Unit 4 : Expression and Operator

Er. Nipun Thapa

4.1. Operator

- Operators are the foundation of any programming language.
- It can define operators as symbols that help us to perform specific mathematical and logical computations on operands.
- In other words, we can say that an operator operates the operands.
- For example, consider the below statement:
  - c = a + b;
  - Here, '+' is the operator known as *addition operator* and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b'.

4.1. Operator

Operator Classification

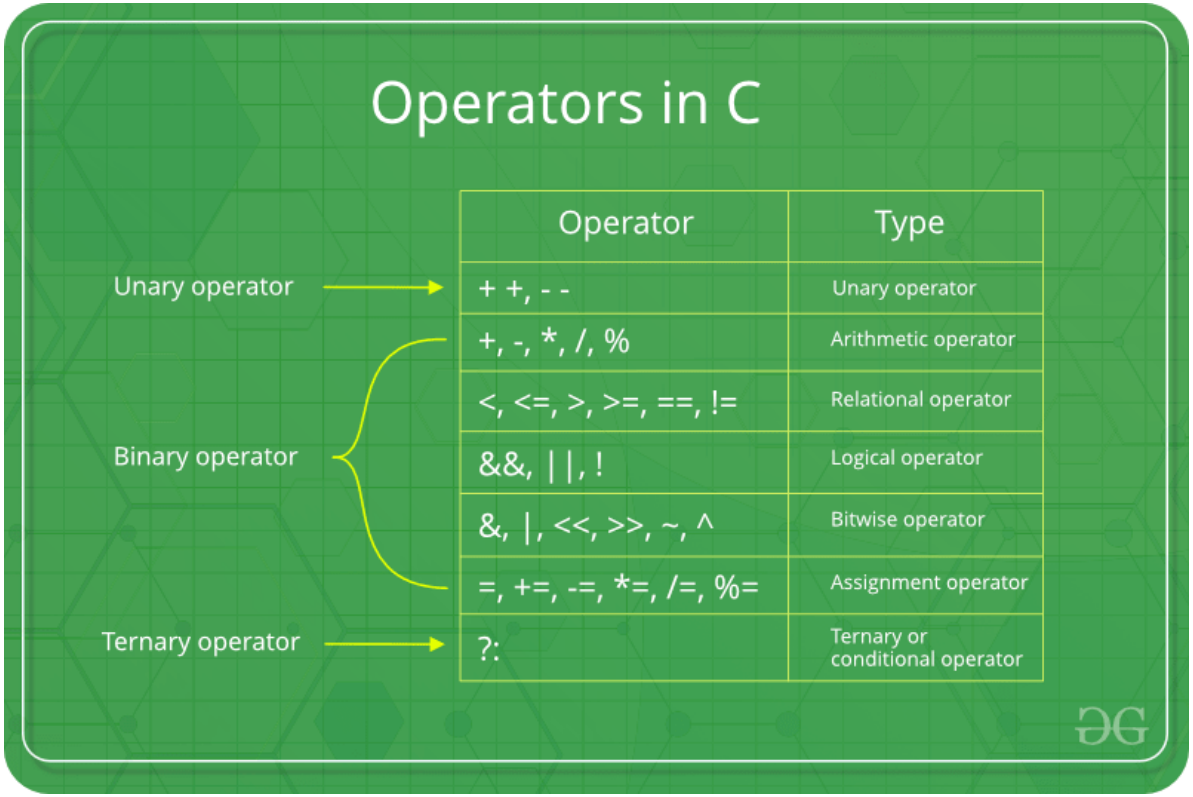
a) According to Number of Operands

- **Unary Operators**: Operators that operates or works with a single operand are unary operators. For example: (++,)
- **Binary Operators**: Operators that operates or works with two operands are binary operators. For example: (+ , – , * , /)
- **Ternary Operators**: The operators which require three operands to operate are called ternary operators. E.g. the operator pair “? :” is ternary operator in C.

4.1. Operator

Operator Classification

a) According to Number of Operands



4.1. Operator

Operator Classification

b) According to utility and Action

- Arithmetic Operator
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional Operators (Ternary Operator)
- Bitwise Operators
- Special Operators(Comma Operator and size of Operator)

4.2.Arithmetic Operators

- These are used to perform arithmetic/mathematical operations on operands. C supports all the basic arithmetic operators.
- The following table shows all the basic arithmetic operators. Here A=10 and B=20

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

4.2.Arithmetic Operators (Example)

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;
    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);
    return 0;
 }
```

4.2.Arithmetic Operators (Example)

- //LAB 2. Qn.4 : Program to convert days into days and month

```
#include<stdio.h>
int main()
{
int day,month;
printf("Enter days:");
scanf("%d",&day);
month=day/30;
day=day%30; // reminder
printf("%d Month",month);
printf("\n%d Days",day);
return 0;
}
```

4.2.Arithmetic Operators (Example)

- //LAB2.Qn.5:Program that read second and convert into hour,minute and second.

```
#include<stdio.h>
int main()
{
int sec,min,hr,rsec;
printf("Enter second:");
scanf("%d",&sec);
hr=sec/3600;
rsec=sec%3600;
min=rsec/60;
sec=rsec%60;
printf("\n %d hour, %d minutes, %d second",hr,min,sec);
return 0;
}
```

4.2.Arithmetic Operators (Example)

- ```
  /*LAB2.Q.N.6:Program to sum the digit of a positive integer
  which is 3 digit long.e.g. 135 : 1+3+5=9*/
  #include<stdio.h>
  int main()
  {
          int n,digit1,digit2,digit3,sum;
          printf("Enter 3 digit number:");
          scanf("%d",&n);

          digit1=n%10;
          n=n/10;
          digit2=n%10;
          n=n/10;
          digit3=n%10;
          sum=digit1+digit2+digit3;
          printf("Sum=%d",sum);
          return 0;
  }
  ```
  ```

4.3. Relational Operators

- A relational operator checks the relationship between two operands.
  - If the relation is true,

    it returns 1;
  - if the relation is false,

    it returns value 0.

4.3. Relational Operators

Assume variable **A** holds 10 and variable **B** holds 20 then –

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.

4.3. Relational Operators

```
// Working of relational operators
#include <stdio.h>
int main()
    {   int a = 5, b = 5, c = 10;
        printf("%d == %d is %d \n", a, b, a == b);
        printf("%d == %d is %d \n", a, c, a == c);
        printf("%d > %d is %d \n", a, b, a > b);
        printf("%d > %d is %d \n", a, c, a > c);
        printf("%d < %d is %d \n", a, b, a < b);
        printf("%d < %d is %d \n", a, c, a < c);
        printf("%d != %d is %d \n", a, b, a != b);
        printf("%d != %d is %d \n", a, c, a != c);
        printf("%d >= %d is %d \n", a, b, a >= b);
        printf("%d >= %d is %d \n", a, c, a >= c);
        printf("%d <= %d is %d \n", a, b, a <= b);
        printf("%d <= %d is %d \n", a, c, a <= c);
        return 0;
    }
```

4.4. Logical Operator

- An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false.
- Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c==5) || (d>5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c==5) equals to 0.

4.4. Logical Operator (Example)

```
#include <stdio.h>
int main()
{           int a = 5, b = 5, c = 10, result;
            result = (a == b) && (c > b);
            printf("(a == b) && (c > b) is %d \n", result);
            result = (a == b) && (c < b);
            printf("(a == b) && (c < b) is %d \n", result);
            result = (a == b) || (c < b);
            printf("(a == b) || (c < b) is %d \n", result);
            result = (a != b) || (c < b);
            printf("(a != b) || (c < b) is %d \n", result);
            result = !(a != b);
            printf("!(a != b) is %d \n", result);
            result = !(a == b);
            printf("!(a == b) is %d \n", result);
            return 0;
}
```

4.4. Logical Operator

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a==b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

4.5. Assignment Operators

- Assignment operators are used to assign value to a variable.
- The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value.
- The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.

4.5. Assignment Operators

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

4.6. Increment and Decrement Operator

- **Increment:**
  - The **'++'** operator is used to increment the value of an integer.
  - When placed before the variable name (also called pre-increment operator), its value is incremented instantly. For example, **++x**.
  - And when it is placed after the variable name (also called post-increment operator), its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement. For example, **x++**.
- **Decrement:**
  - The **' – – '** operator is used to decrement the value of an integer.
  - When placed before the variable name (also called pre-decrement operator), its value is decremented instantly. For example, **– – x**.
  - And when it is placed after the variable name (also called post-decrement operator), its value is preserved temporarily until the execution of this statement and it gets updated before the execution of the next statement. For example, **x – –**.

4.6. Increment and Decrement Operator (Example)

```
#include <stdio.h>
 int main()
{  int a = 10, b = 4, res;
    res = a++;
    printf("a is %d and res is %d\n", a, res);
    res = a--;
    printf("a is %d and res is %d\n", a, res
    res = ++a;
    printf("a is %d and res is %d\n", a, res);
    res = --a;
    printf("a is %d and res is %d\n", a, res);
    return 0;
}
```

4.7. Conditional Operator

- The conditional operators in C language are known by two more names
  1. **Ternary Operator**
  2. **? : Operator**
- It is actually the **if** condition that we use in C language decision making, but using conditional operator, we turn the **if** condition statement into a short and simple operator.

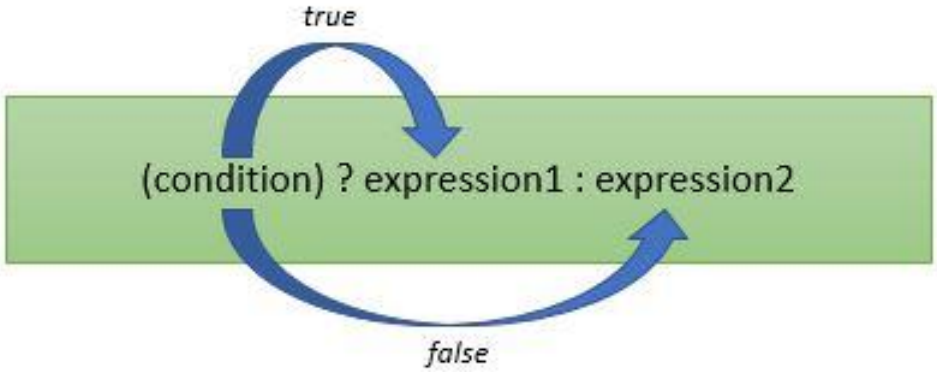
4.7. Conditional Operator

The syntax of a conditional operator is :

expression 1 ? expression 2: expression 3

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.



4.7. Conditional Operator(Example)

```
#include <stdio.h>
int main()
{
   int mark;
   printf("Enter mark: ");
   scanf("%d", &mark);
   puts(mark >= 40 ? "Passed" : "Failed");
   return 0;
}
```

4.7. Conditional Operator(Example)

```
/*Lab2 QN11 Program to to find larger value using
   conditional operator */

   #include<stdio.h>
   int main()
   {
         int n1,n2,larger,smaller;
         printf("Enter two number:");
         scanf("%d%d",&n1,&n2); // 6 8
         larger=n1>n2 ? n1 : n2; // 6>8
         printf("The lager is %d",larger);

         return 0;
   }
```

4.7. Conditional Operator(Example)

```
/*lab 2 QN 12. Program to find largest number
   among three integer value using conditional operator.*/

   #include<stdio.h>
   int main()
   {
         int n1,n2,n3,large1,large2;
         printf("Enter three number:");
         scanf("%d%d%d",&n1,&n2,&n3); //2 1 4
         large1=n1>n2?n1:n2; // large1= 2
         large2=large1>n3?large1:n3; // large2=4
         printf("The largest value is %d",large2);
         return 0;
}
```

4.8. Bitwise Operators

- The Bitwise operators is used to perform bit-level operations on the operands.
- The operators are first converted to bit-level and then the calculation is performed on the operands.
- The mathematical operations such as addition, subtraction, multiplication etc. can be performed at bit-level for faster processing.
- For example, the **bitwise AND** represented as **& operator in C or C++** takes two numbers as operands and does AND on every bit of two numbers.
- The result of AND is 1 only if both bits are 1

4.8. Bitwise Operators

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

4.8. Bitwise Operators (Example)

```
#include<stdio.h>
int main()
{
    int num1=178;
     int num2=63;
     int and, or, xor;
     and= num1 & num2;
     or= num1 | num2;
     xor=num1 ^ num2;
     printf("and= %d\n",and);
     printf("or= %d\n",or);
     printf("xor=%d\n",xor);
     return 0;
}
```

Explain (AND)

```
      1011 0010 :   (178)
&     0011 1111 :   (63)
=     0011 0010 :   (50)
```

OR

```
      1011 0010 :   (178)
|     0011 1111 :   (63)
=     1011 1111 :   (190)
```

XOR

```
      1011 0010 :   (178)
^     0011 1111 :   (63)
=     1000 1101 :   (141)
```

4.9. Comma Operator

- The comma operator (represented by the token ,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator

4.9. Comma Operator

1) Comma as an operator:

- The comma operator (represented by the token,) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type). The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

```
/* comma as an operator */
int i = (5, 10);        /* 10 is assigned to i*/
int j = (f1(), f2());   /*  f1() is called (evaluated) first followed by f2().
                            The returned value of f2() is assigned to j */
```

2) Comma as a separator:

- Comma acts as a separator when used with function calls and definitions, function like macros, variable declarations, enum declarations, and similar constructs.

```
/* comma as a separator */
int a = 1, b = 2;
void fun(x, y);
```

4.9. Comma Operator

3) Comma operator in place of a semicolon.

- We know that in C and C++, every statement is terminated with a semicolon but comma operator also used to terminate the statement after satisfying the following rules.
  - The variable declaration statements must be terminated with semicolon.
  - The statements after declaration statement can be terminated by comma operator.
  - The last statement of the program must be terminated by semicolon.

4.10. The size of Operator

- **sizeof** is a much used in the C/C++ programming language.
- It is a compile time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of unsigned integral type which is usually denoted by size_t.
- Basically, sizeof operator is used to compute the size of the variable.

Example :

```
#include <stdio.h>
int main()
{
   printf("%lu\n", sizeof(char));
   printf("%lu\n", sizeof(int));
   printf("%lu\n", sizeof(float));
   printf("%lu", sizeof(double));
   return 0;
}
```

4.11.Precedence and Associativity of Operators

- **Operator precedence:** It dictates the order of evaluation of operators in an expression.
- **Associativity:** It defines the order in which operators of the same precedence are evaluated in an expression. Associativity can be either from left to right or right to left.

example:

- 24 + 5 * 4
- Here we have two operators + and *, Which operation do you think will be evaluated first, addition or multiplication? If the addition is applied first then answer will be 116 and if the multiplication is applied first answer will be 44. To answer such question we need to consult the operator precedence table.

4.11.Precedence and Associativity of Operators

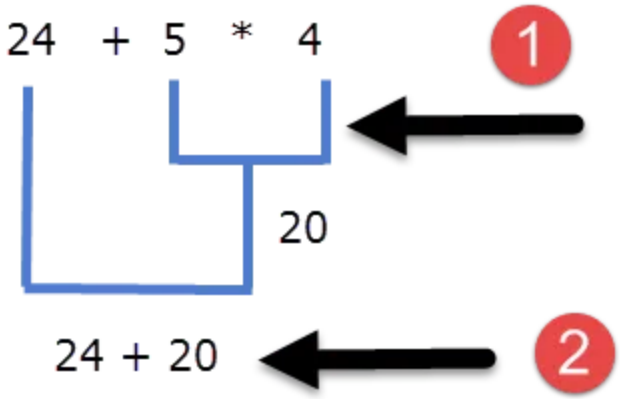
Precedence level	Operator	Description	Associativity
1	() [] . -> ++ --	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
2	++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
3	* / %	Multiplication, division and modulus	left to right
4	+ -	Addition and subtraction	left to right
5	<< >>	Bitwise left shift and right shift	left to right
6	< <= > >=	relational less than/less than equal to relational greater than/greater than or equal to	left to right

4.11.Precedence and Associativity of Operators

7	== !=	Relational equal to and not equal to	left to right
8	&	Bitwise AND	left to right
9	^	Bitwise exclusive OR	left to right
10	|	Bitwise inclusive OR	left to right
11	&&	Logical AND	left to right
12	||	Logical OR	left to right
13	? :	Ternary operator	right to left
14	= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
15	,	Comma operator	left to right

4.11.Precedence and Associativity of Operators

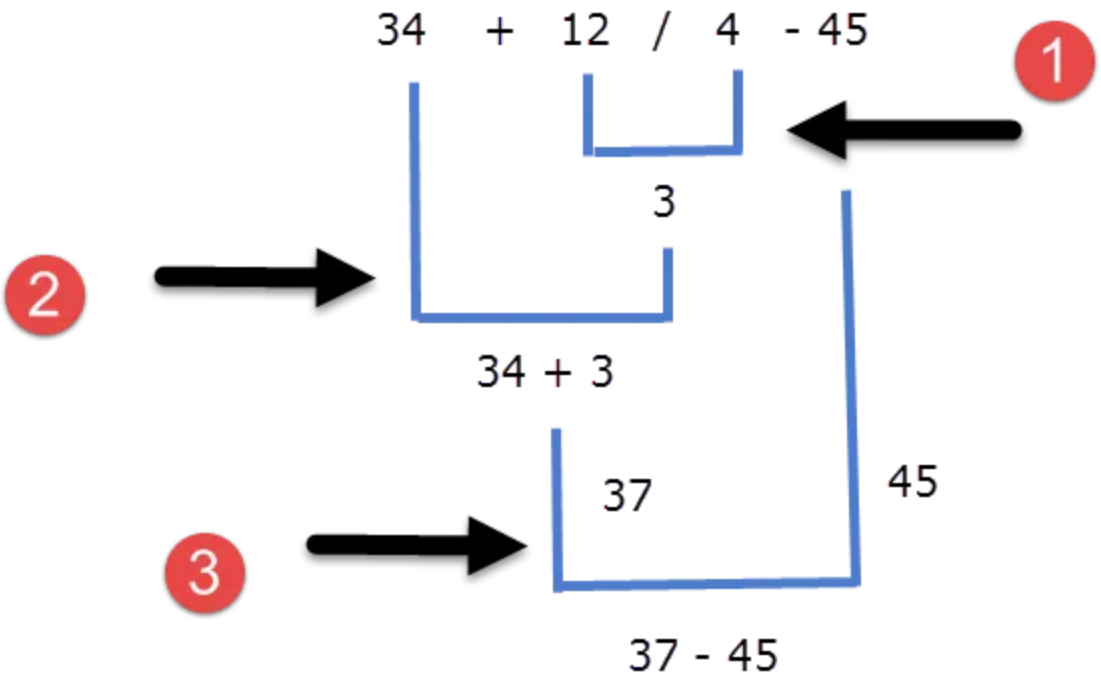
- From the precedence table, we can conclude that the * operator is above the + operator, so the * operator has higher precedence than the + operator, therefore in the expression 24 + 5 * 4, subexpression 5 * 4 will be evaluated first.



Ans : 44

4.11.Precedence and Associativity of Operators

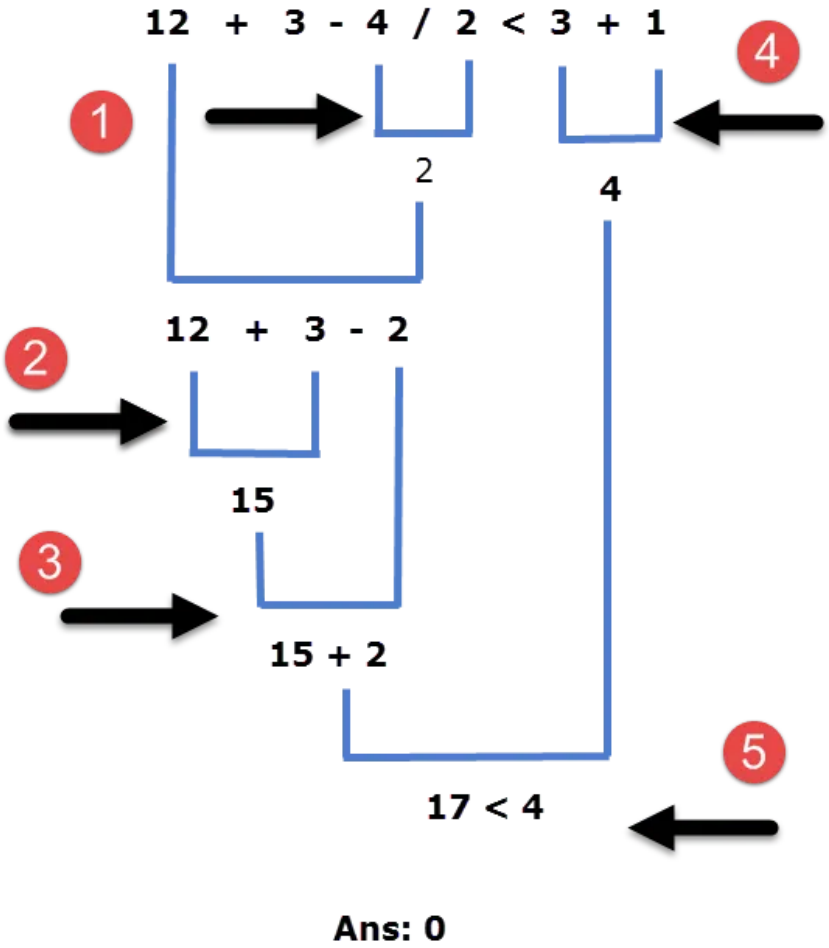
- 34 + 12/4 - 45



Ans : -8

4.11.Precedence and Associativity of Operators

- 12 + 3 - 4 / 2 < 3 + 1



N= 2*3/4+4/4+8-2+5/8 where N is integer

Operator : seen

N= 2*3/4+4/4+8-2+5/8 = 6/4+4/4+8-2+5/8

N= 6/4+4/4+8-2+5/8 = 1+4/4+8-2+5/8

N= 1+4/4+8-2+5/8 = 1+1+8-2+5/8

N= 1+1+8-2+5/8=1+1+8-2+0

N=1+1+8-2 = 2+8-2

N=2+8-2=10-2

N=8

Type Conversion in Expressions

- The **type conversion process in C** is basically converting one type of data type to other to perform some operation.
- The conversion is done only between those datatypes wherein the conversion is possible ex – char to int and vice versa.
- It can classify in two group

  1. Implicit Type Conversion(Automatic Type Conversion)

  2. Explicit Type Conversion(Type cast)

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

- This type of conversion is usually performed by the compiler when necessary without any commands by the user. Thus it is also called **"Automatic Type Conversion"**.
- The compiler usually performs this type of conversion when a particular expression contains more than one data type. In such cases either type promotion or demotion takes place.

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

Rules...

1. char or short type operands will be converted to int during an operation and the outcome data type will also be int.
2. If an operand of type long double is present in the expression, then the corresponding operand will also be converted to long double same for the double data type.
3. If an operand of float type is present then the corresponding operand in the expression will also be converted to float type and the final result will also be float type.
4. If an operand of unsigned long int is present then the other operand will be converted to unsigned long int and the final result will also be unsigned long int.
5. If an operand of long int is present then the other operand will be converted to long int and the final result will also be long int.
6. If an operand of unsigned int is present then the other operand will be converted to unsigned int and the final result will also be unsigned int.

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

Example 1

```
int a = 20;
double b = 20.5;
a + b;
```

Here, first operand is int type and other is of type double. So, as per rule 2, the variable a will be converted to double. Therefore, the final answer is double **a + b = 40.500000.**

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

Example 2

```
char ch='a';
int a =13;
a + c;
```

Here, first operand is **char** type and other is of type **int**. So, as per rule 1, the **char** variable will be converted to **int** type during the operation and the final answer will be of type **int**.

We know the ASCII value for ch is 97. Therefore, final answer is

a + c = 97 + 13 = 110.

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

Example 3

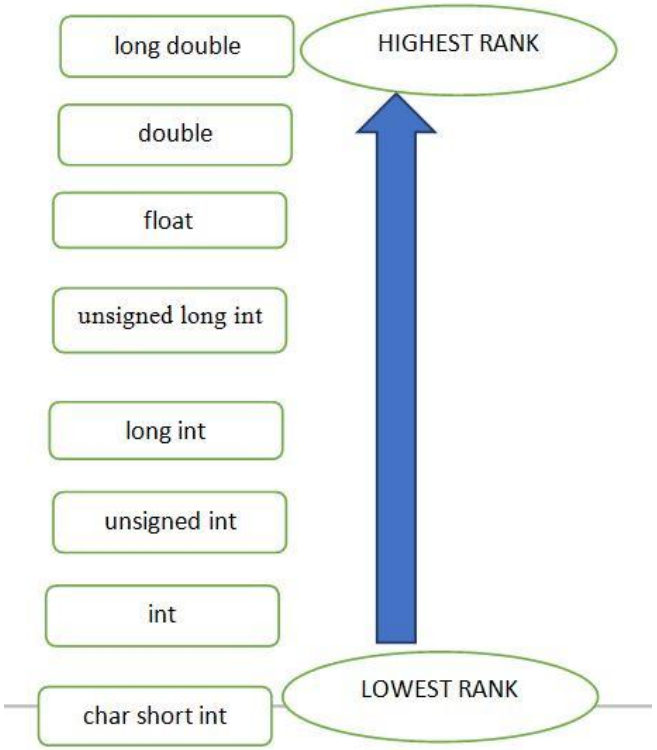
```
char ch='A';
unsigned int a =60;
a * b;
```

Here, the first operand is **char** type and the other is of type **unsigned int**. So, as per rule 6, **char** data type will be converted to **unsigned int** during the operation. Therefore, the final result will be an unsigned int variable, 65 + 60 = 125.

Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

For ease of understanding follow the flowchart given below, in which datatypes have been arranged in a hierarchy of highest to the lowest rank.



Type Conversion in Expressions

1. Implicit Type Conversion(Automatic Type Conversion)

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
   int x = 10;    // integer x
   char y = 'a';  // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.0;

   printf("x = %d, z = %f", x, z);
   return 0;
}
```

Output:

x = 107, z = 108.000000

Type Conversion in Expressions

2) Explicit Type Conversion

- Explicit type conversion rules out the use of compiler for converting one data type to another instead the user explicitly defines within the program the datatype of the operands in the expression.
- The example below illustrates how explicit conversion is done by the user.

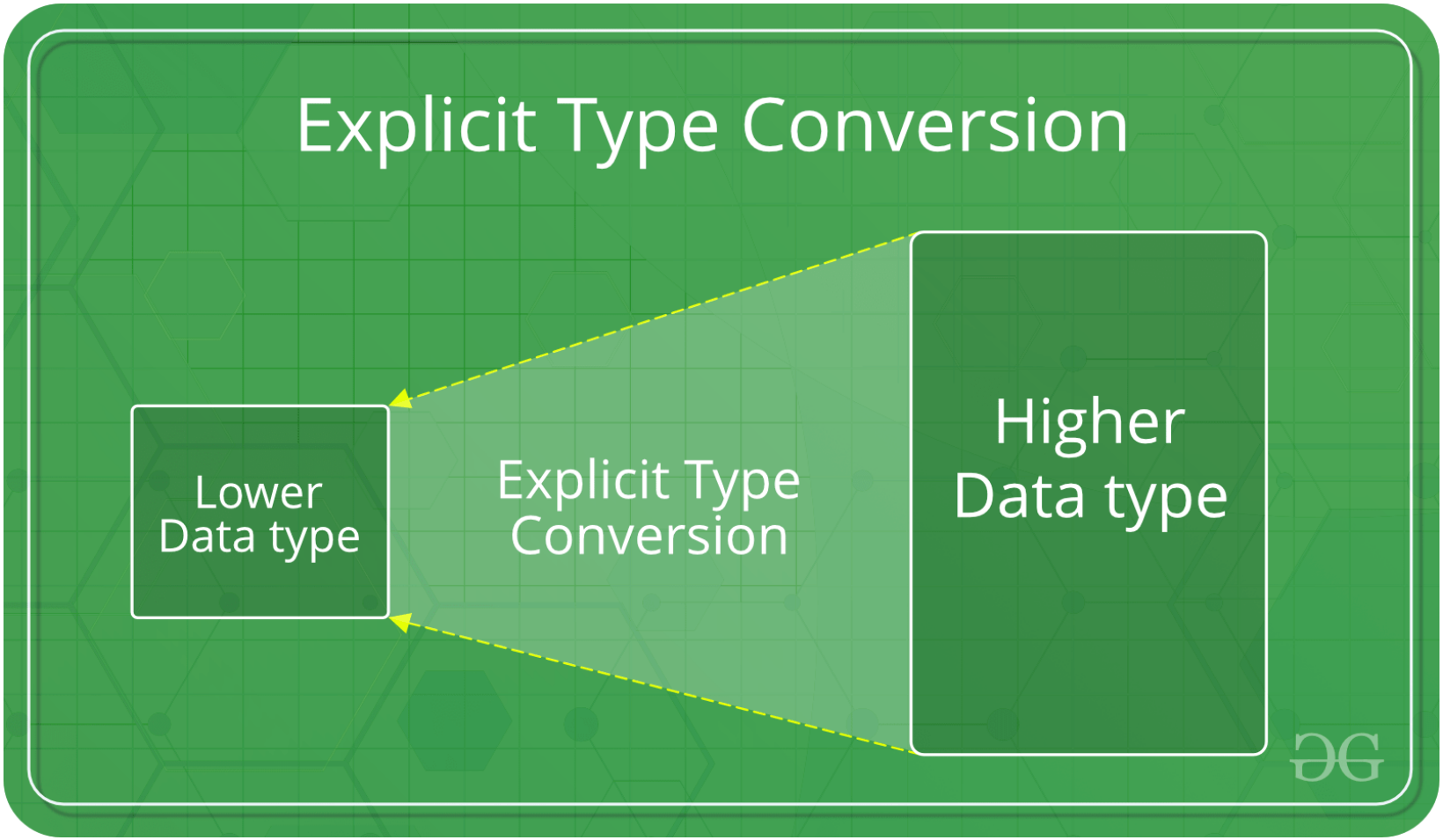
Example:

```
double da = 4.5;
double db = 4.6;
double dc = 4.9;
//explicitly defined by user
int result = (int)da + (int)db + (int)dc;
printf("result = %d", result);
```

Output

result = 12

Type Conversion in Expressions



Type Conversion in Expressions

2) Explicit Type Conversion

- Thus, in the above example we find that the output result is 12 because in the result expression the user has explicitly defined the operands (variables) as integer data type. Hence, there is no implicit conversion of data type by the compiler.
- If in case implicit conversion was used the result would be 13.

Type Conversion in Expressions

2) Explicit Type Conversion

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
   double x = 1.2;

   // Explicit conversion from double to int
   int sum = (int)x + 1;

   printf("sum = %d", sum);

   return 0;
}
```

Output:
sum = 2

Type Conversion in Expressions

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.

Finished

Unit 4