Shastri 4th Semester

Computer Science

Unit:3rd

THE OPERATORS USED IN C LANGUAGE

In C, operators are used to performing various operations on variables and constants. They are used to perform mathematical operations, comparison operations, logical operations, and more.

1. **Arithmetic operators:** These operators are used to perform mathematical operations such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).

   int x = 5, y = 2; int sum = x + y; // sum = 7 int diff = x - y; // diff = 3 int prod = x * y; // prod = 10 int quotient = x / y; // quotient = 2 int mod = x % y; // mod = 1

2. **Relational operators:** These operators are used to compare two values and determine their relationship. They include equal to (==), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

3. **Logical operators:** These operators are used to perform logical operations such as AND (&&), OR (||), and NOT (!).

4. **Assignment operators:** These operators are used to assign a value to a variable. The basic assignment operator is "=". There are also shorthand operators that combine an arithmetic or logical operation with an assignment, such as +=, -=, *=, /=, %=, and more.

   int x = 5; x = 7; // x now has the value 7 x += 3; // x now has the value 10 (x = x + 3) x *= 2; // x now has the value 20 (x = x * 2)

5. **Increment and decrement operators:** These operators are used to increment or decrement a variable by 1. They include the increment operator (++) and the decrement operator (--).

6. **Conditional operator:** This operator is a ternary operator that operates on three operands, it returns one of two values depending on the truth of an expression. The syntax is **expression?value_if_true:value_if_false**.
7. **Bitwise operators:** These operators are used to perform operations on individual bits of a value, such as AND (&), OR (|), NOT (~), XOR (^), left shift (<<), and right shift (>>).
8. **Sizeof operator:** This operator returns the size, in bytes, of a variable or data type.
9. **Comma operator:** This operator allows multiple expressions to be executed in a single statement.
10. **Special Operators:** These operators include the pointers operator, member selection operator and member selection by pointer operator, and address of operator and indirection operator.

These are the most common operators used in C, but there are other operators that are used for specialized purposes, such as the ternary operator, the address

A TABLE OF THE OPERATORS USED IN C LANGUAGE

Operator	Description	Example
Arithmetic operators		
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
Relational operators		
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical operators		
&&	Logical AND	x && y
||	Logical OR	x
!	Logical NOT	!x
Assignment operators		
=	Basic assignment	x = y
+=	Add and assign	x += y
-=	Subtract and assign	x -= y
*=	Multiply and assign	x *= y
/=	Divide and assign	x /= y
%=	Modulus and assign	x %= y
Increment and decrement operators		
++	Increment	++x
--	Decrement	--x
Conditional operator		
?:	Conditional operator	x>0 ? x : -x
Bitwise operators		
&	Bitwise AND	x & y
|	Bitwise OR	

The table that lists the multiple assignment operators in C and their use

These operators take two operands and perform the corresponding operation before reassigning the result to the left operand variable.

It is a shorthand notation for combining the operation and assignment in one statement and it can be used to improve the readability and conciseness of code.

Operator	Description	Example
+=	Add and assign	x += y; (x = x + y)
-=	Subtract and assign	x -= y; (x = x - y)
*=	Multiply and assign	x *= y; (x = x * y)
/=	Divide and assign	x /= y; (x = x / y)
%=	Modulus and assign	x %= y; (x = x % y)
&=	Bitwise AND and assign	x &= y; (x = x & y)
|=	Bitwise OR and assign	x |= y; (x = x | y)
^=	Bitwise XOR and assign	x ^= y; (x = x ^ y)

<<=	Left shift and assign	x <<= y; (x = x << y)
>>=	Right shift and assign	x >>= y; (x = x >> y)

5. **Increment and decrement operators:** These operators are used to increment or decrement a variable by 1. They include the increment operator (++) and the decrement operator (--).

   For example:

Operator	Description	Example
++	Increment operator	x++; (increment x by 1)
--	Decrement operator	x--; (decrement x by 1)
++x	Pre-increment operator	++x; (increment x by 1, returns the new value of x)
x++	Post-increment operator	x++; (increment x by 1, returns the original value of x)
--x	Pre-decrement operator	--x; (decrement x by 1, returns the new value of x)
x--	Post-decrement operator	x--; (decrement x by 1, returns the original value of x)

int x = 5; x++; // x now has the value 6 x--; // x now has the value 5

the difference between pre-increment and post-increment (or pre-decrement and post-decrement) operators is the value returned by the operator and the order in which the operation takes place. Pre-increment/decrement operator first increments/decrements the value and then uses it, whereas the post-increment/decrement operator first uses the value and then increments/decrements it.

6. **Conditional operator:** This operator is a ternary operator that operates on three operands, it returns one of two values depending on the truth of an expression. The syntax is an expression?value_if_true:value_if_false.

   int x = 5;

   int y = x>0 ? x : -x; // if x is greater than 0, y = x, otherwise y = -x = 5

Bitwise operators: These operators are used to perform operations on individual bits of a value, such as AND (&), OR (|), NOT (~), XOR (^), left shift (<<), and right shift (>>).

```
int x = 6; //binary representation: 110

int y = 3; //binary representation: 011

int z;

z = x & y; // z = 2 (binary representation: 010)

z = x | y; // z = 7 (binary representation: 111)

z = x ^ y; // z = 5 (binary representation: 101)

z = x << 1; // z = 12 (binary representation: 1100)

z = x >> 1; // z = 3 (binary representation: 0011)
```

Sizeof operator: This operator returns the size, in bytes, of a variable or data type.

Example

```
int x;

double y;

printf("Size of int variable x : %lu bytes", sizeof(x));

printf("Size of double variable y : %lu bytes", sizeof(y));
```

Comma operator: This operator allows multiple expressions to be executed in a single statement.

Example

```
int x=5, y=3;

int z;

z = (x++, ++y); // x=6, y=4, z = 4
```

Special Operators: These operators include the pointers operator, member selection operator and member selection by pointer operator, and address of operator and indirection operator.

Example

```
int x = 5;

int *p;  // pointer to an int

p = &x;  // p points to the memory address of x

int y = *p; // y gets the value stored at the memory location pointed by p which is 5
```

Example

```
struct MyStruct {
   int a;
   double b;
};

struct MyStruct s;

s.a = 5; // Member selection operator

s.b = 3.14;
```

These are just a few examples of how these operators can be used in C. Keep in mind that the use of each operator will vary depending on the specific situation and the context in which it is used.

SWITCH CASE USED IN C LANGUAGE

The switch statement in C is used to make a decision based on multiple possible cases. It tests a variable or expression against a list of case values, and when a match is found, it executes the code associated with that case. If no case matches, then the code in the default case, if any, is executed.

Here is an example of how the switch statement can be used to determine the name of a day of the week based on a number:

Example

```
int day = 3;

switch (day)
```

```
{
 case 1: printf("Monday");
break;
case 2: printf("Tuesday");
 break;
 case 3: printf("Wednesday");
 break;
case 4: printf("Thursday");
break;
case 5: printf("Friday");
break;
case 6: printf("Saturday");
 break;
 case 7: printf("Sunday");
break;
default: printf("Invalid day");
 }
```

In this example, the value of the variable "day" is first tested against the case values 1 through 7. If the value of the day is 3, then the code in the case 3 block is executed, which outputs "Wednesday" to the screen. If the value of the day does not match any of the case values, the code in the default block is executed, which outputs "Invalid day" to the screen.

The **break** statement at the end of each case block is important, it tells the compiler to exit the switch statement when a match is found and prevents the execution of any of the other cases that might match the test value.

That you can use expressions as well in the cases, but the expression should be constant. Also, you can use 'case' multiple times in the same switch statement with the same value.

Some Statements

1. **continue statement**: The continue statement is used in loops (such as for, while, and do-while) to skip the current iteration and move on to the next one. When a continue statement is encountered, the program skips any remaining code in the current iteration of the loop and moves on to the next one. Here's an example of how it can be used in a for loop:

Example

```
for (int i = 0;
i < 10; i++)
{ if (i % 2 == 0)
{
continue; // Skip the rest of the code in this iteration
 if i is even
}
printf("%d ", i); // Only odd numbers will be printed
}
```

2. **goto statement**: The goto statement is used to transfer control to a labeled statement within the same function. It can be used to jump out of a nested loop or to jump to a specific point in the code. The statement is followed by a label to which the control is transferred. It is generally considered as poor practice to use goto statement in code and it can lead to hard-to-understand and hard-to-debug code. Here's an example of how it can be used:

Example

```
int i;
for (i = 0; i < 10; i++)
```

```
{
if (i == 5)
 {
goto end; // Jump out of the loop when i reaches 5
 }
 printf("%d ", i);
 }
 end: printf("Jumped out at i = %d", i);
```

3. **break statement**: The break statement is used in loops (such as for, while and do-while) and switch statements to terminate the loop or switch statement early and move control to the next statement following the loop or switch statement. When a break statement is encountered inside a loop, the loop is immediately terminated and the program resumes execution with the next statement following the loop. Here's an example of how it can be used in a while loop:

```
int i = 0;
while (i < 10)
 {
 i++;
 if (i == 5)
{
 break; // Exit the loop when i reaches 5
}
 printf("%d ", i);
}
```

All of these statements have their specific use cases and usage but should be used cautiously and sparingly, especially the **goto** statement which can lead to messy and hard-to-read code.

C language: Explain Nested Condition with a Simple Example

In C language, nested conditions refer to the practice of using one or more control flow statements (such as if, else, for, while, etc.) inside another control flow statement.

For example, consider the following code:

int age = 20;

if (age >= 18) { printf("You are an adult.\n"); if (age >= 21) { printf("You can buy alcohol.\n"); } else { printf("You cannot buy alcohol.\n"); } } else { printf("You are a minor.\n"); }

In this example, we have an if-else statement within another if-else statement. The outer if-else statement checks if the value of the variable **age** is greater than or equal to 18. If it is, it will print "You are an adult" and then check if the value of the variable **age** is greater than or equal to 21, if it is, it will print "You can buy alcohol" otherwise it will print "You cannot buy alcohol" and if the outer if condition fails it will print "You are a minor".

Nested conditions are useful when we have multiple conditions to check, one after another. It makes the code more readable and maintainable.

too many nested conditions can make the code complex and hard to understand, so it's important to use them judiciously and consider refactoring the code when it becomes hard to understand.

C language: Structured Programming

Structured programming is a programming paradigm that emphasizes the use of structured control flow constructs, such as selection statements (if-else), loops (while, for), and subroutines (functions), in order to organize and simplify the logic of a program.

The structured programming paradigm was first proposed in the 1960s as a way to improve the readability, reliability, and maintainability of software. It is contrasted with the older paradigm of unstructured programming, which relied heavily on the use of goto statements to control the flow of execution.

C language follows the structured programming paradigm, it has the basic structured control flow constructs like if-else, while, for, etc. which helps in organizing the program in a logical and easy-to-understand way.

For example, consider the following code:

Example

```
int x = 5;
int y = 3;
if (x > y)
 {
 printf("x is greater than y\n");
 }
 Else
 {
printf("y is greater than x\n");
}
```

In this example, we have an if-else statement which is a structured control flow construct, it checks the condition **x > y** if it's true then it will execute the first block of code and if it's false it will execute the second block of code.

Structured programming helps in keeping the code organized, easy to understand, and maintain. It also helps in avoiding errors caused by using too many goto statements or complicated control flow statements. The structured programming paradigm is widely used and considered a best practice, it's not the only approach and there are other paradigms like Object-Oriented programming and Functional programming which are widely used as well.

LOOP USED IN C LANGUAGE

A loop is a control flow statement that allows you to repeat a block of code a certain number of times or until a certain condition is met. In C, there are several types of loops that you can use:

1. **while loop**: The while loop repeatedly executes a block of code as long as a certain condition is true. The condition is tested before each iteration of the loop. Here's an example of how a while loop can be used:

Example

```
int i = 0;
 while (i < 10)
{
 printf("%d ", i); i++;
 }
```

In this example, the code inside the loop will be executed repeatedly as long as the condition i < 10 is true. The variable i is incremented after each iteration, and when it reaches 10, the loop will exit and the program will continue with the next statement.

2. **do-while loop**: The do-while loop is similar to the while loop, but it checks the condition after executing the loop body. This means that the body of the loop will always be executed at least once. Here's an example:

Example

```
int i = 0;
do
{
 printf("%d ", i); i++;
}
while (i < 10);
```

In this example, the code inside the loop will be executed at least once, and then the condition i < 10 is tested. If the condition is true, the loop will repeat, otherwise, the loop will exit and the program will continue with the next statement.

3. **for loop**: The for loop is used for the repeated execution of a block of code as long as a given condition is true. The for loop has a special syntax that includes an initialization, a test condition, and an increment/decrement statement all in one line. Here's an example:

```
for (int i = 0; i < 10; i++)

{

printf("%d ", i);

}
```

In this example, the variable i is first initialized to 0, then the condition i < 10 is tested, and if it's true the code inside the loop is executed. After each iteration, the i variable is incremented. As soon as i variable reaches 10, the condition i < 10 becomes false, and the loop exits.

Examples of Loop

some examples of how loops can be used in C, with brief explanations of what they do:

1. **Printing numbers from 1 to 10**: You can use a for loop to print the numbers from 1 to 10.

Example

```
for (int i = 1; i <= 10; i++)

{

 printf("%d ", i);

}
```

In this example, the variable i is initialized to 1 at the start of the loop. The test condition i <= 10 is checked before each iteration, and if it is true, the loop body is executed and the value of i is printed. The variable i is incremented by 1 after each

iteration, until the value of i becomes 11, when the test condition i <= 10 is false, the loop exits.

2. **Calculating the factorial of a number**: You can use a while loop to calculate the factorial of a given number.

Example

```
int num = 5, factorial = 1;

 while (num > 0)

{

 factorial *= num; num--;

 }

 printf("The factorial of 5 is %d", factorial);
```

In this example, the variable num is initialized to 5 and the variable factorial is initialized to 1. The test condition num > 0 is checked before each iteration, and if it is true, the loop body is executed. Inside the loop, the value of the factorial is multiplied by the value of num, and the value of num is decremented by 1. The loop continues to run as long as the value of num is greater than 0. Finally, the program will print "The factorial of 5 is 120"

3. **Printing the multiples of a number**: You can use a do-while loop to print the multiples of a given number up to a certain limit.

Example

```
int num = 5, limit = 50, count = 1;

do

 {

 printf("%d ", num*count); count++;

 }

while (num*count <= limit);
```

In this example, the variable num is initialized to 5, the variable limit is initialized to 50 and the variable count is initialized to 1. The code inside the loop is executed

at least once and the value of num *count is printed. After the first iteration, the variable count is incremented by 1, and the test condition num* count <= limit is checked. If the condition is true, the loop will continue to run and print multiples of num, up to a maximum value of 50. When the condition becomes false, the loop will exit.

These are just a few examples of how loops can be used in C and the possibilities are endless. Loops are powerful constructs that allow you to repeat a block of code multiple times, until a certain condition is met, making it a very important building block of programming.

1. **Printing numbers from 1 to 10**: A common use of loops is to iterate a specific number of times, here's an example of a while loop that prints numbers from 1 to 10:

Example

```
int i = 1; while (i <= 10)
 {
printf("%d ", i); i++;
}
```

2. **Summing up all even numbers from 1 to 10**: A for loop can be used to iterate over a range of numbers, here's an example of a for loop that sums up all even numbers from 1 to 10:

Example

```
int sum = 0;
for (int i = 2; i <= 10; i += 2)
{
sum += i;
}
 printf("Sum: %d\n", sum);
```

3. **Printing the first 10 Fibonacci numbers**: You can use a do-while loop to iterate over a process that needs at least one iteration before checking the condition, here's an example of a do-while loop that prints the first 10 Fibonacci numbers:

```
int a = 0, b = 1, c, i;
 printf("%d %d", a, b); i = 2;
 do
 {
 c = a + b; printf(" %d", c);
 a = b;
 b = c;
i++;
}
 while (i < 10);
```

These examples should provide you with a general idea of how loops work and how they can be used in C. Loops are powerful tools that allow you to repeat actions and are especially useful when working with collections of data such as arrays and lists.

"Top-tested loops" and "bottom-tested loops"

"Top-tested loops" and "bottom-tested loops" are terms used to describe the order in which the test condition of a loop is evaluated.

In a "top-tested loop", the test condition is evaluated at the top of the loop before any iterations take place. This means that if the test condition is false, the loop body will not execute at all. Examples of top-tested loops include the while loop and the for a loop. For example, the while loop checks the condition at the start of the loop and if it's false, the loop will be skipped entirely.

Example

```
int i = 0;
while (i < 10)
 {
```

```
 //top-tested
 printf("%d ", i); i++;
}
```

On the other hand, a "bottom-tested loop" evaluates the test condition at the bottom of the loop, after the loop body has been executed at least once. This means that the loop body will always execute at least once, regardless of the test condition. An example of a bottom-tested loop is the do-while loop, as the test is done after the first iteration of the loop.

Example

```
int i = 0;
 do
{
// bottom-tested
 printf("%d ", i); i++;
 }
 while (i < 10);
```

The choice of which type of loop to use depends on the specific requirements of the task at hand and the specific behavior you want to achieve. However, it's worth keeping in mind that top-tested loops (while and for) are generally more efficient than bottom-tested loops(do-while) as in the case of bottom-tested loops, the code inside the loop will always execute once even if the condition is false.

differences between "top-tested loops" and "bottom-tested loops":

the key difference between the two types of loops is when the test condition is evaluated and whether the loop body is guaranteed to execute at least once.

Type of Loop	Test condition evaluation	Execution of loop body
Top-Tested loops	At the top of the loop before any iterations take place	If the test condition is true, the loop body will execute; if the test condition is false, the loop body will not execute
Bottom-Tested loops	At the bottom of the loop after the loop body has executed	Loop body will always execute at least once, regardless of the test condition

Loop	Example
Top-Tested loops	while, for
Bottom-Tested loops	do-while

Some Question for practice

1. What are the different types of operators in C?
2. How do you use arithmetic operators in C?
3. How do you use relational operators in C?
4. How do you use logical operators in C?
5. How do you use bitwise operators in C?
6. How do you use the ternary operator in C?
7. What is the difference between the = and == operators in C?
8. How do you use the conditional operator in C?
9. What is the if-else statement and how do you use it in C?
10. How do you use the switch-case statement in C?

11. What is the while loop and how do you use it in C?
12. What is the do-while loop and how do you use it in C?
13. What is the for loop and how do you use it in C?
14. How do you use the break statement in C?
15. How do you use the continue statement in C?
16. How do you use the goto statement in C?
17. How do you use the sizeof operator in C?
18. How do you use the typecast operator in C?
19. How do you use the comma operator in C?
20. How do you use the conditional operator in C?
21. What are the different types of operators in C?
22. How do you use arithmetic operators in C?
23. How do you use relational operators in C?
24. How do you use logical operators in C?
25. How do you use bitwise operators in C?
26. How do you use the ternary operator in C?
27. What is the difference between the = and == operators in C?
28. How do you use the conditional operator in C?
29. What is the if-else statement and how do you use it in C?
30. How do you use the switch-case statement in C?
31. What is the while loop and how do you use it in C?
32. What is the do-while loop and how do you use it in C?
33. What is the for loop and how do you use it in C?
34. How do you use the break statement in C?
35. How do you use the continue statement in C?

36.How do you use the goto statement in C?

37.How do you use the sizeof operator in C?

38.How do you use the typecast operator in C?

39.How do you use the comma operator in C?

40.How do you use the conditional operator in C?