2);
   printf("Subtraction of op2 from op1: %d\n", op1 - op2);
   printf("Multiplication of op1 and op2: %d\n", op1 * op2);
   printf("Division of op1 by op2: %d\n", op1/op2);

   return 0;
}
```

Relational Operators

We are also acquainted with relational operators while learning secondary mathematics. These operators are used to compare two operands and return a boolean value (true or false). They are used in a boolean expression.

The most common relational operators are less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equal to (==), and not equal to (!=). Relational operators are also binary operators, needing two numeric operands.

For example, in the Boolean expression −

a > b

Here, ">" is a relational operator.

We shall learn more about with relational operators and their usage in one of the following chapters.

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.

<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Program

```
#include <stdio.h>

int main(){

   int op1 = 5;
   int op2 = 3;

   printf("op1: %d op2: %d op1 < op2: %d\n", op1, op2, op1 < op2);

   return 0;
}
```

Logical Operators

These operators are used to combine two or more boolean expressions. We can form a compound Boolean expression by combining Boolean expression with these operators. An example of logical operator is as follows −

a >= 50 && b >= 50

The most common logical operators are AND (&&), OR(||), and NOT (!). Logical operators are also binary operators.

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.

!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

We will discuss more about Logical Operators in C in a subsequent chapter.

Program

```
#include <stdio.h>
 int main(){
 int a = 5;
int b = 20;
if (a && b){
 printf("Line 1 - Condition is true\n" );
 }
if (a || b){
printf("Line 2 - Condition is true\n" );
} /* lets change the value of a and b */
 a = 0;
 b = 10;
if (a && b){
printf("Line 3 - Condition is true\n" );
 }
else {
printf("Line 3 - Condition is not true\n" );
}
if (!(a && b)){
 printf("Line 4 - Condition is true\n" ); } return 0;
 }
```

Bitwise Operators

Bitwise operators let you manipulate data stored in computers memory. These operators are used to perform bit-level operations on operands.

The most common bitwise operators are AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>). Here the "~" operator is a unary operator, while most of the other bitwise operators are binary in narure.

Bitwise operator works on bits and perform bit−by−bit operation. The truth tables for &, "|", and "^" are as follows −

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Show Examples

Operator	**Description**	**Example**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e.,

		0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e,. - 0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Program

```c
#include <stdio.h>

int main(){

 unsigned int a = 60; /* 60 = 0011 1100 */

 unsigned int b = 13; /* 13 = 0000 1101 */

int c = 0; c = a & b; /* 12 = 0000 1100 */

printf("Line 1 - Value of c is %d\n", c );

 c = a | b; /* 61 = 0011 1101 */

printf("Line 2 - Value of c is %d\n", c ); c = a ^ b; /* 49 =
0011 0001 */

 printf("Line 3 - Value of c is %d\n", c ); c = ~a; /*-61 =
1100 0011 */

printf("Line 4 - Value of c is %d\n", c ); c = a << 2; /* 240 =
1111 0000 */

printf("Line 5 - Value of c is %d\n", c ); c = a >> 2; /* 15 =
0000 1111 */

printf("Line 6 - Value of c is %d\n", c );

return 0;

}
```

Assignment Operators

As the name suggests, an assignment operator "assigns" or sets a value to a named variable in C. These operators are used to assign values to variables. The "=" symbol is defined as assignment operator in C, however it is not to be confused with its usage in mathematics.

The following table lists the assignment operators supported by the C language –

Show Examples

Operator	Description	Example

=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Hence, the expression "a = 5" assigns 5 to the variable "a", but "5 = a" is an invalid expression in C.

The "=" operator, combined with the other arithmetic, relational and bitwise operators form augmented assignment operators. For example, the += operator is used as add and assign operator. The most common assignment operators are =, +=, -=, *=, /=, %=, &=, |=, and ^=.

Increment and Decrement Operators

The increment operator (++) increments the value of a variable by 1, while the decrement operator (--) decrements the value.

Increment and decrement operators are frequently used in the construction of counted loops in C (with the **for** loop). They also have their application in the traversal of array and pointer arithmetic.

The ++ and -- operators are unary and can be used as a prefix or posfix to a variable.

Example of Increment and Decrement Operators

The following example contains multiple statements demonstrating the use of increment and decrement operators with different variations –

```c
#include <stdio.h>

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

  a++; // postfix increment
  ++b; // prefix increment
  c--; // postfix decrement
  --d; // prefix decrement

  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d\n", c);
  printf("d = %d\n", d);

  return 0;
}
```

Ternary Operators

he ternary operator (?:) in C is a type of conditional operator. The term "ternary" implies that the operator has three operands. The ternary operator is often used to put multiple conditional (if-else) statements in a more compact manner.

Syntax of Ternary Operator in C

The ternary operator is used with the following syntax −

```
exp1 ? exp2 : exp3
```

It uses three **operands** −

- **exp1** − A Boolean expression evaluating to true or false
- **exp2** − Returned by the ? operator when exp1 is true
- **exp3** − Returned by the ? operator when exp1 is false

```c
#include <stdio.h>

int main(){

   int a = 10;

   (a % 2 == 0) ? printf("%d is Even \n", a) : printf("%d is Odd \n", a);

   return 0;
}
```

The sizeof Operator

The **sizeof operator** is a compile−time **unary operator**. It is used to compute the size of its operand, which may be a data type or a variable. It returns the size in number of bytes.

It can be applied to any data type, float type, or pointer type variables.

```
sizeof(type or var);
```

When sizeof() is used with a data type, it simply returns the amount of memory allocated to that data type. The outputs can be different on different machines, for example, a 32-bit system can show a different output as compared to a 64-bit system.When sizeof() is used with a data type, it simply returns the amount of memory allocated to that data type. The outputs can be different on different machines, for example, a 32-bit system can show a different output as compared to a 64-bit system.

```c
#include <stdio.h>

int main(){

   int a = 16;

   printf("Size of variable a: %d \n",sizeof(a));
   printf("Size of int data type: %d \n",sizeof(int));
   printf("Size of char data type: %d \n",sizeof(char));
   printf("Size of float data type: %d \n",sizeof(float));
   printf("Size of double data type: %d \n",sizeof(double));

   return 0;
}
```

Answer No. - 2

Operators Precedence/ Operator Associativity in C

A single expression in C may have multiple operators of different types. The C compiler evaluates its value based on the operator precedence and associativity of operators.

The precedence of operators determines the order in which they are evaluated in an expression. Operators with higher precedence are evaluated first.

For example, take a look at this expression –

x = 7 + 3 * 2;

Here, the multiplication operator "*" has a higher precedence than the addition operator "+". So, the multiplication 3*2 is performed first and then adds into 7, resulting in "x = 13".

The following table lists the order of precedence of operators in C. Here, operators with the highest precedence appear at the top of the table, and those with the lowest appear at the bottom.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left

Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Within an expression, higher precedence operators will be evaluated first.

Operator Associativity

In C, the associativity of operators refers to the direction (left to right or right to left) an expression is evaluated within a program. Operator associativity is used when two operators of the same precedence appear in an expression.

In the following example −

15 / 5 * 2

Both the "/" (division) and "*" (multiplication) operators have the same precedence, so the order of evaluation will be decided by associativity.

As per the above table, the associativity of the multiplicative operators is from Left to Right. So, the expression is evaluated as −

(15 / 5) * 2

It evaluates to −

3 * 2 = 6

Example 1

In the following code, the multiplication and division operators have higher precedence than the addition operator.

The left−to−right associativity of multiplicative operator results in multiplication of "**b**" and "**c**" divided by "**e**". The result then adds up to the value of "**a**".

```
#include <stdio.h>

int main(){


   int a = 20;

   int b = 10;

   int c = 15;

   int d = 5;


   int e;

   e = a + b * c / d;

   printf("e : %d\n" ,  e );


   return 0;

}
```

Output

e: 50

Example 2

We can use parenthesis to change the order of evaluation. Parenthesis () got the highest priority among all the C operators.

```
#include <stdio.h>

int main(){


  int a = 20;

  int b = 10;

  int c = 15;

  int d = 5;

  int e;


  e = (a + b) * c / d;

  printf("e:  %d\n",  e);


  return 0;

}
```

Output

```
e: 90
```

In this expression, the addition of a and b in parenthesis is first. The result is multiplied by c and then the division by d takes place.

Example 3

In the expression that calculates e, we have placed a+b in one parenthesis, and c/d in another, multiplying the result of the two.

```
#include <stdio.h>

int main(){
```

```
   int a = 20;

   int b = 10;

   int c = 15;

   int d = 5;


   int e;

   e = (a + b) * (c / d);

   printf("e: %d\n",  e );


   return 0;

}
```

Output

```
e: 90
```

Precedence of Post / Prefix Increment / Decrement Operators

The "++" and "− −" operators act as increment and decrement operators, respectively. They are unary in nature and can be used as a prefix or postfix to a variable.

When used as a standalone, using these operators in a prefix or post−fix manner has the same effect. In other words, "a++" has the same effect as "++a". However, when these "++" or "− −" operators appear along with other operators in an expression, they behave differently.

Postfix increment and decrement operators have higher precedence than prefix increment and decrement operators.

Example

The following example shows how you can use the increment and decrement operators in a C program −

```
#include <stdio.h>

int main(){

   int x = 5, y = 5, z;
   printf("x: %d \n", x);

   z = x++;
   printf("Postfix increment: x: %d z: %d\n", x, z);

   z = ++y;
   printf("Prefix increment. y: %d z: %d\n", y ,z);

   return 0;
}
```

Output

```
x: 5
Postfix increment: x: 6 z: 5
Prefix increment. y: 6 z: 6
```

Logical operators have left−to−right associativity. However, the compiler evaluates the least number of operands needed to determine the result of the expression. As a result, some operands of the expression may not be evaluated.

For example, take a look at the following expression −

```
x > 50 && y > 50
```

Answer No. - 3

Control Statements :-

Control statements in C language are instructions that determine the flow of a program's execution based on certain conditions or repetitions. They help decide whether a block of code should run, repeat, or skip. These statements make programs dynamic by adding decision-making, looping, and jumping capabilities, which are essential for solving real-world problems.

Let's understand it with a real-life example:

Think of control statements like traffic signals. A green light allows cars to move (execution), a red light stops them (condition not met), and a yellow light prepares them for the next action (transition).

Types of Control Statements in C

There are three types of control statements in C programming:

1. Decision-Making Statements: These statements allow the program to make decisions and execute specific blocks of code based on conditions.

2. Loop Control Statements: These statements repeatedly execute a block of code as long as a specified condition is true.

3. Jump Statements: These statements transfer the program's control from one part of the code to another, either conditionally or unconditionally.

Decision-Making Control Statements in C

Decision-making statements in C allow the program to make choices and execute specific blocks of code based on conditions. These statements enable a program to behave dynamically and handle different scenarios effectively.

Let's discuss the different types of decision control statements in C:

1. if Statement

The if statement in C executes a block of code only if the given condition is true.

Syntax:

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

Example:

```
#include <stdio.h>
int main() {
   int number = 10;
   if (number > 0) {
      printf("The number is positive.");
   }
   return 0;
}
```

When to Use: Use the if statement when you need to perform an action based on a single condition.

2. if-else Statement

The if-else statement in C executes one block of code if the condition is true and another block if it is false.

Syntax:

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

Example:

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

```
int main() {
    int number = -5;
    if (number >= 0) {
        printf("The number is non-negative.");
    } else {
        printf("The number is negative.");
    }
    return 0;
}
```

When to Use: Use if-else control statement in C programming when you need to perform one action for a true condition and another for a false condition.

3. nested if Statement

It is an if statement inside another if statement, used to check multiple conditions.

Syntax:

```
if (condition1) {
    if (condition2) {
        // Code if both conditions are true
    }
}
```

Example:

```
#include <stdio.h>
int main() {
    int number = 5;
    if (number > 0) {
```

```
        if (number % 2 == 0) {
            printf("The number is positive and even.");
        } else {
            printf("The number is positive and odd.");
        }
    }
    return 0;
}
```

When to Use: Use nested if when you need to test multiple related conditions.

4. if-else-if Ladder

This conditional statement in C checks multiple conditions one by one until a true condition is found.

Syntax:

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

Example:

```
#include <stdio.h>
int main() {
    int marks = 85;
```

```
    if (marks >= 90) {
        printf("Grade: A");
    } else if (marks >= 75) {
        printf("Grade: B");
    } else if (marks >= 50) {
        printf("Grade: C");
    } else {
        printf("Grade: F");
    }
    return 0;
}
```

When to Use: Use the if-else-if ladder when you need to check multiple conditions in sequence.

5. switch case Statement

The switch case in C tests a variable against multiple values (cases) and executes the matching block of code.

Syntax:

```
switch (variable) {
    case value1:
        // Code for case 1
        break;
    case value2:
        // Code for case 2
        break;
    default:
```

```
    // Code if no case matches
}
```

Example:

```
#include <stdio.h>
int main() {
   int day = 3;
   switch (day) {
      case 1:
         printf("Monday");
         break;
      case 2:
         printf("Tuesday");
         break;
      case 3:
         printf("Wednesday");
         break;
      default:
         printf("Invalid day");
   }
   return 0;
}
```

When to Use: Use switch when you need to test a variable against a fixed set of values.

Loop Control Statements in C

Loop control statements in C are used to repeatedly execute a block of code as long as a specific condition is true. They simplify repetitive tasks and help in writing concise programs.

Imagine you are filling bottles with water. You continue filling bottles one by one until you run out of empty bottles. Similarly, in programming, looping statements repeatedly perform tasks until a condition is met.

Let's understand the different loops in C programming:

1. for Loop

The for loop in C executes a block of code a specific number of times, controlled by an initialization, condition, and increment/decrement.

Syntax:

```
for (initialization; condition; increment/decrement) {
    // Code to execute in each iteration
}
```

Example:

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

When to Use: Use the for loop when the number of iterations is known beforehand.

2. while Loop

The while loop in C executes a block of code repeatedly as long as the specified condition is true.

Syntax:

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

Example:

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

When to Use: Use the while loop when the number of iterations is not known and depends on a condition.

3. do-while Loop

The do-while loop in C executes a block of code at least once and then continues to execute as long as the condition is true.

Syntax:

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

Example:

```
#include <stdio.h>
int main() {
```

```
    int i = 1;
    do {
        printf("Number: %d\n", i);
        i++;
    } while (i <= 5);
    return 0;
}
```

When to Use: Use the do-while loop when you want the code to execute at least once, regardless of the condition.

Comparison of Looping Control Statements in C

Loop Type	Condition Checked	Best Used For
for	Before the first iteration	Known number of iterations.
while	Before each iteration	Unknown iterations, based on a condition.
do-while	After the first iteration	Code must run at least once.

Jumping Control Statements in C

Jumping statements in C are used to transfer control of the program from one point to another. These statements enable conditional or unconditional jumps, allowing flexibility in program flow.

Real-Life Example:

Imagine reading a book:

- You can skip pages you don't need (continue).
- You can stop reading altogether (break).
- You can flip directly to a specific page (goto).

Jumping statements in C programming work similarly by moving control within the code.

There are four main jump control statements in C language:

1. break Statement

The break statement in C terminates the nearest enclosing loop or switch statement and transfers control to the next statement after it.

Syntax:

```
break;
```

Example:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            break;
        }
        printf("Number: %d\n", i);
    }
    return 0;
}
```

When to Use: Use break to exit a loop or switch prematurely when a specific condition is met.

2. continue Statement

The continue statement in C skips the current iteration of the loop and moves to the next iteration.

Syntax:

```
continue;
```

Example:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;
        }
        printf("Number: %d\n", i);
    }
    return 0;
}
```

When to Use: Use continue when you want to skip specific iterations of a loop without exiting it.

3. goto Statement

The goto statement in C transfers control to a labeled statement within the program.

Syntax:

```
goto label;
...
label:
    // Code to execute
```

Example:

```
#include <stdio.h>

int main() {
    int number = 3;
    if (number < 5) {
        goto small;
```

```
    }
    printf("Number is not small.\n");
    return 0;

    small:
    printf("Number is small.\n");
}
```

When to Use: Use goto sparingly, typically in cases like error handling, where other approaches may complicate the code.

4. return Statement

The return statement in C exits the current function and optionally returns a value to the calling function.

Syntax:

```
return; // Without value
return value; // With value
```

Example:

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 5);
    printf("Result: %d\n", result);
    return 0;
```

}

When to Use: Use return to exit a function and optionally pass a value back to the calling function.

Comparison of Jumping Statements

Jump Type	Purpose	Best Used For
break	Exit a loop or switch	To terminate loops/switch on a condition.
continue	Skip to the next iteration	To bypass specific iterations in a loop.
goto	Jump to a labeled statement	Rarely, for complex flow control or errors.
return	Exit a function	To end function execution and return a value.

Examples of Control Statements in C

Below are some examples of C language control statements:

1. Decision-Making: Checking Eligibility for a Driving License

Scenario: A person needs to be at least 18 years old to apply for a driving license.

```
#include <stdio.h>

int main() {
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);


   if (age >= 18) {
      printf("You are eligible for a driving license.\n");
   } else {
      printf("You are not eligible for a driving license.\n");
   }
```

```
    return 0;
}
```

Run Code

Output:

Enter your age: 24

You are eligible for a driving license.

2. Looping: Printing Multiplication Table

Scenario: Generate and display the multiplication table of a given number in C.

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", num, i, num * i);
    }
    return 0;
}
```

Run Code

Output:

Enter a number: 12

12 x 1 = 12

12 x 2 = 24

12 x 3 = 36

12 x 4 = 48

12 x 5 = 60

12 x 6 = 72

12 x 7 = 84

12 x 8 = 96

12 x 9 = 108

12 x 10 = 120

3. Combining Decision and Loop: Odd or Even Numbers in a Range

Scenario: Print all odd numbers in a given range.

```
#include <stdio.h>

int main() {
   int start, end;
   printf("Enter the start and end of the range: ");
   scanf("%d %d", &start, &end);


   for (int i = start; i <= end; i++) {
      if (i % 2 != 0) {
         printf("%d ", i);
      }
   }
   return 0;
}
```

Run Code

Output:

Enter the start and end of the range: 5 100

5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99

4. Nested Loops: Printing a Star Pattern

Scenario: Create a pyramid-like star pattern for user-defined rows.

```
#include <stdio.h>

int main() {

   int rows;

   printf("Enter the number of rows: ");

   scanf("%d", &rows);



   for (int i = 1; i <= rows; i++) {

      for (int j = 1; j <= i; j++) {

         printf("* ");

      }

      printf("\n");

   }

   return 0;

}
```

Run Code

Output:

Enter the number of rows: 7

```
*

* *

* * *

* * * *

* * * * *

* * * * * *

* * * * * * *
```

5. switch Statement: Simple Calculator

Scenario: Write a simple calculator program in C to perform addition, subtraction, multiplication, or division based on user input.

```
#include <stdio.h>

int main() {
    char operator;
    double num1, num2;
    printf("Enter an operator (+, -, *, /): ");
    scanf(" %c", &operator);
    printf("Enter two numbers: ");
    scanf("%lf %lf", &num1, &num2);

    switch (operator) {
        case '+':
            printf("Result: %.2lf\n", num1 + num2);
            break;
```

```
        case '-':
            printf("Result: %.2lf\n", num1 - num2);
            break;
        case '*':
            printf("Result: %.2lf\n", num1 * num2);
            break;
        case '/':
            if (num2 != 0) {
                printf("Result: %.2lf\n", num1 / num2);
            } else {
                printf("Error: Division by zero.\n");
            }
            break;
        default:
            printf("Invalid operator.\n");
    }
    return 0;
}
```

Run Code

Output:

```
Enter an operator (+, -, *, /): *
Enter two numbers: 2 5
Result: 10.00
```

Answer No. – 4

Array:-

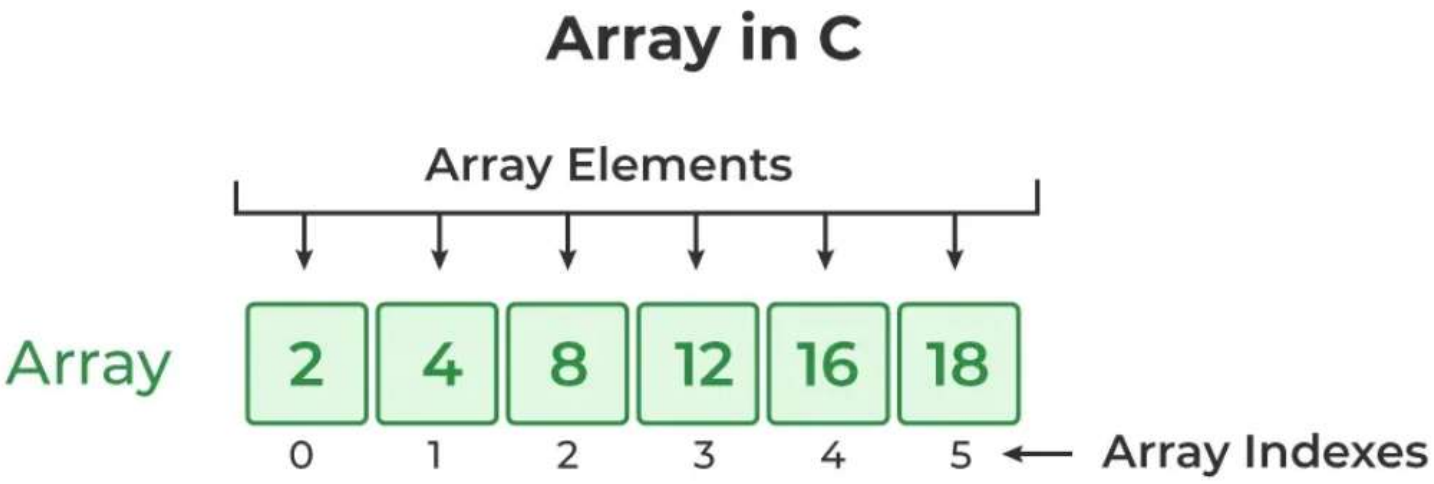
An **array** is a linear data structure that stores a fixed-size sequence of elements of the same data type in contiguous memory locations. Each element can be accessed directly using its index, which allows for efficient retrieval and modification.

```
#include <stdio.h>


int main() {


    int arr[] = {2, 4, 8, 12, 16, 18};

    int n = sizeof(arr)/sizeof(arr[0]);


    // Printing array elements

    for (int i = 0; i < n; i++) {

        printf("%d ", arr[i]);

    }


    return 0;

}
```

Output

2 4 8 12 16 18

The below image shows the array created in the above program.



To understand the key characteristics of arrays such as fixed size, contiguous memory allocation, and random access. Refer to this article: Properties of Arrays

Creating an Array

The whole process of creating an array can be divided into two primary sub processes i.e.

1. Array Declaration

Array declaration is the process of specifying the type, name, and size of the array. In C, we have to declare the array like any other variable before using it.

data_type array_name[size];

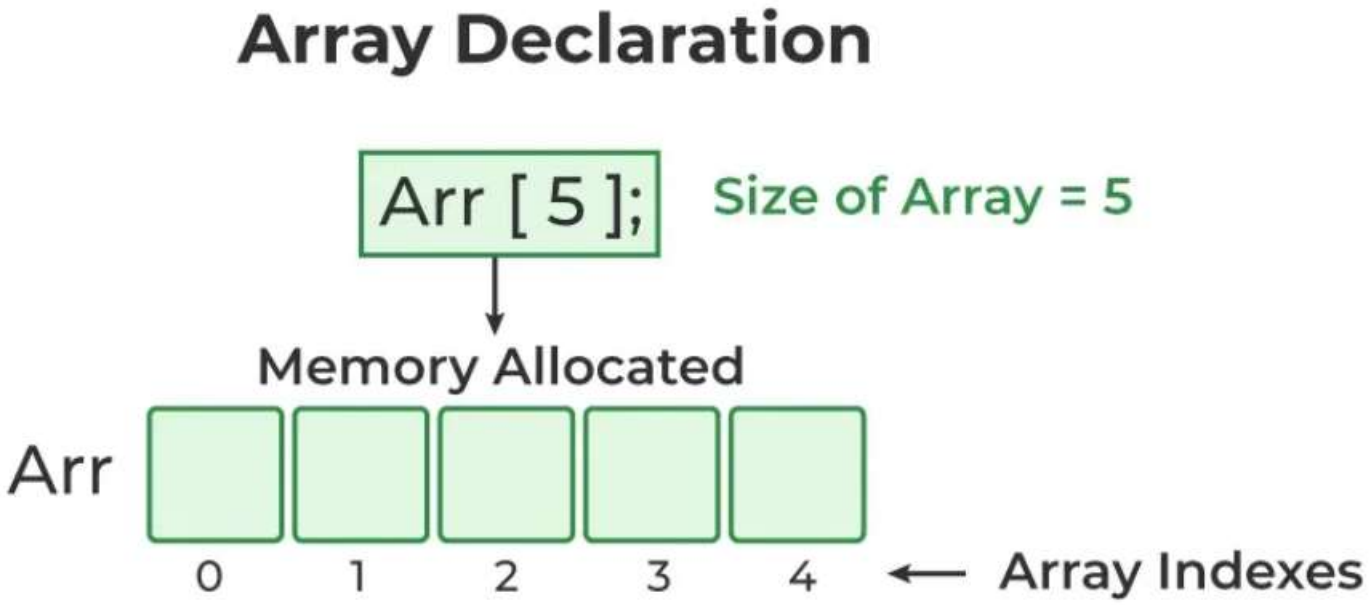
The above statements create an array with the name **array_name**, and it can store a specified number of elements of the same data type.

Example:

// Creates array arr to store 5 integer values.

int arr[5];

When we declare an array in C, the compiler allocates the memory block of the specified size to the array name.



2. Array Initialization

Initialization in C is the process to assign some initial value to the variable. When the array is declared or allocated memory, the elements of the array contain some garbage value. So, we need to initialize the array to some meaningful values.

Syntax:

int arr[5] = {2, 4, 8, 12, 16};

The above statement creates an array **arr** and assigns the values **{2, 4, 8, 12, 16}** at the time of declaration.

We can skip mentioning the size of the array if declaration and initialisation are done at the same time. This will create an array of size n where n is the number of elements defined during array initialisation. We can also **partially initialize** the array. In this case, the remaining elements will be assigned the value 0 (or equivalent according to the type).

//Partial Initialisation

int arr[5] = {2, 4, 8};

//Skiping the size of the array.

int arr[] = {2, 4, 8, 12, 16};

//initialize an array with all elements set to 0.

```
int arr[5] = {0};
```

Accessing Array Elements

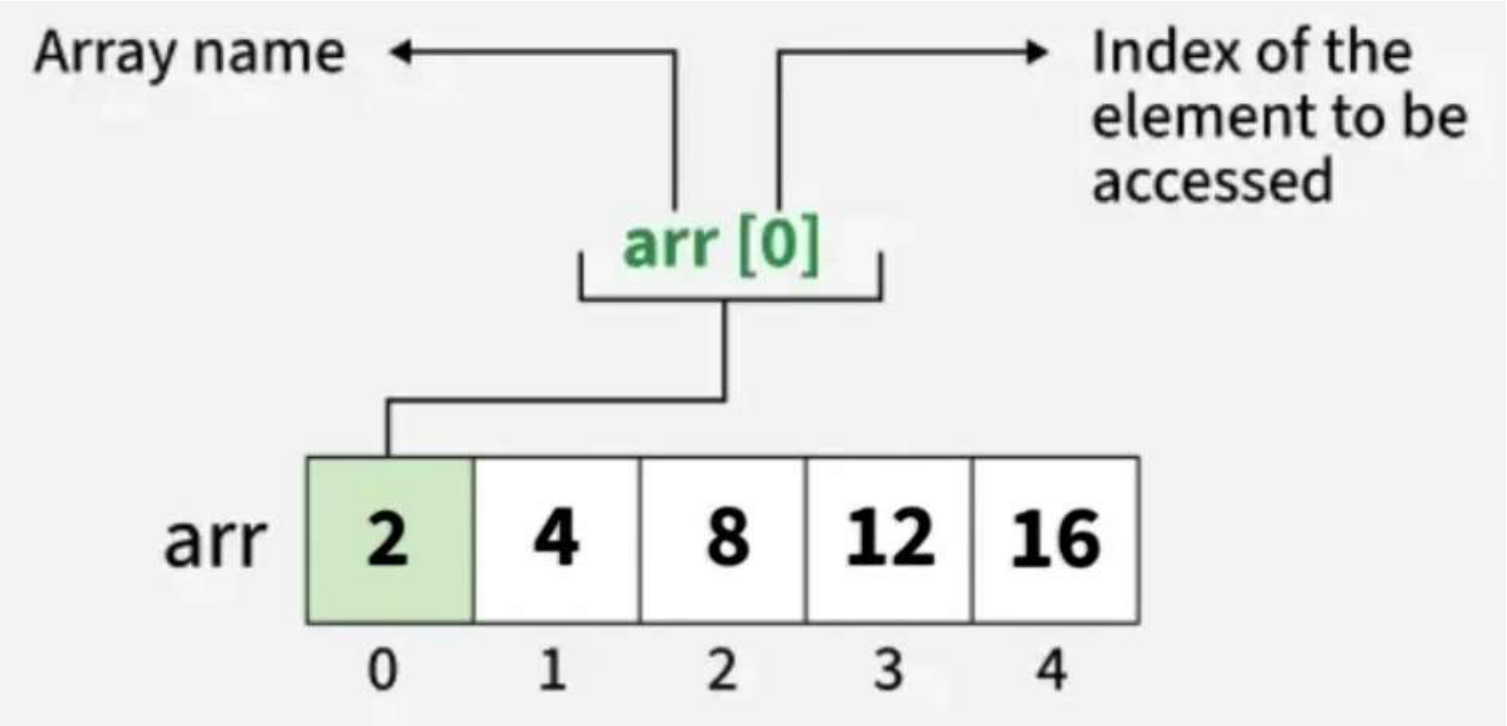
Array in C provides random access to its elements, which means that we can access any element of the array by providing the position of the element, called the index.

Syntax:

The index values start from **0** and goes up to **array_size-1**. We pass the index inside **square brackets []** with the name of the array.

```
array_name [index];
```

where, index value lies into this range - **(0 ≤ index ≤ size-1)**.



Example:

```
#include <stdio.h>


int main() {
```

```
    // array declaration and initialization
    int arr[5] = {2, 4, 8, 12, 16};

    // accessing element at index 2 i.e 3rd element
    printf("%d ", arr[2]);

    // accessing element at index 4 i.e last element
    printf("%d ", arr[4]);

    // accessing element at index 0 i.e first element
    printf("%d ", arr[0]);
    return 0;
}
```

Output

8 16 2

Answer No. – 5

2D Array:-

A two dimensional array in C is like a collection of data stored in rows and columns, similar to a table or grid. It allows you to organize related information in a structured format, making it easier to access specific elements using their row and column numbers.

For example, storing numbers in a 2D array means each number can be identified clearly by its position, such as "third row, second column."

Declaration of Two Dimensional Array in C

Syntax:

data_type array_name[rows][columns];

- **data_type:** Specifies the type of elements the array will store (like int, float, or char).
- **array_name:** The name given to the array.
- **[rows] and [columns]:** Define the size of the array, specifying how many rows and columns it contains.

Example:

int matrix[3][4];

This example declares a two-dimensional array named matrix of type int with 3 rows and 4 columns. In total, this array can hold 3 x 4 = 12 integer values.

Initialization of 2D Array in C Programming

Initialization of two-dimensional arrays in C can be done in two main ways:

1. Initialization at Declaration:

You can initialize a 2D array at the time you declare it using braces {}.

Syntax:

```
data_type array_name[rows][columns] = {
   {val1, val2, val3},
   {val4, val5, val6}
};
```

Example:

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

Addition

```
#include <stdio.h>


int main() {
   int m, n;
   printf("Enter number of rows and columns: ");
   scanf("%d %d", &m, &n);

   int A[m][n], B[m][n], Sum[m][n];

   // Input matrix A
   printf("Enter elements of matrix A:\n");
   for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
         scanf("%d", &A[i][j]);
      }
   }

   // Input matrix B
   printf("Enter elements of matrix B:\n");
   for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
         scanf("%d", &B[i][j]);
      }
```

```
    }

    // Addition
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            Sum[i][j] = A[i][j] + B[i][j];
        }
    }

    // Output result
    printf("Resultant Matrix after Addition:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d\t", Sum[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Example Output:

```
Enter number of rows and columns: 2 2
Enter elements of matrix A:
1 2
```

3 4

Enter elements of matrix B:

5 6

7 8

Resultant Matrix after Addition:

6 8

10 12

Multiplication:

```
#include <stdio.h>


int main() {
    int m, n, p;
    printf("Enter rows and columns of matrix A: ");
    scanf("%d %d", &m, &n);
    printf("Enter columns of matrix B: ");
    scanf("%d", &p);


    int A[m][n], B[n][p], C[m][p];


    // Input matrix A
    printf("Enter elements of matrix A:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
```

```
        scanf("%d", &A[i][j]);
    }
}

// Input matrix B
printf("Enter elements of matrix B:\n");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < p; j++) {
        scanf("%d", &B[i][j]);
    }
}

// Initialize result matrix with 0
for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
        C[i][j] = 0;
    }
}

// Multiplication
for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
```

```
            }
        }
    }

    // Output result
    printf("Resultant Matrix after Multiplication:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            printf("%d\t", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Example Output:

```
Enter rows and columns of matrix A: 2 2
Enter columns of matrix B: 2
Enter elements of matrix A:
1 2
3 4
Enter elements of matrix B:
5 6
7 8
```

Resultant Matrix after Multiplication:

19 22

43 50

Answer No. – 6

String

C language provides various built-in functions that can be used for various operations and manipulations on strings. These string functions make it easier to perform tasks such as string copy, concatenation, comparison, length, etc. The **<string.h>** header file contains these string functions.

strlen()

The **strlen()** function is used to find the length of a string. It returns the number of characters in a string, excluding the null terminator ('\0').

Example

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

int main() {
    char s[] = "Gfg";

    // Finding and printing length of string s
    printf("%lu", strlen(s));
    return 0;
}
```

Output

strcpy()

The **strcpy()** function copies a string from the source to the destination. It copies the entire string, including the null terminator.

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

    char src[] = "Hello";

    char dest[20];


    // Copies "Hello" to dest

    strcpy(dest, src);

    printf("%s", dest);

    return 0;

}
```

Output

Hello

strncpy()

The **strncpy()** function is similar to strcpy(), but it copies at most n bytes from source to destination string. If source is shorter than n, strncpy() adds a null character to destination to ensure n characters are written.

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

   char src[] = "Hello";

   char dest[20];


   // Copies "Hello" to dest

   strncpy(dest, src, 4);

   printf("%s", dest);

   return 0;

}
```

Output

```
Hell
```

strcat()

The **strcat()** function is used to concatenate (append) one string to the end of another. It appends the source string to the destination string, replacing the null terminator of the destination with the source string’s content.

Example

```
#include <stdio.h>

#include <string.h>


int main() {

   char s1[30] = "Hello, ";
```

```
    char s2[] = "Geeks!";

    // Appends "Geeks!" to "Hello, "
    strcat(s1, s2);
    printf("%s", s1);
    return 0;
}
```

Output

```
Hello, Geeks!
```

strncat()

In C, there is a function **strncat()** similar to strcat(). This function appends not more than n characters from the string pointed to by source to the end of the string pointed to by destination plus a terminating NULL character.

Example:

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

int main() {
    char s1[30] = "Hello, ";
    char s2[] = "Geeks!";

    // Appends "Geeks!" to "Hello, "
    strncat(s1, s2, 4);
    printf("%s", s1);
```

```
    return 0;
}
```

Output

```
Hello, Geek
```

strcmp()

The **strcmp()** is a built-in library function in C. This function takes two strings as arguments, compares these two strings lexicographically and returns an integer value as a result of comparison.

Example

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

int main() {
    char s1[] = "Apple";
    char s2[] = "Applet";

    // Compare two strings
    // and print result
    int res = strcmp(s1, s2);
    if (res == 0)
        printf("s1 and s2 are same");
    else if (res < 0)
        printf("s1 is lexicographically "
               "smaller than s2");
    else
```

```
    printf("s1 is lexicographically "
           "greater than s2");
  return 0;
}
```

Output

s1 is lexicographically smaller than s2

strncmp()

This function lexicographically compares the first n characters from the two null-terminated strings and returns an integer based on the outcome.

Example:

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

int main() {
  char s1[] = "Apple";
  char s2[] = "Applet";

  // Compare two strings upto
  // 4 characters and print result
  int res = strncmp(s1, s2, 4);
  if (res == 0)
    printf("s1 and s2 are same");
  else if (res < 0)
    printf("s1 is lexicographically "
```

```
            "smaller than s2");
    else
        printf("s1 is lexicographically "
        "greater than s2");
    return 0;
}
```

Output

```
s1 and s2 are same
```

strchr()

The **strchr()** function is used to find the first occurrence of a given character in a string. If the character is found, it returns a pointer to the first occurrence of the character; otherwise, it returns NULL.

Example:

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

int main() {
    char s[] = "Hello, World!";

    // Finding the first occurence of 'o' in string s
    char *res = strchr(s, 'o');
    if (res != NULL)
        printf("Character found at: %ld index", res - s);
    else
```

```
    printf("Character not found");
  return 0;
}
```

Output

Character found at: 4 index

strrchr()

In C, **strrchr()** function is similar to strchr() function used to find the last occurrence of a given character in a string.

Example:

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

int main() {
  char s[] = "Hello, World!";

  // Finding the last occurence of 'o' is string s
  char *res = strrchr(s, 'o');

  if (res != NULL)
    printf("Character found at: %ld index", res - s);
  else
    printf("Character not found\n");
  return 0;
}
```

Output

Character found at: 8 index

strstr()

The **strstr(**) function in C is used to search the first occurrence of a substring in another string. If it is not found, it returns a NULL.

Example:

```
#include <stdio.h>

#include <string.h>


int main() {

   char s[] = "Hello, Geeks!";


   // Find the occurence of "Geeks" in string s

   char *pos = strstr(s, "Geeks");


   if (pos != NULL)

      printf("Found");

   else

      printf("Not Found");

   return 0;

}
```

Output

Found

sprintf()

The **sprintf()** function is used to format a string and store it in a buffer. It is similar to printf(), but instead of printing the result, it stores it in a string.

Example:

```
#include <stdio.h>


int main() {
    char s[50];
    int n = 10;


    // Output formatted string into string bugger s
    sprintf(s, "The value is %d", n);
    printf("%s", s);
    return 0;
}
```

Output

The value is 10

strtok()

The **strtok()** function is used to split a string into tokens based on specified delimiters. It modifies the original string by replacing delimiters with null characters ('\0').

Example:

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

```
int main() {
    char s[] = "Hello, Geeks, C!";

    // Initializing tokens
    char *t = strtok(s, ", ");

    // Printing rest of the tokens
    while (t != NULL) {
        printf("%s\n", t);
        t = strtok(NULL, ", ");
    }
    return 0;
}
```

Output

```
Hello
Geeks
C!
```

Function	Description	Syntax
strlen()	Find the length of a string excluding '\0' NULL character.	**strlen**(str);
strcpy()	Copies a string from the source to the destination.	**strcpy**(dest, src);

Function	Description	Syntax
strncpy()	Copies n characters from source to the destination.	**strncpy**(dest, src, n);
strcat()	Concatenate one string to the end of another.	**strcat**(dest, src);
strncat()	Concatenate n characters from the string pointed to by src to the end of the string pointed to by dest.	**strncat**(dest, src, n);
strcmp()	Compares these two strings lexicographically.	**strcmp**(s1, s2);
strncmp()	Compares first n characters from the two strings lexicographically.	**strncmp**(s1, s2, n);
strchr()	Find the first occurrence of a character in a string.	**strchr**(s, c);
strrchr()	Find the last occurrence of a character in a string.	**strchr**(*s*, *ch*);
strstr()	First occurrence of a substring in another string.	**strstr**(s, subS);
sprintf()	Format a string and store it in a string buffer.	**sprintf**(s, format, ...);
strtok()	Split a string into tokens based on specified delimiters.	**strtok**(s, delim);

Answer No. – 7

Functions:

A function is a named block of code that performs a specific task. It allows you to write a piece of logic once and reuse it wherever needed in the program. This helps keep your code clean, organized, and easier to understand.

Functions play a vital role in building modular programs. They allow you to break down complex problems into smaller, manageable parts.

```
#include <stdio.h>


// Void function definition
void hello() {
    printf("GeeksforGeeks\n");
}


// Return-type function definition
int square(int x) {
    return x * x;
}


int main() {

    // Calling the void function
    hello();


    // Calling the return-type function
```

```
    int result = square(5);

    printf("Square of 5 is: %d", result);


    return 0;
}
```

Output

```
GeeksforGeeks

Square of 5 is: 25
```

In the above example, there are three functions:

- **main() function:** This is the starting point of every C program. When the program runs, execution begins from the main function.
- **hello() function:** This is a user-defined function that does not take any input and does not return a value. Its purpose is to print "GeeksforGeeks" to the screen. It is called inside the main function using hello();.
- **square() function:** This is another user-defined function, but unlike hello(), it has a return type. It takes one integer as input and returns the square of that number. In main(), we call square(5) and store the returned result in a variable to print it.

How Functions Work in C?

Function Syntax

Here is the basic structure:

```
return_type function_name(parameter_list) {
    // body of the function
}
```

Explanation of each part:

- **Return type**: Specifies the type of value the function will return. Use void if the function does not return anything.
- **Function name**: A unique name that identifies the function. It follows the same naming rules as variables.
- **Parameter list**: A set of input values passed to the function. If the function takes no inputs, this can be left empty or written as void.
- **Function body**: The block of code that runs when the function is called. It is enclosed in curly braces { }.

Function Declaration vs Definition

It's important to understand the difference between declaring a function and defining it. Both play different roles in how the compiler understands and uses your function.

Function Declaration

A declaration tells the compiler about the function's name, return type, and parameters before it is actually used. It does not contain the function's body. This is often placed at the top of the program or in a header file.

```
// function declaration

int add(int a, int b);
```

Function Definition

A definition provides the actual implementation of the function. It includes the full code or logic that runs when the function is called.

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

Why is declaration needed?

If a function is defined after the main function or another function that uses it, then a declaration is needed before it is called. This helps the compiler recognize the function and check for correct usage.

In short, the declaration introduces the function to the compiler, and the definition explains what it actually does.

Calling a Function

Once a function is defined, you can use it by simply calling its name followed by parentheses. This tells the program to execute the code inside that function.

```
#include <stdio.h>


// Function definition
int add(int a, int b) {
    return a + b;
}


int main() {


    // Function call
    int result = add(5, 3);
    printf("The sum is: %d", result);
    return 0;
}
```

Output

```
The sum is: 8
```

In this example, the function add is called with the values 5 and 3. The function runs its logic (adding the numbers) and returns the result, which is then stored in the variable result.

You can call a function as many times as needed from main or other functions. This helps avoid writing the same code multiple times and keeps your program clean and organized.

Types of Function in C

In C programming, functions can be grouped into two main categories: **library functions** and **user-defined functions**. Based on how they handle input and output, user-defined functions can be further classified into different types.

1. Library Functions: These are built-in functions provided by C, such as printf(), scanf(), sqrt(), and many others. You can use them by including the appropriate header file, like #include <stdio.h> or #include <math.h>.

2. User-Defined Functions: These are functions that you create yourself to perform specific tasks in your program. Depending on whether they take input or return a value, they can be of four types:

- **No arguments, no return value**: The function neither takes input nor returns any result.
- **Arguments, no return value:** The function takes input but does not return anything.
- **No arguments, return value:** The function does not take input but returns a result.
- **Arguments and return value:** The function takes input and returns a result.

Each type serves different purposes depending on what the program needs. Using the right type helps make your code more organized and efficient.

Memory Management of Functions

When a function is called, memory for its variables and other data is allocated in a separate block in a stack called a **stack frame.** The stack in which it is created is called **function call stack**. When the function completes its execution, its stack frame is deleted from the stack, freeing up the memory.

Refer to this article to know more Function Call Stack in C

Advantages of Functions:

- **Modularity:**

Functions promote modular programming, where a large program is divided into smaller, independent modules (functions). Each function handles a specific part of the overall task, making the code easier to understand, organize, and manage.

- **Example:** A program calculating student grades might have separate functions for calculateAverage(), assignGrade(), and printReport().

- **Code Reusability:**

Once a function is defined, it can be called multiple times from different parts of the program without rewriting the same code. This reduces redundancy and makes the code more efficient.

- **Example:** A square(int num) function can be called whenever the square of a number is needed, instead of writing num * num repeatedly.

C

```
int square(int num) {
    return num * num;
}

int main() {
    int x = 5;
    int y = 7;
    printf("Square of %d is %d\n", x, square(x));
    printf("Square of %d is %d\n", y, square(y));
    return 0;
}
```

- **Easier Debugging and Maintenance:**

When a program is divided into functions, isolating and fixing errors becomes simpler. If an issue arises, the problem can often be narrowed down to a specific function, rather than searching through the entire codebase. Similarly, modifying or updating a particular functionality only requires changes within its corresponding function.

- **Example:** If there's an error in calculating the average in a student grading system, the debugger can focus on the calculateAverage() function.

- **Improved Readability and Understandability:**

Functions make code more readable by providing meaningful names for blocks of code that perform specific actions. This enhances the clarity of the program's logic and makes it easier for other developers (or the original developer later on) to understand how the program works.

    - **Example:** A function named displayWelcomeMessage() clearly indicates its purpose, unlike a block of code without a descriptive name.

- **Reduced Code Size:**

By reusing functions, the overall size of the program can be significantly reduced, as the same code block is not duplicated throughout the program.

Answer No. – 8

Formal and Actual Parameters

What are Formal Parameters?

Formal parameters, also known as formal arguments, are placeholders defined in the function signature or declaration. They represent the data that the function expects to receive when called. Formal parameters serve as variables within the function's scope and are used to perform operations on the input data.

Syntax of Formal parameters:

Below is the syntax for Formal parameters:

```
// Here, 'name' is the formal parameter
function gfgFnc(name) {
    // Function body
}
```

What are Actual Parameters?

Actual parameters, also called actual arguments or arguments, are the values or expressions provided to a function or method when it is called. They correspond to the formal parameters in

the function's definition and supply the necessary input data for the function to execute. Actual parameters can be constants, variables, expressions, or even function calls.

Syntax of Actual parameters:

Below is the syntax for Actual parameters:

```
function gfgFnc(name) {
   // Function body
}


// Actual Parameter
gfgFnc("Geek");
```

Formal and Actual parameters in C:

Below is the implementation of Formal and Actual Parameters in C:

```
#include <stdio.h>
// a and b are formal parameters
int sum_numbers(int a, int b) { return a + b; }
int main()
{
   // 3 and 5 are actual parameters
   int result = sum_numbers(3, 5);
   printf("Sum: %d\n", result);
   return 0;
}
```

Output

```
Sum: 8
```

Answer No. – 9

Bubble sort and Selection sort

The task of arranging elements of an array in a particular order is referred to as **sorting**. The sorting of an array or a list is mainly done to make the searching easier. There are two types of sorting algorithms namely, **Bubble Sort** and **Selection Sort**.

Bubble sort performs sorting of data by exchanging the elements, while the selection sort performs sorting of data by selecting the elements.

Read this article to learn more about bubble sort and selection sort and how these two sorting techniques are different from each other.

What is Bubble Sort?

Bubble sort is a simple sorting algorithm. Bubble sort iterates through a list and compares adjacent pairs of elements to sort them. It swaps the elements of an array based on the adjacent elements.

The major advantage of bubble sort is that it is more efficient in comparison to selection sort. However, it is slower in comparison to selection sort.

Bubble sort uses item exchanging to swap elements. Therefore, the elements are repeatedly swapped in bubble sorting until all the elements are in the right order.

Following is the Bubble Sort Algorithm

```
begin BubbleSort(list)
  for all elements of list
    if list[i] > list[i+1]
          swap(list[i], list[i+1])
    end if
  end for
return list
end BubbleSort
```

What is Selection Sort?

Selection sort is a sorting algorithm which takes either the minimum value (ascending order) or the maximum value (descending order) in the list and places it at the proper position. The minimum or the maximum number from the list is obtained first. Next, it selects the minimum or maximum element from unsorted sub-array and puts it in the next position of the sorted sub-array, and so on. Selection sort is considered as an unstable sorting algorithm.

Selection sort algorithm is relatively more efficient in comparison to bubble sort. However, the number of comparisons made during iterations is more than the element swapping that is done. Also, in selection sort, the location of every element in a list is previously known. This means the user only searches for the element that needs to be inserted at a specific position. Selection sort is quick in comparison to bubble sort and it uses item selection for sorting of elements.

Following is the Selection Sort Algorithm

- **Step 1** ? Set MIN to location 0.
- **Step 2** ? Search the minimum element in the list.
- **Step 3** ? Swap with value at location MIN.
- **Step 4** ? Increment MIN to point to next element.
- **Step 5** ? Repeat until list is sorted.

Now, let us discuss the differences between bubble sort and selection sort in detail.

Difference between Bubble Sort and Selection Sort

The following are the important differences between bubble sort and selection sort ?

S.No.	Bubble Sort	Selection Sort
1.	Bubble sort is a simple sorting algorithm which continuously moves through the list and compares the adjacent pairs for proper sorting of the elements.	Selection sort is a sorting algorithm which takes either smallest value (ascending order) or largest value (descending order) in the list and place it at the proper position in the list.

2.	Bubble sort compares the adjacent elements and move accordingly.	Selection sort selects the smallest element from the unsorted list and moves it at the next position of the sorted list.
3.	Bubble sort performs a large number of swaps or moves to sort the list.	Selection sort performs comparatively less number of swaps or moves to sort the list.
4.	Bubble sort is relatively slower.	Selection sort is faster as compared to bubble sort.
5.	The efficiency of the bubble sort is less.	The efficiency of the selection sort is high.
6.	Bubble sort performs sorting of an array by exchanging elements.	Selection sort performs sorting of a list by the selection of element.

Answer No. - 10

Insertion sort and Quick sort

- **Insertion Sort:** This is a simple algorithm that builds a sorted array one element at a time by “inserting” each element into its correct position within the already sorted portion of the array. It is similar to how you might sort a hand of playing cards.
- **Quick Sort:** This is a more advanced algorithm that follows a “divide and conquer” approach. It selects a ‘pivot’ element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

If you need a refresher, here's a helpful link to the insertion sort article: Introduction to Insertion Sort Algorithm and quick sort article: Introduction to Quick Sort Algorithm.

Key Differences

Feature	Insertion Sort	Quick Sort
How it Works	Builds a sorted array by inserting elements into their correct positions.	Selects a 'pivot' element, partitions array around it, and recursively sorts the partitions.
Time Complexity	O(n) (best case), O(n^2) (average and worst case)	O(n log n) (average case), O(n^2) (worst case)
Space Complexity	O(1) (in-place)	O(log n) (average case), O(n) (worst case) due to recursion stack
Stability	Stable (maintains the relative order of equal elements)	Unstable (typically, unless a stable partitioning strategy is used)
Method	Iterative	Divide and Conquer, Recursive
Adaptability	Adaptive (performs well on partially sorted data)	Not adaptive (performance depends on pivot selection rather than existing order)
Implementation	Simple	More complex
Performance	Efficient for small or nearly sorted datasets	Generally more efficient for larger datasets

Answer No. - 11

Merge sort

Merge Sort is a comparison-based sorting algorithm that works by dividing the input array into two halves, then calling itself for these two halves, and finally it merges the two sorted halves. In this article, we will learn how to implement merge sort in C language.

What is Merge Sort Algorithm?

Merge sort is based on the three principles: divide, conquer and combine which is better implemented using recursion using two functions:

1. **mergeSort()** - For Divide
2. **merge()** - For Conquer and Combine

The **mergeSort()** function keeps dividing array into subarrays till it cannot be further divided (i.e. single element). Then **merge()** function is called to merge two subarrays at a time in the required order until we get back the whole array in the sorted order.

Implementation of Merge Sort in C

C language does not have inbuilt function for merge sort, so we have to manually implement it.

```
// C program for the implementation of merge sort

#include <stdio.h>

#include <stdlib.h>


// Merges two subarrays of arr[].

// First subarray is arr[left..mid]

// Second subarray is arr[mid+1..right]

void merge(int arr[], int left, int mid, int right) {

   int i, j, k;

   int n1 = mid - left + 1;
```

```
int n2 = right - mid;


// Create temporary arrays
int leftArr[n1], rightArr[n2];


// Copy data to temporary arrays
for (i = 0; i < n1; i++)
    leftArr[i] = arr[left + i];
for (j = 0; j < n2; j++)
    rightArr[j] = arr[mid + 1 + j];


// Merge the temporary arrays back into arr[left..right]
i = 0;
j = 0;
k = left;
while (i < n1 && j < n2) {
    if (leftArr[i] <= rightArr[j]) {
        arr[k] = leftArr[i];
        i++;
    }
    else {
        arr[k] = rightArr[j];
        j++;
    }
```

```
        k++;
    }

    // Copy the remaining elements of leftArr[], if any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    // Copy the remaining elements of rightArr[], if any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

// The subarray to be sorted is in the index range [left-right]
void mergeSort(int arr[], int left, int right) {
    if (left < right) {

        // Calculate the midpoint
        int mid = left + (right - left) / 2;
```

```
        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
        // Sorting arr using mergesort
    mergeSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Output

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

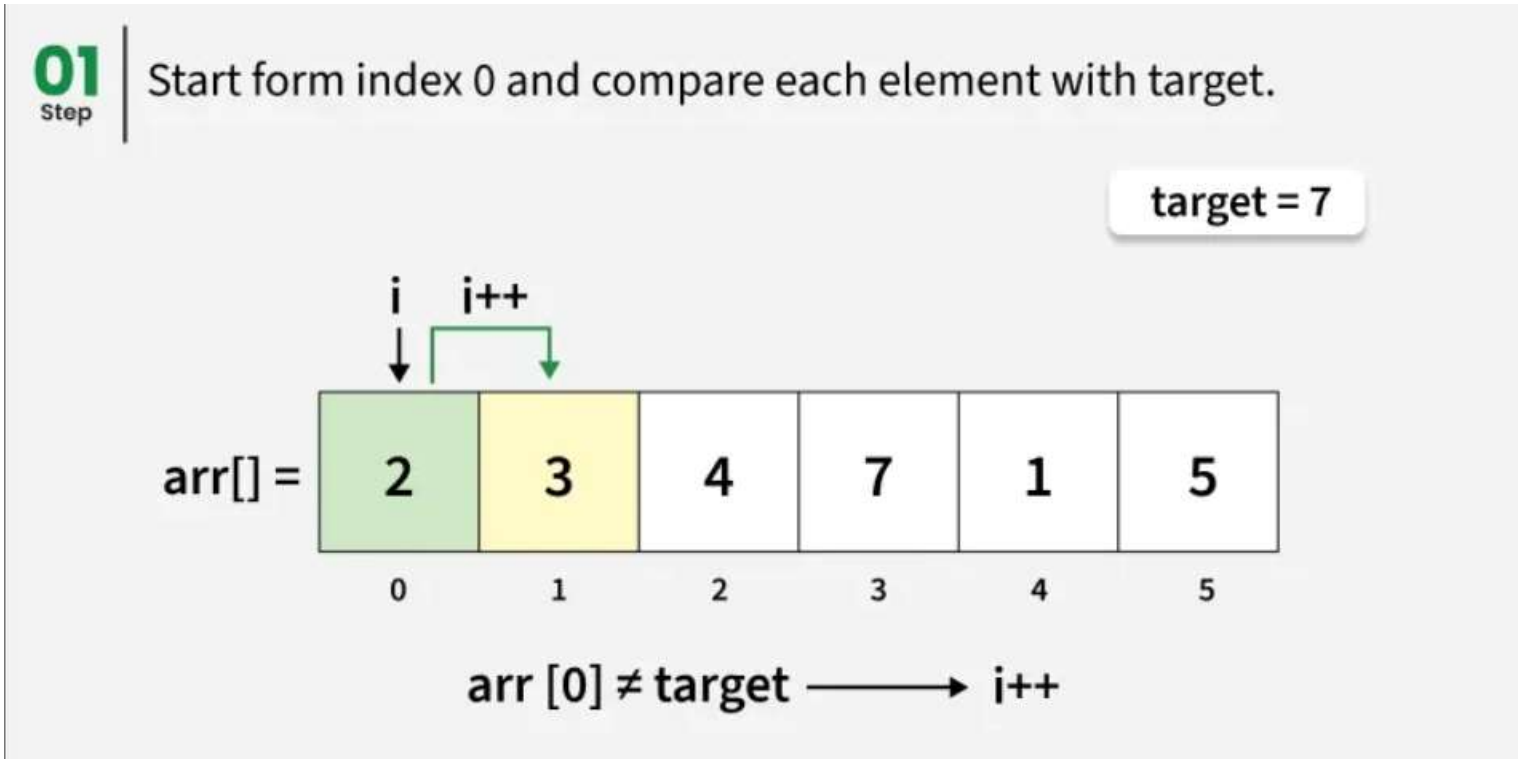
Answer No. – 12

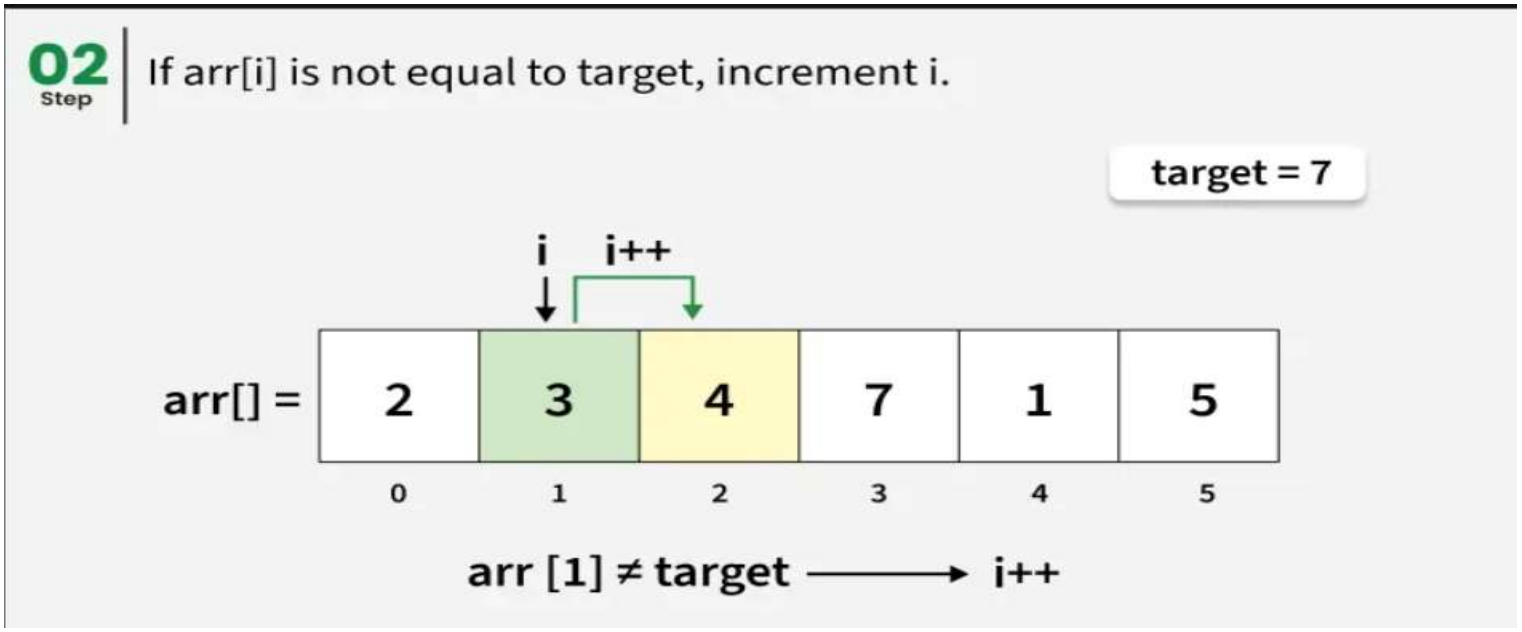
linear search and Binary search

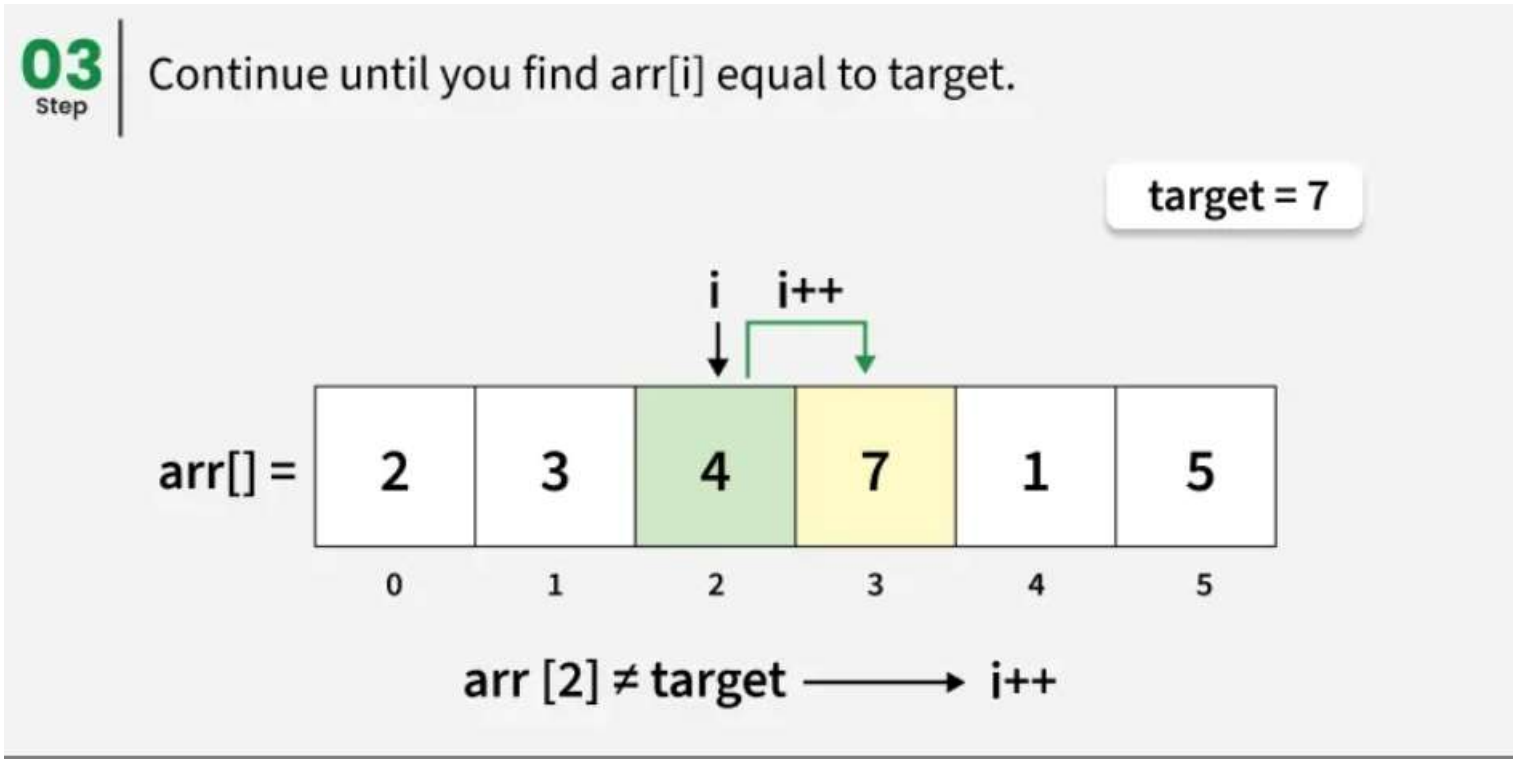
Linear Search	**Binary Search**
In linear search input data need not to be in sorted. It is also called sequential search.	In binary search input data need to be in sorted order. It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional array can be used. Linear search performs equality comparisons. It is less complex. It is very slow process.	Only single dimensional array is used. Binary search performs ordering comparisons. It is more complex. It is very fast process.

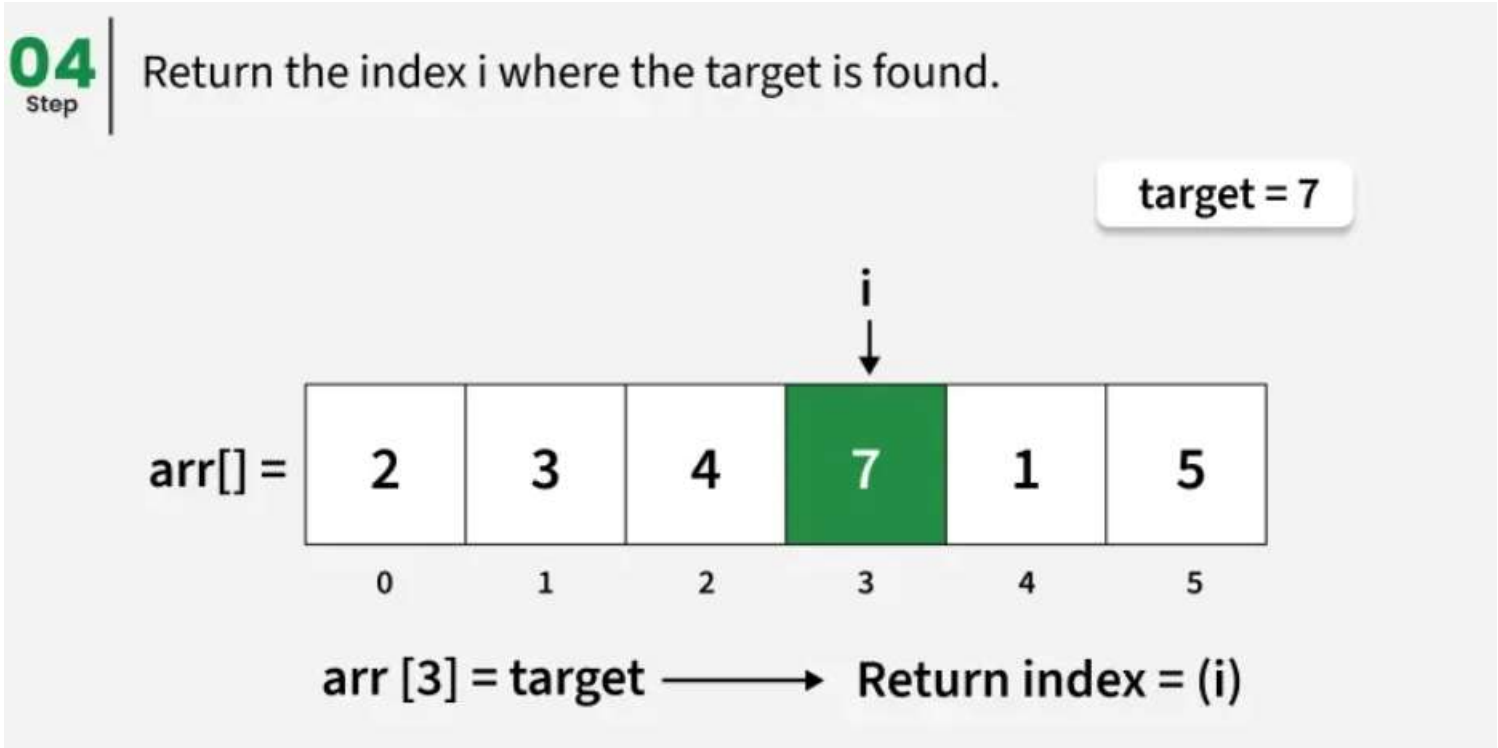
BINARY SEARCH

Binary Search is a more optimized form of searching algorithm. It cuts down the search space in halves achieving logarithmic time complexity on a sorted data. We take two extremes lower bound and upper bound and compare our target element with the middle element. In the process we discard one half where we are sure our target element can not be found and update our lower and upper bound accordingly.










```
#include <stdio.h>


int binarySearch(int arr[], int target, int low, int high) {
```

```
    // Repeat until the pointers low and
    // high meet each other
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {2, 3, 4, 7, 9,10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    int low = 0;
    int high = n;
    int index = binarySearch(arr, target, low, high);
    printf("%d\n", index);
```

```
    return 0;
}
```

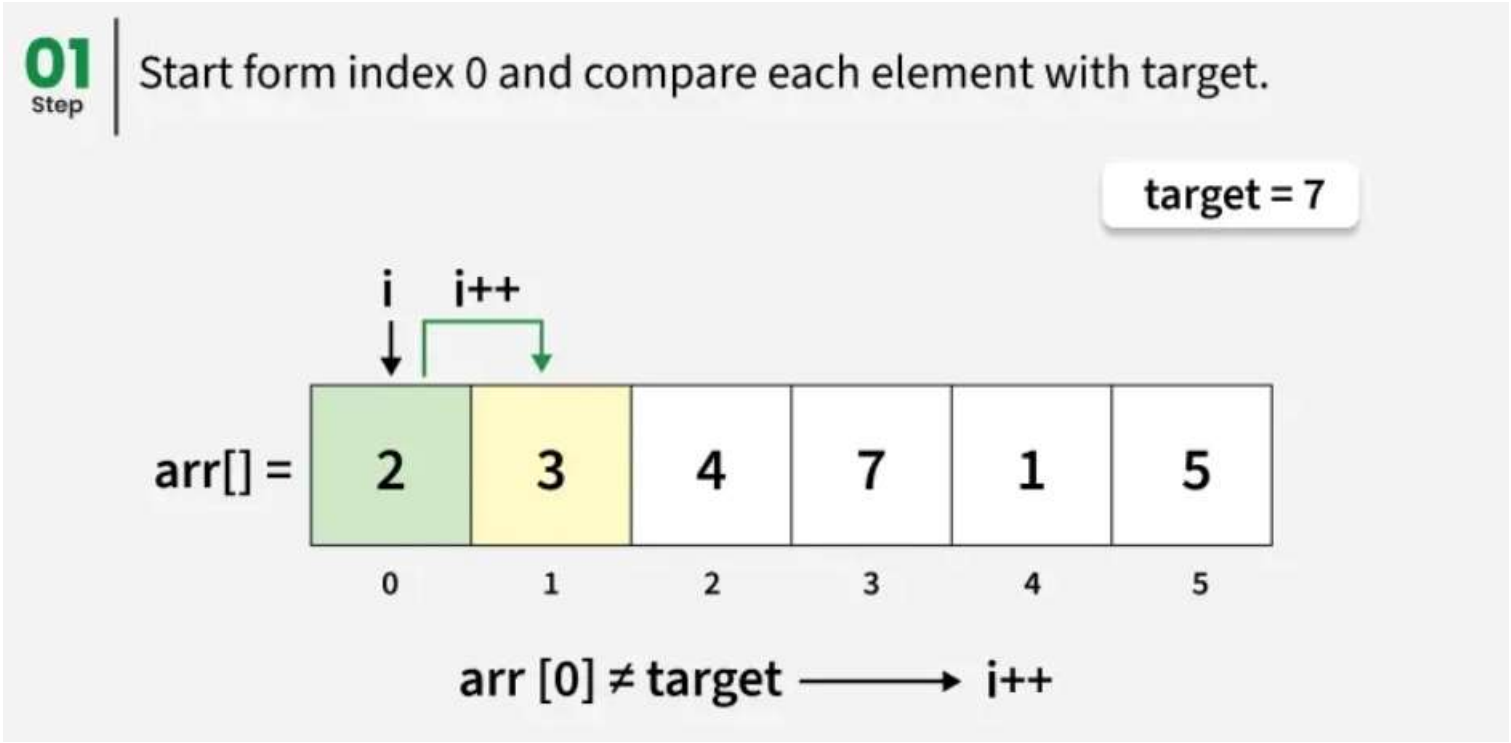
Output

3

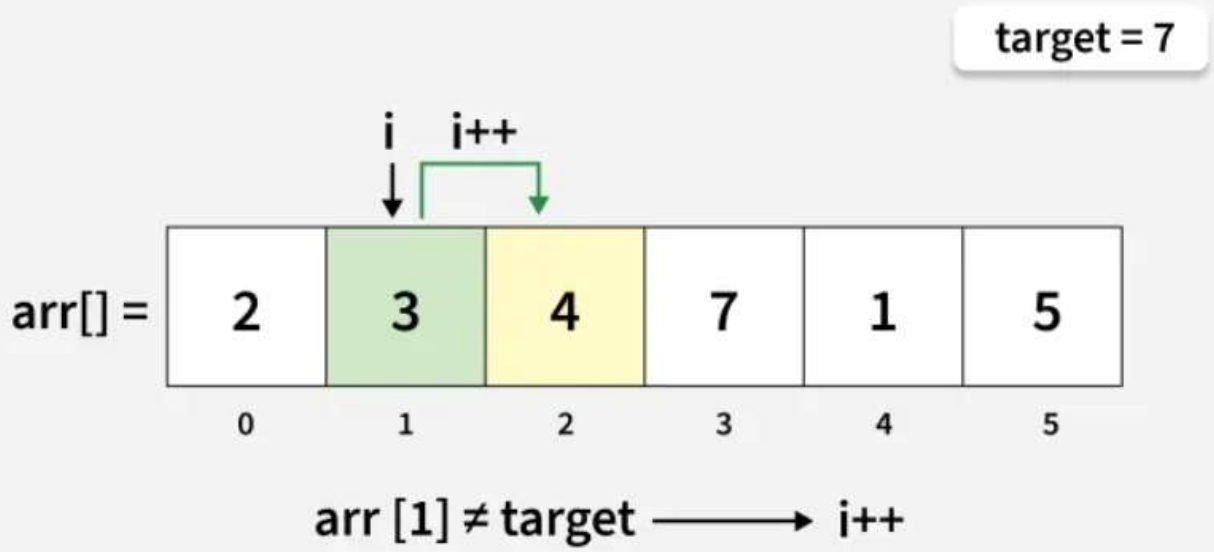
Time Complexity: O(log n) - Binary search algorithm divides the input array in half at every step, reducing the search space by half, and hence has a time complexity of logarithmic order.
Auxiliary Space: O(1) - Binary search algorithm requires only constant space for storing the low, high, and mid indices, and does not require any additional data structures, so its auxiliary space complexity is O(1).

LINEAR SEARCH

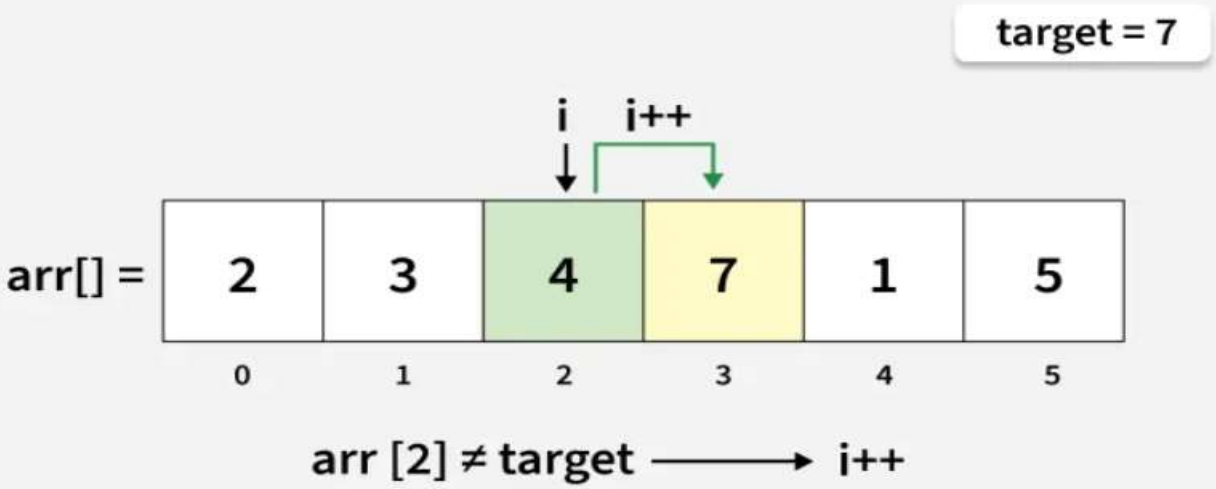
Suppose we are searching a target element in an array. In linear search we begin with the first position of the array, and traverse the whole array in order to find the target element. If we find the target element we return the index of the element. Otherwise, we will move to the next position. If we arrive at the last position of an array and still can not find the target, we return -1. This is called the Linear search or Sequential search.



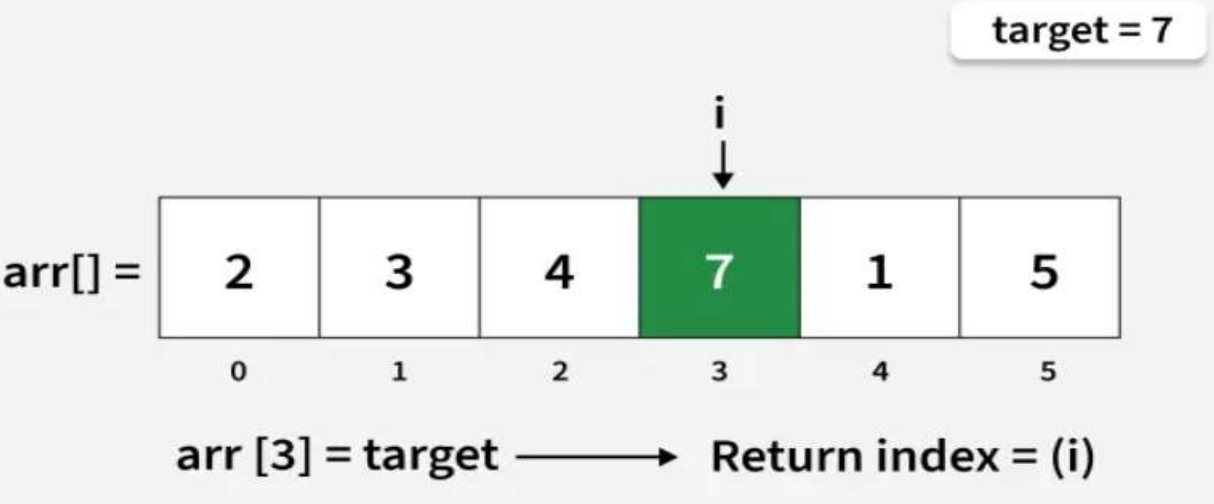
02 Step | If arr[i] is not equal to target, increment i.

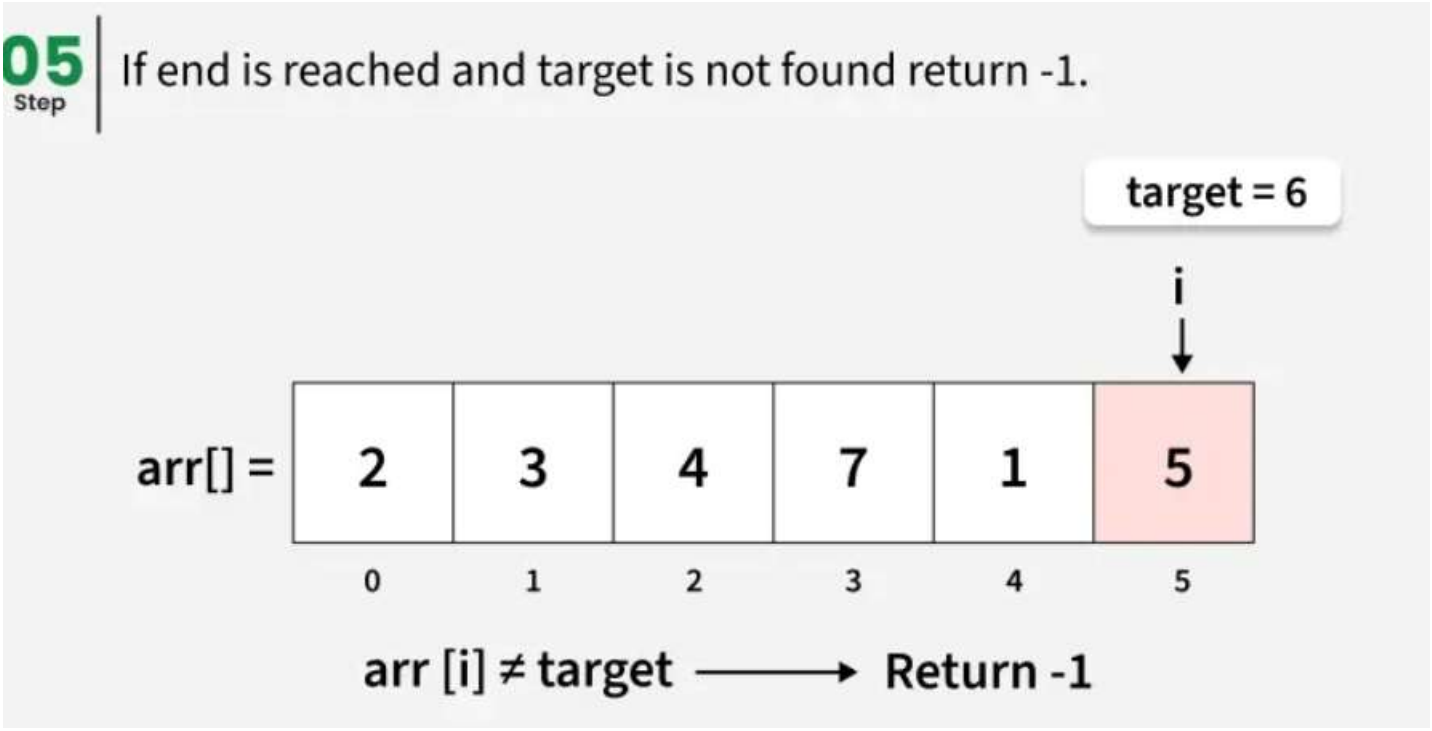


03 Step | Continue until you find arr[i] equal to target.



04 Step | Return the index i where the target is found.






```
#include <stdio.h>


int search(int arr[], int n, int target) {


    // Iterate linearly through the array
    for (int i = 0; i < n; i++)
        if (arr[i] == target)
            return i;
    return -1;
}


int main() {
    int arr[] = {2, 3, 4, 7, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 7;
    int index = search(arr, n, target);
```

```
    printf("%d\n", index);

    return 0;

}
```

Output : -

3

Time Complexity: O(n), where n is the size of the input array. The worst-case scenario is when the target element is not present in the array, and the function has to go through the entire array to figure that out.
Auxiliary Space: O(1), the function uses only a constant amount of extra space to store variables. The amount of extra space used does not depend on the size of the input array.

Answer No. – 13

Structure

In C, a **structure** is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct** keyword is used to define a structure. The items in the structure are called its **members** and they can be of any valid data type. Applications of structures involve creating data structures Linked List and Tree. Structures in C are also used to represent real world objects in a software like Students and Faculty in a college management software.

Example:

```
#include <stdio.h>


// Defining a structure

struct A {

    int x;

};
```

```
int main() {


    // Creating a structure variable
    struct A a;


    // Initializing member
    a.x = 11;


    printf("%d", a.x);
    return 0;
}
```

Output

11

Explanation: In this example, a structure **A** is defined to hold an integer member **x**. A variable **a** of type **struct A** is created and its member **x** is initialized to **11** by accessing it using dot operator. The value of **a.x** is then printed to the console.

Structures are used when you want to store a collection of different data types, such as integers, floats, or even other structures under a single name. To understand how structures are foundational to building complex data structures, the **C Programming Course Online with Data Structures** provides practical applications and detailed explanations.

Syntax of Structure

There are two steps of creating a structure in C:

1. Structure Definition
2. Creating Structure Variables

Structure Definition

A structure is defined using the **struct** keyword followed by the structure name and its members. It is also called a structure **template** or structure **prototype**, and no memory is allocated to the structure in the declaration.

struct *structure_name {*

data_type1 member1;

data_type2 member2;

...

};

- **structure_name:** Name of the structure.
- **member1, member2, ...:** Name of the members.
- **data_type1**, **data_type2**, ...: Type of the members.

Be careful not to forget the semicolon at the end.

Creating Structure Variable

After structure definition, we have to create variable of that structure to use it. It is similar to the any other type of variable declaration:

struct *structure_name var;*

We can also declare structure variables with structure definition.

struct *structure_name {*

...

}var1, var2....;

Program

```
#include <stdio.h>


// structure definition

struct Student {
```

```
    int roll;
    char name[50];
    float marks;
};

int main() {
    struct Student s1;  // structure variable declaration

    // taking input
    printf("Enter Roll Number: ");
    scanf("%d", &s1.roll);

    printf("Enter Name: ");
    scanf("%s", s1.name);

    printf("Enter Marks: ");
    scanf("%f", &s1.marks);

    // displaying output
    printf("\n---- Student Details ----\n");
    printf("Roll No: %d\n", s1.roll);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);
```

```
    return 0;
}
```

Output

```
Enter Roll Number: 101
Enter Name: Suresh
Enter Marks: 89.5

---- Student Details ----
Roll No: 101
Name: Suresh
Marks: 89.50
```

Answer No. - 14

Pointers

A pointer is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the address where the value is stored in memory. It is the backbone of low-level memory manipulation in C. Accessing the pointer directly will just give us the address that is stored in the pointer. For example,

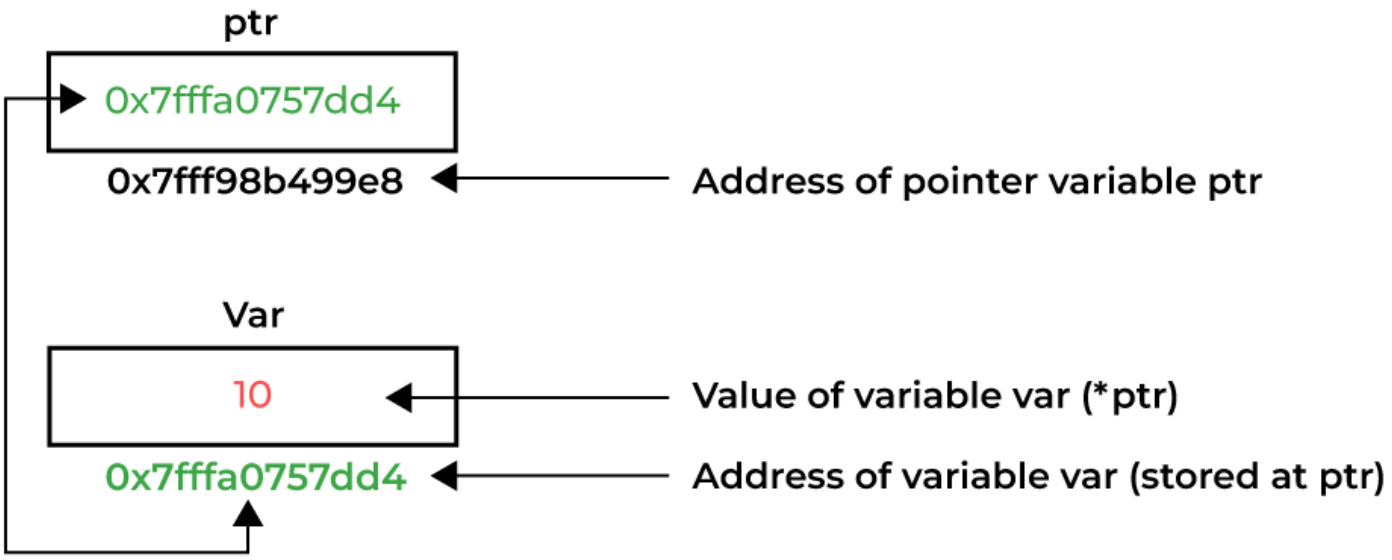
```
#include <stdio.h>
int main() {
    // Normal Variable
    int var = 10;
```

```
    // Pointer Variable ptr that
    // stores address of var
    int* ptr = &var;

    // Directly accessing ptr will
    // give us an address
    printf("%d", ptr);

    return 0;
}
```

Output

0x7fffa0757dd4

This hexadecimal integer (starting with 0x) is the memory address.



Let us understand different steps of the above program.

Declare a Pointer

A pointer is declared by specifying its name and type, just like simple variable declaration but with an **asterisk (*)** symbol added before the pointer's name.

data_type* name

Here, **data_type** defines the type of data that the pointer is pointing to. An integer type pointer can only point to an integer. Similarly, a pointer of float type can point to a floating-point data, and so on.

Example:

int *ptr;

In the above statement, pointer **ptr** can store the address of an integer. It is pronounced as pointer to integer.

Initialize the Pointer

Pointer initialization means assigning some address to the pointer variable. In C, the **(&) addressof operator** is used to get the memory address of any variable. This memory address is then stored in a pointer variable.

Example:

```
int var = 10;


// Initializing ptr
int *ptr = &var;
```

In the above statement, pointer **ptr** store the address of variable **var** which was determined using address-of operator **(&).**

Note: *We can also declare and initialize the pointer in a single step. This is called* ***pointer definition.***

Program

```
#include <stdio.h>


int main() {
    int a = 10;       // normal variable
```

```
    int *p;          // pointer variable


    p = &a;          // p me 'a' ka address store kar diya


    printf("Value of a: %d\n", a);

    printf("Address of a: %p\n", &a);

    printf("Pointer p is storing address: %p\n", p);

    printf("Value at address stored in p: %d\n", *p); // dereferencing


    return 0;

}
```

Output

```
Value of a: 10

Address of a: 0x7ffee8d2a8ac

Pointer p is storing address: 0x7ffee8d2a8ac

Value at address stored in p: 10
```

Answer No. – 15

- **Dynamic Memory Allocation in C**

  Dynamic memory allocation in C means allocating memory while the program is running, instead of at the time of writing the code. It allows you to request memory from the system as needed using functions like malloc(), calloc(), or realloc(), and release it using free(). This gives flexibility to handle data whose size may not be known in advance, like user input or dynamic data structures such as linked lists.

Functions Used for Dynamic Memory Allocation in C

Function	Purpose	Initializes Memory?	Can Resize?	Requires free()?	Header File
malloc()	Allocates single block of memory	No	No	Yes	<stdlib.h>
calloc()	Allocates multiple blocks & zeroes them	Yes	No	Yes	<stdlib.h>
realloc()	Resizes previously allocated memory	No (old data kept)	Yes	Yes	<stdlib.h>
free()	Deallocates previously allocated memory	N/A	N/A	Itself releases	<stdlib.h>

malloc() Function in C

The malloc() function stands for Memory Allocation. It is used to dynamically allocate a single block of memory of the specified size during the execution of the program.

Syntax of malloc() in C

ptr = (cast_type*) malloc(size_in_bytes);

- ptr is the pointer that will store the address of the allocated memory.
- cast_type is the data type you want to store (like int*, float*, etc.).
- size_in_bytes is the total number of bytes to be allocated.

Don't forget to include the header file: #include <stdlib.h>

How malloc() in C Works

- When you call malloc(), it reserves a block of memory of the specified size on the heap.
- It returns a pointer to the first byte of that block.
- The memory is not initialized, so it contains garbage values by default.
- If allocation fails, malloc() returns NULL.

malloc() Example

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

int main() {
    int *arr;
    int n = 5;

    // Allocate memory for 5 integers
    arr = (int*) malloc(n * sizeof(int));

    // Check if memory allocation was successful
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
```

```
        return 1;
    }

    // Assign values and print them
    for (int i = 0; i < n; i++) {
        arr[i] = (i + 1) * 10;
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

Output

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

calloc() Function in C

The calloc() function stands for Contiguous Allocation. It is used to allocate memory for multiple elements and also initializes all of them to zero.

Syntax of calloc() in C

ptr = (cast_type*) calloc(num_elements, size_of_each_element);

- num_elements: Number of elements you want to allocate memory for.
- size_of_each_element: Size of each element in bytes (use sizeof()).
- Returns a pointer of type void*, which should be typecast.

Make sure to include: #include <stdlib.h>

Difference Between malloc() and calloc()

Feature	**malloc()**	**calloc()**
Initialization	Does not initialize memory	Initializes memory to zero
Arguments	Takes total size in bytes	Takes number of elements and size
Use Case	When no initialization is needed	When zero-initialized memory is required

Example of calloc() in C

```
#include <stdio.h>

#include <stdlib.h>


int main() {

    int *arr;

    int n = 5;


    // Allocate memory for 5 integers, all initialized to 0

    arr = (int*) calloc(n, sizeof(int));


    // Check if memory allocation was successful

    if (arr == NULL) {
```

```
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Print default values (should be all zeros)
    for (int i = 0; i < n; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    free(arr);
    return 0;
}
```

When to Prefer calloc() Over malloc()

- When you need all allocated values to be initialized to zero by default.
- For programs involving structures or arrays where zero values avoid undefined behavior.
- When working with bitmaps, flags, or memory buffers that should start clean.

realloc() Function in C

The realloc() function stands for Reallocation of Memory. It is used to resize an already allocated memory block during program execution—either to expand or shrink the existing block.

Syntax of realloc() in C

ptr = (cast_type*) realloc(ptr, new_size_in_bytes);

- ptr: Pointer to the previously allocated memory block.
- new_size_in_bytes: The new size of memory required.

If reallocation is successful, it returns a pointer to the new memory block.

If it fails, it returns NULL.

Include header file: #include <stdlib.h>

How realloc() in C Works

- realloc() attempts to resize the memory block pointed to by ptr.
- If enough space is available at the same memory location, it resizes in place.
- If not, it allocates a new memory block, copies the old data, and frees the old block.
- If ptr is NULL, realloc() behaves like malloc().
- If new_size is 0, it behaves like free().

Example of realloc() in C

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *arr;

   int i;


   // Initial allocation for 3 integers

   arr = (int*) malloc(3 * sizeof(int));


   if (arr == NULL) {

      printf("Initial memory allocation failed.\n");

      return 1;

   }
```

```
// Assign initial values
for (i = 0; i < 3; i++) {
    arr[i] = (i + 1) * 10;
}

// Resize memory to hold 5 integers
arr = (int*) realloc(arr, 5 * sizeof(int));

if (arr == NULL) {
    printf("Memory reallocation failed.\n");
    return 1;
}

// Assign values to new elements
arr[3] = 40;
arr[4] = 50;

// Print all values
for (i = 0; i < 5; i++) {
    printf("arr[%d] = %d\n", i, arr[i]);
}

free(arr); // Free memory
return 0;
```

}

Precautions While Using realloc() in C

Always store realloc() result in a temporary pointer before assigning it back to the original pointer:

```
int* temp = realloc(ptr, new_size);
if (temp != NULL) {
   ptr = temp;
} else {
   // handle allocation failure safely
}
```

This prevents losing the original memory block in case realloc() fails. Check for NULL before using the newly allocated pointer.

If shrinking the memory block, ensure no access is made to freed memory beyond the new size.

Avoid using realloc() on already freed pointers.

free() Function in C

The free() function is used to release dynamically allocated memory back to the system. It helps prevent memory leaks and ensures efficient memory usage during the program's lifecycle.

Syntax of free() in C

```
free(ptr);
```

ptr is a pointer to the memory block that was previously allocated using malloc(), calloc(), or realloc(). Include the required header file: #include <stdlib.h>

Why It Is Important to Deallocate Memory

- Dynamically allocated memory is not freed automatically.
- If you don’t release it using free(), it stays occupied even after it’s no longer needed.
- This leads to memory leaks, especially in long-running or memory-intensive programs.

Example Showing Memory Cleanup

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *arr = (int*) malloc(5 * sizeof(int));


   if (arr == NULL) {

      printf("Memory allocation failed.\n");

      return 1;

   }


   for (int i = 0; i < 5; i++) {

      arr[i] = (i + 1) * 5;

      printf("arr[%d] = %d\n", i, arr[i]);

   }


   // Freeing allocated memory

   free(arr);

   arr = NULL;  // Optional but good practice


   return 0;

}
```

What Happens If free() Is Not Used?

- The memory block remains allocated in the heap even after you're done using it.

- This causes memory leaks, which can degrade performance or crash the program if the memory consumption grows.
- In embedded systems or programs running continuously (like servers), memory leaks can become a critical issue.