Operators:

Operators are used to perform some operations on the input operands. There are different types of operators and they are used for manipulating, calculating, comparing values and for taking logical decisions, etc.

Expressions:

C operators when combined with constants and variables form expressions. Consider the expression B * 3 + C / 5. where, +, *, / are operators, B, C are variables, 3 and 5 are constants.

Types of C operators: C language offers many types of operators. They are:

- Arithmetic operators
- Increment/decrement operators
- Relational operators
- Logical operators
- Bit wise operators
- Conditional operators (ternary operators)
- Assignment operators
- Special operators

Arithmetic Operators

Arithmetic Operators are used to performing mathematical calculations like addition (+), subtraction (-), multiplication (*), division (/) and modulus (%).

Operators	Description	Example	Output

+	Addition	void main()
		{
		
		int a=20, b=3, c;
		c = a+ b;
		printf(“c=%d”, c);
		
		}
			c=23

Bakshi Rohit Prasad- [M.Tech., PhD, IIIT Allahabad]

-	Subtraction	int a=20, b=3, c;
		c = a - b;
		printf(“c=%d”, c);
			c=17

*	Multiplication	int a=20, b=3, c;
		c = a * b;
		printf(“c=%d”, c);
			c=60

/	Division
	• Gives quotient when used with integer operands.
	• If any of numerator or denominator is a real vale then this operator gives real value as a result of division
		int a=20, b=3, c;
		float d;
		c = a / b;
		d = a/3.0
		printf(“c=%d\n”, c);
		printf(“d=%f”, d);
			c=6
			d=6.3333333

%	Modulus
	• Gives remainder when used with integer operands
	• This operator cannot be used with decimal/real operands
		int a=20, b=3, c;
		c = a % b;
		printf(“c=%d”, c);
			c=2

Increment and Decrement Operators

Increment and Decrement operators are shorthand of the calculation; x=x+1 and x=x-1 respectively:

Operator	Description	Example	Output

++	Increment:
	• ++x (Pre-increment) and x++ (Post-increment) means x=x+1
	• Pre-increment, first add 1 to the x and then the value of x will be used in an expression.
	• Post-increment first uses the value of variable x in an expression and then increments the x.
		int x=10,y=10;
		x++;
		++y;
		printf(“x=%d\n”, x);
		printf(“y=%d”, y);
			x=11
			y=11

––	Decrement:
	• --x (Pre-decrement) and x-- (Post-decrement) means x=x-1.
	• Pre-decrement, first subtracts 1 from x and then the value of x will be used in an expression.
	• Post-increment first uses the value of x in an expression and then decrements x.
		int x=10, y=10;
		x--;
		--y;
		printf(“x=%d\n”, x);
		printf(“y=%d”, y);
			x=9
			y=9

Example: To Demonstrate prefix and postfix modes decrement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    //set a and b both equal to 5.
    int a=5, b=5;

    //Print them and decrementing each time.
    //Use postfix mode for a and prefix mode for b.
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    printf("\n%d %d",a--,--b);
    getch();
}
```

Program Output:

```
5 4
4 3
3 2
2 1
1 0
```

Relational Operators

Relational operators are used to:

- Compare two quantities or values.
- This comparison represents some condition check
- If comparison evaluates to true, then checking condition returns True (or 1), otherwise it returns False (or 0).

Operator	Description (Condition that is checked by the operator)	Example	Output

==	Is equal to	int x=10, y=10, z=20;
		
		printf("%d\n", x==y);
		printf("%d", x==z);
			
			1
			0

!=	Is not equal to	int x=10, y=10, z=20;
		
		printf("%d\n", x!=y);
		printf("%d", x!=z);
			
			0
			1

>	Greater than	int x=10, y=10, z=20;
		
		printf("%d\n", x > y);
		printf("%d", z > x);
			
			0
			1

<	Less than	int x=10, y=10, z=20;
		
		printf("%d\n", x < y);
		printf("%d", x < z);
			
			0
			1

>=	Greater than or equal to	int x=10, y=10, z=20;
		
		printf("%d\n", x >= y);
		printf("%d", x > z);
			
			1
			0

<=	Less than or equal to	int x=10, y=10, z=20;
		
		printf("%d\n", x <= y);
		printf("%d", z <= x);
			
			1
			0

Logical Operators

Logical operators allow to join two or more than two test conditions to make a single decision. There are three types of logical operators in C; && (meaning logical AND), || (meaning logical OR) and ! (meaning logical NOT).

Operator	Description	Example	Output

&&	It performs logical conjunction of two expressions.
	• If both expressions evaluate to True, the overall result is True.
	• If any of expressions evaluates to False, the overall result is False
		int a=0, b=0, c=1, d=2;
		int e, f, g, h;
		e = a && b;
		f = a && c;
		g = c && a;
		h = c && d;
		
		printf("e=%d\n", e);
		printf("f=%d\n", f);
		printf("g=%d\n", g);
		printf("h=%d", h);
			
			0
			0
			0
			1

||	It performs a logical disjunction on two expressions.
	• If either or both expressions evaluate to True, the overall result is True
	• If both expressions evaluate to False, the overall result is False
		int a=0, b=0, c=1, d=2;
		int e, f, g, h;
		e = a || b;
		f = a || c;
		g = c || a;
		h = c || d;
		
		printf("e=%d\n", e);
		printf("f=%d\n", f);
		printf("g=%d\n", g);
		printf("h=%d", h);
			
			0
			1
			1
			1

!	It performs logical negation on an expression.
	• If expression evaluates to True, ! gives False and vice-versa.
		int a=0, b=1, c=2;
		int e, f, g, h;
		e = !a;
		f = !b;
		g = !c;
		
		printf("e=%d\n", e);
		printf("f=%d\n", f);
		printf("g=%d", g);
			
			1
			0
			0

Bitwise Operators

C provides a special operator for bit operation between two variables.

Operator	Description	Example	Output

<<	Bitwise Left Shift Operator
	• (<<n) means shift binary bits of number *n* two places left
	• Also, add *n* number of 0s on right side
	
	Example:
	212 = 11010100 (In binary)
	
	212<<0 = 11010100 (Shift by 0)
	
	212<<1 = 110101000 (In binary)
	= 424 (In decimal)
	[Left shift 1 bit, and add one 0's on right side]
	
	212<<2 = 110101000000 (In binary)
	= 848 (In decimal)
	[Left shift 2 bits, and add two 0's on right side]
		#include <stdio.h>
		#include <conio.h>
		
		void main() {
		
		int n=212, i;
		
		printf("Left shift by:\n\n");
		for (i=0; i<=2; i++)
		{
		   n = n<<i;
		   printf("%d bits: %d\n",i, n);
		}
		
		getch();
		}
			Left Shift by:
			
			0 bits: 212
			1 bits: 424
			2 bits: 848

>>	Bitwise Right Shift Operator
	• (>>n) means shift binary bits of number *n* two places right
	• Also, add *n* number of 0s on left side.
	
	Example:
	212 = 11010100 (In binary)
	
	212>>0 = 11010100 (No Shift)
	
	212>>1 = 01101010 (In binary)
	= 106 (In decimal)
	[Right shift 1 bit, and add one 0's on left side]
	
	212>>2 = 00110101 (In binary)
	= 53 (In decimal)
	[Right shift 2 bit, and add two 0's in left side]
		#include <stdio.h>
		#include <conio.h>
		
		void main() {
		
		int n=212, i;
		
		printf("Right shift by:\n\n");
		for (i=0; i<=2; i++)
		{
		   n = n>>i;
		   printf("%d bits: %d\n",i, n);
		}
		
		getch();
		}
			Right Shift by:
			
			0 bits: 212
			1 bits: 106
			2 bits: 53

~	Bitwise NOT (or Ones Complement) Operator
	• Individual bits of the operand is complemented, i.e., 1 is changed to 0 and vice-versa
	
	Example:
	35 = 00100011 (In Binary)
	
	Bitwise complement Operation of 35
	~ 00100011
	
	11011100
	
	(Since, MSB is 1, it implies that the number is negative, so again take the 2’s complement and convert it into decimal value and put – sign before it)
	
	Now, 2’s complement of 220 is:
	
	= -(00100011+1)
	= -(00100100)
	= -36 (In decimal)
		#include <stdio.h>
		#include <conio.h>
		
		void main()
		{
		    printf("%d\n", ~35);
		    printf("%d\", ~-12);
		
		    getch();
		}
			-36
			11

&	Bitwise AND Operator
	• Output of bitwise AND is 1 if the corresponding bits of two operands is 1.
	• Otherwise output is 0.
	
	Example:
	12 = 00001100 (In Binary)
	25 = 00011001 (In Binary)
	
	Bit Operation of 12 and 25
	
	  00001100
	& 00011001
	
	  00001000 = 8 (In decimal)
		#include<stdio.h>
		#include<conio.h>
		
		void main()
		{
		    int a = 12, b = 25;
		    printf("%d", a&b);
		
		    getch();
		}
			8

^	Bitwise XOR Operator
	• Output of bitwise OR is 0 if the corresponding bits of two operands are same, i.e. either both 1 or 0.
	• Otherwise output is 1.
		#include <stdio.h>
		#include <conio.h>
		
		void main()
		{
		    int a = 12, b = 25;
		    printf("%d", a^b);
			21

	Example:
	12 = 00001100 (In Binary)
	25 = 00011001 (In Binary)
	
	Bitwise XOR Operation of 12 and 25
	  00001100
	^ 00011001
	  ________
	  00010101 = 21 (In decimal)
		`    getch();`
		`}`	

|	Bitwise OR Operator
	• Output of bitwise OR is 0 if the corresponding bits of two operands is 0.
	• Otherwise output is 1.
	
	Example:
	12 = 00001100 (In Binary)
	25 = 00011001 (In Binary)
	
	Bitwise OR Operation of 12 and 25
	  00001100
	| 00011001
	  ________
	  00011101 = 29 (In decimal)
		`#include <stdio.h>`
		`#include <conio.h>`
		
		`void main()`
		`{`
		`    int a = 12, b = 25;`
		`    printf("%d", a|b);`
		
		`    getch();`
		`}`
			29

Assignment Operators

Assignment operators applied to assign the result of an expression to a variable. C has a collection of shorthand assignment operators.

Operator	**Description**	**Example**	**Output**

=	Assign	int a=20, c;
		c = a;
		printf(“c=%d”, c);
			c=20

+=	Increments then assign:
	• (a += b), means (a = a + b)
		int a=20, b=3;
		a += b;
		printf(“a=%d”, a);
			a=23

-=	Decrements then assign:
	• (a -= b), means (a = a - b)
		int a=20, b=3;
		a -= b;
		printf(“a=%d”, a);
			a=17

*=	Multiplies then assign:
	• (a *= b), means (a = a * b)
		int a=20, b=3;
		a *= b;
		printf(“a=%d”, a);
			a=60

/=	Divides then assign:
	• (a /= b), means (a = a / b)
		int a=20, b=3;
		a /= b;
		printf(“a=%d”, a);
			a=6

%=	Modulus then assign
	• (a %= b), means (a = a % b)
		int a=20, b=3;
		a %= b;
		printf(“a=%d”, a);
			a=2

Similarly, short-hands corresponding to <<=, >>=, &=, ^=, |= can be expanded and calculated.			

Conditional Operator

C offers a ternary operator which is called as ‘conditional operator’ (? :). This operator is used to construct conditional expressions like if-else.

Operator	**Description**

? :	Conditional operator is a Ternary operator because it works on three operands. As given below in the syntax, there are three operands; Conditional-expression1, expression2 and expression3.
	
	Syntax: *Contional-expression1 ? expression2 : expression3*
	
	First of all, Conditional-expression1 is evaluated.
	✓ If result of expression1 is TRUE, then expression2 is executed
	✓ If result of expression1 is FALSE, then expression3 is executed
	
	Example Program:
	
	void main
	{
	    int x;
	
	    x = 5 > 8 ? 10 : 20; /* Condtion-expression1, i.e. 5>8 is evaluated first. Since, it is false, 20 is assigned to variable x */
	    printf("Value of x= %d", x);
	    getch();
	}
	
	Output:
	
	Value of x= 20

Special Operators

C supports some special operators

Operator	Description
sizeof()	Returns the size of a memory location.
&	Returns the address of a memory location.
*	Pointer to a variable.

Use of sizeof operator

Program Example: Size of data type may vary from compiler to compiler. In following program, Turbo C/C++ compiler has been considered as reference.

```
#include<stdio.h>
#include<conio.h>
void main()
{
     /* Variables Defining and Assign values */
     int a=10;
     float b=4.32;
     printf("integer: %d\n", sizeof(a));
     printf("float: %d\n", sizeof(b));
     getch();
}
```

Program Output:

```
integer:  2
float:  4
```

Operator Precedence:

C language has a predefined rule of priority for the operators called operator precedence. If more than one operators are involved in an expression, this rule of priority of operators decides the order in which these operators will be executed. For example, in C, precedence of arithmetic operators (*, %, /, +, -) is higher than relational operators (==, !=, >, <, >=, <=) and precedence of relational operator is higher than logical operators(&&, || and !).

Example of precedence: Let’s take following expression containing multiple types of operators:

(1 > 2 + 3 && 4)

Here, there are three types of operators; Relational operator >, Arithmetic operator +, Logical operator &&. Considering operator precedence, first of all operator + is executed, then operator > is executed and finally, operator && is executed. Thus, above expression will be processed as following:

```
  1 > 2 + 3 && 4          // 2 + 3 executes first resulting into 5
= 1 > 5 && 4              // 1 > 5 executes resulting into 0 (False)
= 0 && 4                  // 0 && 4 executes resulting into 0 (False)
= 0
```

Associativity of operators

If two operators of same precedence (priority) is present in an expression, Associativity of operators indicate the order in which they execute.

Example of associativity: Consider following expression having multiple logical operators having same precedence.

1 == 2 != 3

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression on the left is executed first and moves towards the right. That means, == is executed first and then != operates. As given below, the above expression is equivalent to:

```
  ((1 == 2) != 3)         // (1 == 2) executes first resulting into 0 (false)
= ( 0 != 3 )              // (0 != 3) executes resulting into 1 (true)
= 1
```

Operator Precedence and Associativity in C Language

Note: Precedence of operators decreases from top to bottom in the given table.

Operator	Description	Precedence / Priority	Associativity
( ) [ ] . ->	Function call Array element reference Member selection via object name Member selection via pointer	1	left-to-right
++ — + – ! ~ (type) * & sizeof	increment / decrement Unary plus / minus Logical negation / bitwise complement Type Cast (conversion of value's data type) Dereference Address-of Determine size in bytes	2	right-to-left
* / %	Multiplication/division/modulus	3	left-to-right
+ –	Addition/subtraction	4	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	5	left-to-right
< <= > >=	Relational less than / less than or equal to Relational greater than / greater than or equal to	6	left-to-right
== !=	Relational is equal to / is not equal to	7	left-to-right
&	Bitwise AND	8	left-to-right
^	Bitwise exclusive OR	9	left-to-right
|	Bitwise inclusive OR	10	left-to-right
&&	Logical AND	11	left-to-right
| |	Logical OR	12	left-to-right
? :	Ternary conditional	13	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	14	right-to-left
,	Comma (separate expressions)	15	left-to-right

Bakshi Rohit Prasad- [M.Tech., PhD, IIIT Allahabad]

Mixed Operands: In an expression, when operands (constants or variables) of different data types are used, then such expressions are called mixed-operand expressions. In such cases, type conversion becomes necessary for evaluating the expression. **For example:** Following expression has mixed operands (some are integer values and some are float values):

3 + 5.12 - 4.5 * 10

Type Conversion: In a mixed operand expression, all operands are converted to same data type during calculation. This process of converting a value of predefined data type into a value of another data type is called Type Conversion. In C language, type conversion is classified into two types:

1. **Automatic Type Conversion (Implicit Type Conversion):** In a mixed operand expression, when **compiler automatically converts all the operands of lower size data types into largest size data type.** This process is called Automatic Type Conversion or Implicit Type Conversion. **For example:** Consider following expression with mixed operands:

   3 + 5.12 - 0.5 * 10

   Here, two values are integer and two values are floating point. Integer is a lower sized data type than float. Therefore, during the calculation, first of all integer values are internally converted to float. Thus, above expression becomes:

```
     3.0 + 5.12 – 0.5 * 10.0      /* Multiplication has highest precedence, hence, calculated first */
  =  3.0 + 5.12 – 5.0             /* Operator + and – has equal precedence, but left to right associativity,
                                     hence, + will be executed first */
  =  8.12 – 5.0
  =  3.12
```

2. **Type Casting (Explicit Type Conversion):** Type Casting is used to convert an operand of one data type to another data type. **This kind of conversion in a program is explicitly written by the programmer.** After type casting, during calculation, compiler considers the operand as newer data type. The syntax is as follows:

   Syntax: *(data_type_name) value*

   Example: Consider following expression which gives an integer value 5 which is stored in variable x:

	Without Type Casting	**With Type Casting**

Program	`#include<stdio.h>`
	`#include<conio.h>`
	`void main ()`
	`{`
	`    float x;`
	`    x = 10 / 4;`
	`    printf("value of x = %f", x);`
	`    getch();`
	`}`
		`#include<stdio.h>`
		`#include<conio.h>`
		`void main ()`
		`{`
		`    float x;`
		`    x = (float) 10 / 4;`
		`    printf("value of x = %f", x);`
		`    getch();`
		`}`

Output	value of x = 2.000000	value of x = 2.500000
Description	Since, 10 and 4 both are integers therefore, division operator / will give quotient, i.e. 2.	Here, 10 is type casted to float and then division is operated. Therefore, division operator / will perform floating point division and result will be i.e. 2.500000