UNIT- 2 OPERATORS AND EXPRESSIONS

UNIT STRUCTURE

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define operators and operands
- define and use different types of operators like arithmetic, logical, relational, assignment, conditional, bitwise and special operators
- learn about the order of precedence among operators and the direction in which each associates.
- use expession in programming
- perform type conversion to get the correct result in expression

2.2 INTRODUCTION

In our previous unit we have learnt to use variables, constants, data types in programming. With the help of operators, these variables, constants and other elements can be combined to form expressions.

In this unit we will discuss various operators supported by C

language such as Arithmetic operators, Relational operators, Logical operators, Assignment operators, Increments, Decrement operators, Conditional operators, Bitwise operators, Special operators etc. Besides, we will learn to carry out type conversion.

2.3 OPERATORS

Operators are special symbols which instruct the compiler to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. The data items that operators act upon are called ***operands***. Operators are used with operands to build expressions. Some operators require two operands, while other act upon only one operand.

C includes a large number of operators which fall into different categories. These are:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increments and Decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

2.3.1 Arithmetic Operators

There are five main arithmetic operators in C language. They are '+' for additions, '-' for subtraction, '*' for multiplication, '/' for division and '%' for remainder after integer division. This '%' operator is also known as *modulus* operator. The operators +, -, * and / all work the same way as they do in other languages. These operators can operate on any built-in data types allowed in C. Some uses of arithmetic operators are :

```
x + y
x - y
- x + y
a * b + c
- a * b
```

Here, ***a, b, c, x*** and ***y*** are *operands*. The modulus (%) operator produces the remainder of an integer division. For example :

```
5 / 2 = 2       (Division)
where as  5 % 2 = 1     (Modulo Division or modulus)
```

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be *integer quantities*, *floating-point quantities* or *characters* (character constants represent integer values, as determined by the computer's character set). Modulus cannot be used with floating-point numbers. It requires that both operands be integers and the second operand be nonzero.

Division of one integer quantity by another is referred to as *interger division*. The result of this operation will be a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a flaoting-point quotient.

Let us consider the following expression.

x = 100+ 2/4;

What will be the actual value of x ?

Is it 100 + 0.5 =100.5

or 102 / 4 = 25.5

Since / has precedence over +, the expression will be evaluated as

100 + 0.5 = 100.5

To avoid ambiguity, there are defined presedence rules for operators in C language. The concept of precedence will be discussed later in this unit.

Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers then such an operation is called ***integer arithmetic***. It always gives an integer as the result. Let, x = 5 and y = 2 be two integer numbers. Then the integer arithmetic leads to the following results:

x + y = 5 + 2 = 7	(Addition)
x – y = 5 - 2 = 3	(Subtraction)
x * y = 5 * 2 = 10	(Multiplication)
x / y = 5 / 2 = 2	(Division)
x % y = 5 % 2 = 1	(Modulus)

In integer division the fractional part is truncated. *Division* gives the quotient, whereas *modulus* gives the remainder of division. Following program is an example to illustrate the above operations.

Program1: Summation, subtraction, multiplication, division and modulo division of two integer numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n1, n2, sum, sub, mul, div, mod;
     clrscr( );
    scanf (“%d %d”, &n1, &n2);          //inputs the operands
     sum = n1+n2;
    printf(“\n The sum is = %d”, sum);    //display the output
    sub = n1-n2;
    printf(“\n The difference is = %d”, sub);
    mul = n1*n2;
    printf(“\n The product is = %d”, mul);
    div = n1/n2;
    printf(“\n The division is = %d”, div);
     mod = n1%n2;
    printf(“\n The modulus is = %d”, mod);
     getch( );
}
```

If we enter 5 for n1 and 2 for n2, then the output of the above program will be :

```
The sum is = 7
The difference is =  3
The product is =  10
The division is = 2
The modulus is = 1
```

Floating point arithmetic

When an arithmetic operation is performed on two real numbers or fractional numbers, such an operation is called ***floating point arithmetic***. The floating point results can be truncated according to the properties requirement. The modulus operator is not applicable for fractional numbers. Let us consider two operands x and y with floating point values 15.0 and 2.0 respectively. Then,

```
x + y = 15.0 +2.0 = 17.0
x – y = 15.0 - 2.0 = 13.0
x * y = 15.0 * 2.0 = 30.0
x / y = 15.0 / 2.0 = 7.5
```

Mixed mode arithmetic

When one of the operands is real and the other is an integer and if the arithmetic operation is carried out on these two operands, then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real, thus 15 / 10.0 = 1.5

2.3.2 Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. Two variables of same type may have a relationship between them. They can be equal or one can be greater than the other or less than the other. We can check this by using ***relational operators***. While checking, the outcome may be *true* or *false*. *"True"* is represented as 1 and *"False"* is represented as 0. It is also by convention that any non-zero value is considered as 1(true) and zero value is considered as 0 (false). For example, we may compare the age of two persons, marks of students, salary of persons, or the price of two items, and so on.

There are four relational operators in C. They are:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

All these operators fall within the same precedence group, which is lower than the arithmetic and unary operators. The associativity of these operators is left to right. There are two *equality operators* associated with the relational operators. They are:

==	equal to
!=	not equal to

For checking equality the double equal sign is used, which is different from other programming languages. The statement ***a == b*** checks whether ***a*** is equal to ***b*** or not. If they are equal, the output will be *true*; otherwise , it will be *false*. The statement ***a = b*** assigns the value of ***b*** to ***a***. For example, if ***b = 10***, then ***a*** is also assigns the value of ***10***.
>, >=, <, <= have precedence over == and !=. Again, the arithmetic operators +, -, *, / have precedence over relational and logical operators. Therefore, in the following statement :

a - 4 > 6

a - 4 will be evaluated first and only then the relation will be checked. So, there is no need to enclose a - 4 within parenthesis.
A simple relational expression contains only one relational operator and takes the following form:

exp1 ***relational operator*** exp2

where *exp1* and *exp2* are expressions, which may be simple constants, variables or combination of them. Some examples of relational expressions and their evaluated values are listed below:

4.5 <= 12	TRUE
-5 > 0	FALSE
10 < 8 + 5	TRUE
5 == 2	FALSE
6! = 2	TRUE

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. Relational expressions are used in decision making statements of C language such as ***if***, ***while*** and ***for*** statements to decide the course of action of a running program. We shall learn about ***if, while*** and ***for*** statement very soon in our next unit.

2.3.3 Logical Operators

Logical operators compare or evaluate logical and relational expressions. C language has the following logical operators:

&& denoting **Logical AND**
|| denoting **Logical OR**
! denoting **Logical NOT**

The logical AND and Logical OR operators are used when we want to test more than one condition and make decisions.

Logical AND (&&)

The logical AND operator is used for evaluating two conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator are true, then the whole compound expression is true. For example:

a > b && x = = 8

The expression to the left is a > b and that on the right is x == 8. The whole expression is true only if both expressions are true i.e., if **a** is greater than **b** and **x** is equal to **8**.

Logical OR (||)

The logical OR is used to combine two expressions and the condition evaluates to true if any one of the two expressions is true. For example:

a < m || a < n

The expression evaluates to true if any one of the expressions **a<m** and **a<n** is true or if both of them are true. It evaluates to true if **a** is less than either **m** or **n** and when **a** is less than both **m** and **n.**

Logical NOT (!)

The logical NOT operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words, it just reverses the value of the expression.

For example, ! (x >= y)

This NOT expression evaluates to true only if the value of **x** is neither greater than nor equal to **y** .

2.3.4 Assignment Operators

In C language, there are several different assignment operators. The most commonly used assignment operator is =. The assignment operator(=) evaluates the expression on the right of the operator and substitutes it to the value or variable on the left of the operand. For example:

x = a + b ;

In the above statement, the value of a + b is evaluated and substituted to the variable x. Let us consider the statement x = x + 1; This will have the effect of incrementing the value of x by 1. This can also be written as x + = 1;

The commonly used *shorthand assignment operators* are as follows:

a = a + 1	is same as	a += 1
a = a – 1	is same as	a -= 1
a = a * (n+1)	is same as	a *= (n+1)
a = a / (n+1)	is same as	a /= (n+1)

a = a % b is same as a %= b

The assignment operators =, + =, - =, * =, / =, % = , have the same precedence than the arithmetic operators. Therefore, the arithmetic operations will be carried out first before they are used to assign the values. C language allows multiple assignments in the following form:

identifier1 = identifier2 = identifier3 = ...= expression

For example, a = b = c = 50;

The assignment operator = and the equality operator == are distinctly different. The assignment operator is to assign a value to an identifier, whereas the equality operator is used to determine if two expressions have the same value.

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left. The entire assignment expression will then be of this same data type. Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example:

- A floating-point value may be truncated if assigned to an inte ger identifier. For example,

```
int n;
n = 5.5;
```

The above expression will cause the integer 5 to be assigned to i.

- A double-precision value may be rounded if assigned to an integer identifier.
- An integer quantity may be altered if assigned to a shorter integer identifier or to a character identifier.

Example: Let us consider i , j are two integer variables and p, q are floating-point variables whose values are

```
i = 8;
j = 5;
p = 4.5;
q = -2.75;
```

Several assignments that make use of these variables are shown below:

Expression	Shorthand expression	Final value
i = i + 2;	i += 2;	10
j = j * (i - 2);	j * = (i - 2) ;	30
p = p / 2 ;	p /= 2;	2.25
i = i % (j - 2);	i %= (j - 2);	2
p = p - q ;	p - = q;	7.25

Program 2: Calculate the sum and average of five numbers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      float a,b,c,d,e,sum,avg;
       clrscr( );
      printf("Enter the five numbers:\n ");
      scanf("%f%f%f%f%f ", &a,&b,&c,&d,&e);
       sum=a+b+c+d+e;
      avg=sum/5.0;
       printf("\n\nSum is = %f ",sum);
      printf("\nAverage is = %f ",avg);
       getch( );
}
```

If we enter 4,10,12, 3 and 6, then the output of the above program will be :

```
Enter the five numbers:
4       10      12      3       6

Sum is = 35.00
Average is = 7.00
```

EXERCISE

Q. What is the output of the following program?

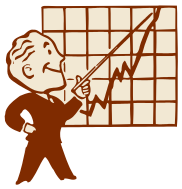
```
#include<stdio.h>
void main()
  {
        int i=15, j=4, m, n;
         m = i > 9;
         n = j > 2 &&  j  != 2;
         printf("m=%d n=%d", m,n);
  }
```

LET US KNOW

Unary Operators

The operator that acts upon a single operand to produce a new value is called ***unary operator***. Unary operators usually precede their single operands, though some unary operators are written after their operands. Unary minus operation is distinctly different from the arithmetic operator which denotes subtraction (-). The subtraction operator requires two separate operands. All unary operators are of equal precedence and have right-to-left associativity. Following are some examples of the use of unary minus operation:

```
-145        // unary minus is followed by an integer constant
-0.5        //unary minus is followed by an floting-point constant
- a                //unary minus is followed by a variable 'a'
-5 *(a + b) //unary minus is followed by an arithmetic expresion
```



CHECK YOUR PROGRESS

1. Choose the correct option:

   (i) The shorthand expression for x = x + 10 is:

   a) x += 10; b) +x = 10; c) x =+ 10; d) x = 10+;

   (ii) What is the value of *sum* for the expression

   sum = 5 + 3 * 4 - 1 % 3;

   a) 31 b) 8 c) 7 d) 16

   (iii) The expression i=30*10+27 evaluates to

   a) 327 b) -327 c) 810 d) 0

2. State whether the following expressions are true or false.

   (i) The modulus operators % can be used only with integers.

   (ii) The modulo division operator produces the remainder of an integer division

   (iii) 10 % 3 yields a result of 3.

   (iv) Unary operator requires more than one operands.

   (v) If both the expressions to the left and to the right of the && operator is true, then the whole compound expression is true.

2.3.5 Increment and Decrement Operators

C language contains two increment and decrement operators which are present in postfix and prefix forms. Both forms are used to increment or decrement the appropriate variables. The increment and decrement operators are one of the unary operators which are very useful in C language.

Pointer:
A pointer is a variable which holds the address of another variable

- **Increment operator ++**

The increment operator (++) adds 1 to the value of an operand or if the operand is a pointer then increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. We can put ++ before or after the operand. These operators are used in a program as follows:

```
+ + i ;    (prefix form)
or  i + +;   (postfix form)
```

The statement + + i; is equivalent to i = i + 1; or i + = 1;

In the above statements, *i* is an integer type variable. ++ i and i ++ means the same thing when they form statements independently. But they behave differently when they are used in expressions on right-hand side of an assignment statement. If ++ appears before the operand (***prefix*** form), the operand is incremented first and then used in the expression. If we put ++ after the operand (***postfix*** form), the value of the operand is used in the expression before the operand is incremented. Let us consider the following statements:

```
i = 5;
j = ++i;     // pre increment
printf("%d%d", i, j);
```

In this case, the value of j and i would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. Thus, 1 is added to i and the value of i becomes 6. Then this incremented value of i is assigned to j and the value of j also becomes 6. If we rewrite the above statements as

```
i = 5;
j = i++;    // post increment
```

then the value of j would be 5 and i would be 6. This is because a postfix operator first assigns the value to the variable on the left and then increments the operands. Thus, 5 is first assigned to j and then

i is incremented by 1.

- **Decrement operator - -**

The decrement operator (- -) subtracts 1 from the value of a operand or if the operand is a pointer, it decreases the operand by the size of the object to whcih it points. The operand reveives the result of the decrement operation.

Like increment operator, decrement operator - - can be put before or after the operand. If it appears before the operand, the operand is decremented and the decremented value is used in the expression. But if - - appears after the operand then the current value of the operand is first used in the expression and then the operand is decremented.

- - i ; (prefix form)

or i - - ; (postfix form)

The statement - - i; is equivalent to i = i -1; or i - = 1;

Rules for + + and - - Operators

Increment and decrement operators are unary operators and they require single variable as their operand.

- When postfix ++ (or - -) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented
- When prefix ++ (or - -) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associativity of ++ and - - operators are the same as those of unary + and -.

2.3.6 Conditional Operator

The conditional operator is also termed as *ternary operator* and is denoted by (? :) . The syntax for the conditional operator is as follows:

expression1 ? expression2 : expression3

When evaluating a conditional expression, **expression1** is evaluated first. If **expression1** is true (i.e., if its value is nonzero), then **expression2** is evaluated and this becomes the value of the expres-

sion. However, if **expression1** is false (i.e., if its value is zero) , **expression3** is evaluated and this becomes the value of the conditional expression. Only one of the expressions is evaluated.
For example :

```
a = 10;
b = 15;
x = (a > b) ? a : b ;
```

Here **x** will be assigned to the value of **b**. The condition follows that the expression is false; therefore **b** is assigned to **x**.

Program4: Program to illustrate the use of conditional operator.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int age;
     clrscr( );
    printf("Enter your age in years: ");
    scanf("%d",&age);
   (age>=18)? printf("\nYou can vote\n") : printf("You can't vote");
     getch( );
}
```

Output :
```
Enter your age in years:
26
You should vote
```

If we run the program again and enter *age=15,* then the output will be:

```
Enter your age in years:
15
You cannot vote
```

Program5: Program for finding the larger value of two given values using conditional operator

```
#include<stdio.h>
void main()
{   int i,j, large;
   printf ("Enter 2 integers : "); //ask the user to input 2 numbers
   scanf("%d %d",&i, &j);
   large = i > j ? i : j;          //evaluation using conditional operator
   printf("The largest of two numbers is %d \n", large);
}
```

Output : Enter 2 integers : 14 25
The largest of two numbers is 25

Program6: Conversion of centigrate to fahrenheit and vice-versa

```
#include<iostream.h>
void main()
{
    float c, f;
    printf("Enter temperature in celcius: );
    scanf("%f", &c);
    f=1.8 * c+32;
   printf("\nEquivalent Fehrenheit will be: %f",f);
    printf("\nEnter temperature in Fehrenheit: );
    scanf("%f", &f);
    c=(f-32)/1.8;
   printf("\nEquivalent Celsius will be: %f",c);
}
```

2.3.7 Bitwise Operators

The bitwise operatos are used for testing, complementing or shifting bits to the right or left. A bitwise operator operates on each bit of data. Bitwise operators may not be applied to a float or double. The bitwise operators with their meaning are listed below:

&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

Bitwise Logical Operators :

The logical bitwise operators are similar to the Boolean or Logical operators, except that they operate on every bit in the operand(s). For instance, the bitwise AND operator (&) compares each bit of the left operand to the corresponding bit in the right hand operand. If both bits are 1, a 1 is placed at that bit position in the result. Otherwise, a 0 is placed at that bit position.

Bitwise AND (&) Operator :

The bitwise AND operator performs logical operations on a bit-by-bit level using the following truth table:

Bit x of operator1	Bit x of of operator2	Bit x of result
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table for the bitwise AND (&) operator

Let us consider the following program segment for understanding AND(&) operation.

```
void main( )
{    unsigned int a = 60;          // a= 60 = 0011 1100
     unsigned int b = 13;          // b= 13 = 0000 1101
     unsigned int c = 0;
      c = a & b;                    //c= 12 = 0000 1100
      printf("%d",c);
}
```

The output will be 12

Bitwise OR (|) operator :

The bitwise OR operator performs logical operations on a bit-by-bit level using the following truth table:

Bit x of operator1	Bit x of operator2	Bit x of result
0	0	0
0	1	1
1	0	1
1	1	1

Truth Table for the bitwise OR (|) operator

The bitwise OR operator (|) places a 1 in the corresponding value's bit position if either operand has a bit **set** (i.e.,1) at the position. Bitwise OR(|) operation can be understood with the following example:

```
void main( )
{   unsigned int a = 60;          // 60 = 0011 1100
    unsigned int b = 13;          // 13 = 0000 1101
    unsigned int c = 0;
    c = a | b;                    // 61 = 0011 1101
 }
```

Bitwise exclusive OR (^) :

The bitwise exclusive OR(XOR) operator performs logical operations on a bit-by-bit level using the following truth table:

Bit x of operator1	Bit x of operator2	Bit x of result
0	0	0
0	1	1
1	0	1
1	1	0

Truth Table for the exclusive OR(^)

The bitwise exclusive OR(^) operator sets a bit in the resulting value's bit position if either operand (but not both) has a bit **set** (i.e.,1)at the position. Bitwise exclusive OR(^) operation can be understood with the following example:

```
void main( )
{   unsigned int a = 60;     // 60 = 0011 1100
    unsigned int b = 13;     // 13 = 0000 1101
    unsigned int c = 0;
    c = a ^ b;               // 49 = 0011 0001
 }
```

Bitwise Complement (~)

The bitwise complement operator (~) performs logical operations on a bit-by-bit level using the following truth table:

bit x of op2	result
0	0
0	1

Truth table for the ~, Bitwise Complement

The bitwise complement operator (~) reverses each bit in the operand.

Bitwise Shift Operators :

C provides two bitwise shift operators, bitwise left shift (**<<**) and bitwise right shift (**>>**), for shifting bits left or right by an integral number of positions in integral data. Both of these operators are binary, and the left operand is the integral data whose bits are to be shifted, and the right operand, called the shift count, specifies the number of positions by which bits need shifting. The shift count must be nonnegative and less than the number of bits required to represent data of the type of the left operand.

Left-Shift (<<) operator

The left shift operator shift bits to the left. As bits are shifted toward high-order positions, **0** bits enter the low-order positions. Bits shifted out through the high-order position are lost. For example, let us consider the following declaration:

unsigned int Z = 5;

and Z in binary is 00000000 00000101 when 16 bits are used to store integer values.

Now if we apply left-shift, then

Z << 1 is 00000000 00001010 or 10 decimal

and Z << 15 is 10000000 00000000 or 32768 decimal.

Left-Shift is useful when we want to MULTIPLY an integer (not floating point numbers) by a power of 2. The operator, takes 2 operands like this:

a << b

This expression returns the value of a multiplied by 2 to the power of b.

For example, let us consider 4 << 2. In binary, 4 is 100. Adding 2 zeros to the end gives 10000, which is 16, i.e., $4*2^2 = 4*4 = 16$.

Similarly, 4 << 3 can be evaluated by adding 3 zeros to get 100000, which is $4*2^3 = 4*8 = 32$.

Shifting once to the left multiplies the number by 2.Multiple shifts of 1 to the left results in multiplying the number by 2 over and over again. In other words, multiplying by a power of 2. Some examples are:

$5 << 3 = 5*2^3 = 5*8 = 40$
$8 << 4 = 8*2^4 = 8*16 = 128$
$1 << 2 = 1*2^2 = 1*4 = 4$

Right-Shift (>>) operator

The right shift operator shifts bits to the right. As bits are shifted towards low-order positions, **0** bits enter the high-order positions, if the data is unsigned. If the data is signed and the sign bit is **0**, then **0** bits also enter the high- order positions. However, if the sign bit is **1**, the bits entering high-order positions are implementation-dependent. On some machines **1**s, and on others **0**s, are shifted in. The former type of operation is known as the arithmetic right shift, and the latter type the logical right shift. For example,

unsigned int Z = 40960;

and Z in binary 16-bit format is 10100000 00000000

Now, if we apply right-shift, then

Z >> 1 is 01010000 00000000 or 20480 decimal

and Z >> 15 is 00000000 00000001 or 1 decimal

In the second example, the **1** originally in the fourteenth bit position has dropped off. Another right shift will drop off the **1** in the first bit position, and **Z** will become zero. Bitwise Right-Shift does the opposite, and takes away bits on the right.

2.3.8 Other Operators

There are some other useful operators supported by C language. These are: *comma* operator, *sizeof* operator, *member selection operator* (. and ->), *pointer operators* (* and &) etc. Here we will discuss *comma* and *sizeof* operators. The other two will be covered in later unit.

The Comma Operator

The ***comma*** operator can be used to link related expressions together. The comma allows for the use of multiple expressions to be used where normally only one would be allowed.

The comma operator forces all operations that appear to the left to be fully completed before proceeding to the right of comma. This helps eliminate side effects of the expression evaluation.

```
num1 = num2 + 1, num2 = 2;
```

The comma ensures that **num2** will not be changed to a 2 before **num2** has been added to 1 and the result placed into **num1**. Some examples of comma operator are:

In *for* loops:

```
for (n=1, m=15, n <=m; n++,m++)
```

In *while* loops:

```
while (c=getchar(), c != '15')
```

Exchanging values :

```
temp = x, x = y, y = temp;
```

The concept of loop will be discussed very soon in the next unit.

Program7: Swap (interchange) two numbers using a temporary variable.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,temp;
    clrscr();
    printf("\nEnter the two integer numbers:");
    scanf("%d%d",&a,&b);
    printf("\nEntered numbers are.:");
    printf("%d%8d",a,b);
    temp=a,a=b,b=temp;        // comma operator is used
    printf("\n\nSwapped numbers are:%d%8d",a,b);
    getch();
}
```

Output : (Suppose we have entered 2 and 4)

```
Enter the two integer numbers.: 2    4
Entered numbers are: 2       4
Swapped numbers are: 4       2
```

The Sizeof Operator

The **sizeof** operator returns the physical size, in bytes of the data item for which it is applied. It can be used with any type of data item except bit fields.

When **sizeof** is used on a character field the result returned

is 1 (if a character is stored in one byte). When used on an integer the result returned is the size in bytes of that integer.
For example:

```
`       s = sizeof (sum);
        t = sizeof (long int);
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

For example:

```
s = sizeof (sum);
t = sizeof (long int);
```

The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

Program8: Program that employs different kinds of operators like arithmetic, increment, conditional and sizeof operators.

```
#include<stdio.h>
#include<conio.h>
void main( )
{     int a, b, c, d,s;
      clrscr( );
      a = 20;
      b = 10;
      c = ++a-b;
      printf (“a = %d, b = %d, c = %d\n”, a,b,c);
      d=b++ + a;
      printf (“a = %d, b = %d, d = %d\n, a,b,d);
      printf (“a / b = %d\n, a / b);
      printf (“a % b = %d\n, a % b);
      printf (“a *= b = %d\n, a *= b);
      printf (“%d\n, (c < d) ? 1 : 0 );
      printf (“%d\n, (c > d) ? 1 : 0 );
      s=sizeof(a);
      printf(“\nSize is: %d bytes”,s);
}
```

Output : a=21 b=10 c=11
a=21 b=11 d=32
a/b=1

a%b=10
a*=b=231
1
0
2 bytes

The increment operator ++ works when used in an expression. In the statement c = ++a – b; new value a = 16 is used thus giving value 6 to C. That is **a** is incremented by 1 before using in expression However in the statement d = b++ + a; the old value b = 10 is used in the expression. Here **b** is incremented after it is used in the expres sion.

EXERCISE

Q. Write a program that reads a floating-point number and then display the right-most digit of the integral part of the number.

CHECK YOUR PROGRESS

3. Find the output of the following program segment?

(a)
```
void main(){     int x = 50;
                 printf("%d\n",5+ x++);
                 printf("%d\n",5+ ++x);   }
```

(b)
```
void main(){     int x, y;
                 x = 50;
                 y =100;
                 printf("%d\n",x+ y++);
                 printf("%d\n",++y -3);    }
```

(c)
```
void main(){     int s1,s2;
                 char c='A';
                 float f;
                 s1=sizeof(c);
                 s2=sizeof(f);
                 printf("ASCII value of 'A' is %d",c);
printf("\nSize of s1 and s2 in bytes:%d%8d",s1,s2);}
```

4. Find the output of the following C program:

```
void main( )
{    int a,b,c;
     a=b=c=0;
    printf("Initial value of a,b,c :%d%d%d\n",a,b,c);
     a=++b + ++c;
     printf("\na=++b + ++c=%d%d%d\n",a,b,c);
     a= b++ + c++;
     printf("\na=b++  + c++= %d%d%d\n",a,b,c);
     a=++b + c++;
     printf("\na=++b + c++= %d%d%d\n",a,b,c);
     a = b- - + c - -;
     printf("\na=b-- +c --= %d%d%d\n",a,b,c);
}
```

5. Choose the correct option:

(i) If i=6, and j=++i, the the value of j and i will be

(a) i=6,j=6 (b) i=6, j=7 (c)i=7,j=6 (d)i=7,j=7

(ii) If the following variables are set to the values as shown below, then what will be the expression following it?

```
answer=2;
marks=10;
!("answer<5")&& (marks>2))
```

(a) 1 (b) 0 (c) -1 (d) 2

6. Write a C program to find the area of a triangle when base and height are given.

7. What will be output of the following code?

```
#include<stdio.h>
void main()
{
        int n;
        n = 20;
        printf("\nValue of n : %d", sizeof(n));
        printf("\nSizeof n: %d", sizeof(n));
}
```

2.4 PRECEDENCE AND ASSOCIATIVITY

There are two important characteristics of operators which determine how operands group with operators. These are ***precedence*** and ***associativity***.

The operators have an order of precedence among themselves. This order of precedence dictates in what order the operators are evaluated when several operators are together in a statement or expression. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses. Also, with each operator is an associativity factor that tells in what order the operands associated with the operator are to be evaluated. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence.

For example, ***operator precedence*** is why the expression 6 + 4 * 3 is calculated as 6 + (4 * 3), giving 18, and not as (6 + 4) *3, giving 30. We say that the multiplication operator (*) has higher precedence than the addition operator (+), so the multiplication must be performed first. ***Operator associativity*** is why the expression 8 - 4 - 2 is calculated as (8 - 4) - 2, giving 2, and and not as 8 - (4 - 2), giving 6. We say that the subtraction operator (-) is left associative, so the left subtraction must be performed first. When we cannot decide by operator precedence alone in which order to calculate an expression, we must use associativity.

The following table lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied. R indicates *Right* and L indicates *Left*.

Operator category	**Operators**	**Associativity**
Unary Operator	-- ++ ! sizeof(type)	R to L
Arithmetic multiply, Divide and remainder	* / %	L to R
Arithmetic add,subtract	+ -	L to R
Relational Operators	< <= > >=	L to R
Equality Operators	== !=	L to R
Logical AND	&&	L to R
Logical OR	||	L to R
Conditional operator	? :	R to L
Assignment operator	= += -+ *= /+ %=	R to L

Precedence and Associativity of operators

In the following statements, the value 10 is assigned to both ***a*** and ***b*** because of the right-to-left associativity of the = operator. The value of ***c*** is assigned to ***b*** first, and then the value of ***b*** is assigned to ***a***.

```
b = 9;
c = 10;
a = b = c;
```

In the expression

```
a + b * c / d
```

the ***** and **/** operations are performed before **+** because of precedence. ***b*** is multiplied by ***c*** before it is divided by ***d*** because of associativity.

2.5 EXPRESSIONS

C Expressions are based on algebra expressions - they are very similar to what we learn in Algebra, but they are not exactly the same. An expression is a combination of variables, constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Here are some examples of expressions:

```
15       // a constant
i        // a variable
i+15     // a variable plus a constant
(m + n) * (x + y)
```

The following program illustrates the effect of presence of parenthesis in expressions.

Program9:

```
#include<stdio.h>
#include<conio.h>
void main( )
   float a, b, c x, y, z;
   a = 9;
   b = 12;
   c = 3;
   x = a – b / 3 + c * 2 – 1;
   y = a – b / (3 + c) * (2 – 1);
   z = a – ( b / (3 + c) * 2) – 1;
   printf (“x = %fn”,x);
   printf (“y = %fn”,y);
```

```
    printf ("z = %fn",z);
}
```

Output

x = 10.00
y = 7.00
z = 4.00

Rules for evaluation of expression

- First parenthesized sub expression left to right are evaluated.
- If parenthesis are nested, the evaluation begins with the inner most sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When Parenthesis are used, the expressions within parenthesis assume highest priority.

2.6 TYPE CONVERSION

The **type conversion** or **typecasting** refers to changing an entity of one data type into another i.e., values of one type can be converted to a value of another type. For example, the integer 355 can be converted to the string "355". Again, 355 + 5 will give 400, "355" + "5" will give "3555".

The language C allows programmer to perform typecasting by placing the type name in parentheses and placing this in front of the value. The form of the cast data type is :

(type) expression

For example,

```
(float)25       // Gives the float 25.0
(int)5.8        // Gives the int 5
(string)5.8     // Gives the string "5.8"
(float)"4.3"    // Gives the float 4.3
```

Let us consider the case where we want to divide two integers a/b, where the result must be an integer. However, we may want to force the output to be a float type in order to keep the fraction part of the

division. The typecast operator is used in such a case. It will do the conversion without any loss of fractional part of data.

Program10:

```
#include<stdio.h>
void main()
{
        int a,b;
        a=3,b=2;
        printf("\n%f",(float)a/b);
}
```

The output of the above program will be 1.500000. This is because data type cast (float) is used to force the type of the result to be of the type float.

From the above it is clear that the usage of typecasting is to make a variable of one type act like another type for one single operation. So by using this ability of typecasting it is possible to create ASCII characters by typecasting integer to its character equivalent. Typecasting is also used in arithmetic operation to get correct result. This is very much needed in case of division when integer gets divided and the remainder is omitted. In order to get correct precision value, one can make use of typecast as shown in example above. Another use of the typecasting is shown in the example below:

For instance:

```
void  main()
{
    int a = 5000, b = 7000 ; long int c = a * b ;
 }
```

Here, two integers are multiplied and the result is truncated and stored in variable c of type long int. But this would not fetch correct result for all. To get a more desired output the code is written as

```
long int c = (long int) a * b;
```

Though typecast has so many uses one must take care about its usage since using typecast in wrong places may cause loss of data like, for instance, truncating a *float* when typecasting to an *int*.

Some conversions are done automatically. For example, in the expression "Hello" + 15 the integer 15 will be automatically converted to the string "15", before the two strings are concatenated, giving the result "Hello15".

CHECK YOUR PROGRESS

8. State whether the following expressions are true or false.

(i) Conditional operator (? :) has right to left associativity.

(ii) Logical OR operator has right to left associativity

(iii) C permits mixing of constants and variables of different types in an expression.

(iv) Precedence dictates in what order the operators are evalu ated when several operators are together in a statement or expression.

(v) A typecast is used to force a value to be of a particular variable type.

2.7 LET US SUM UP

Operators form expressions by joining individual constants, variables,array elements etc. C language includes a large number of operators which fall into different categories. In this unit we have seen how arithmetic operators, assignment operators, unary operators, relational and logical operators, the conditional operators are used to form expressions.The data items on which operators act upon are called ***operands***. Some operators require two operands while others require only one operand. A list of operators with their meaning are given below:

Arithmetic Operator

Operator	Description
*	multiplication
/	division
%	modulo division
+	addition
-	subtraction

Logical Operator

Operator	Description
!	NOT
&&	AND
||	OR

Relational Operator

Operator	Description
<	less than
>	greater than
>=	greater than or equal
==	equal to
!=	not equal

Bitwise Operators

Operator	Description
~	One's complement
<<	Left shift
>>	Right shift
&	Bitwise AND
^	Bitwise XOR

Assignment Operator

The Assignment Operator(=) evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression. For example: sum = n1+n2 ;
value of *n1* and *n2* are added and the result is assigned to the variable *sum*.

Increment and Decrement

There are two formsof increment and decrement operators. These are:

Operator	Description	Example
++	increment	a++ (post increment)
		++a (pre increment)
--	decrement	a-- (post decrement)
		--a (pree decrement)

The Conditional Operator

It works on three values. The conditional operator is used to replace *if-else* logic in some situations. It is a two-symbol operator **?:** with the format:

result = condition ? expression1 :expression2;

Comma Operator

We can use the comma operator(,) available in c language, to build a compound expression by putting several expressions inside a set of parentheses. The expressions are evaluated from left to right and the final value is evaluated last.

sizeof Operator

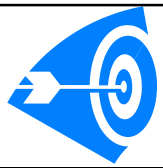
The **sizeof** operator returns the physical size in bytes of the data item for which it is applied. It can be used with any type of data item except bit fields. The general form is: s = sizeof (item);

Expressions in C are syntactically valid combinations of operators amd operands that compute to a value determined by the priority and associativity of the operators.

Converting an expression of a given type into another type is known as ***type-casting*** or ***type conversion***. Type conversions depend on the specified operator and the type of the operand or operators.

2.8 FURTHER READINGS

1. Venugopal, K L and Prasad S R : *Mastering C,* Tata McGraw-Hill publication.
2. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
3. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.



2.9 ANSWERS TO CHECK YOUR PROGRESS

1.(i) (a) x = x +10 (ii)(d) 16 (iii) (a) 327

2. (i)True (ii) True (iii) False (iv) False (v) True

3. (a) 55 (b)150
99 57
(c) ASCII value of 'A' is 65
Size of s1 and s2 in bytes: 1 4

4. Initial value of a,b,c : 0 0 0
a= ++b + ++c = 2 1 1
a= b++ + c++ = 2 2 2
a= ++b + c++ = 5 3 3
a= b-- + c -- = 6 2 2

5. (i)(d)i=7,j=7 (ii) (b) 0

6. //area of a triangle given the base and height

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float b, h;
     float area;
    printf("Enter the base: ");
    scanf("%f", &b);
    printf("\nEnter the height:");
    scanf("%f", &h);
```

```
        area= b * h / 2;
        printf("\nArea of the triangle is %f", area);
    }
```

7. Value of n is: 20
   Size of n is: 2

8. (i) True (ii)False (iii)True (iv)True (v) True

2.10 MODEL QUESTIONS

1.What is an operator ? What are the types of operators that are included in C.
2. What is an operand ? What is the relationship between operator and operand?
3.Describe the three logical operators included in C?
4. Write a C program to compute the surface area and volume of a cube if the side of the cube is taken as input.
5. What is unary operator ? How many operands are associated with a unary operator?
6. What is meant by operator precedence?
7. What is meant by associativity? What is the asslociativity of the arithmetic operators?
8. What will be the output of the following program:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
      printf("The size of char is %d",sizeof(char));
      printf("\nThe size of int is %d",sizeof(int));
      printf("\nThe size of short is %d",sizeof(short));
      printf("\nThe size of float is %d",sizeof(float));
      printf("\nThe size of long is %d",sizeof(long));
      printf("\nThe size of char is %d",sizeof(char));
      printf("The size of double is %d",sizeof(double));
   getch();
}
```

9. List the relational operator and their meaning.
10 . Write programs for computing the volume of a sphere, a cone

and a cylinder. Assume, the dimensions are integers. Use type casting whereever necessary.

11. If a,b,c,d and e are declared using the statement

int a, b, c, d;

What value is assigned to the variablea in the following statement

a = b > c ? c > d ? 12 : d > e ? 13 : 14 : 15 in each of the following cases:

(i) b = 5; c = 15; d = e = 8;

(ii) b = 15; c = 10; d = e = 8;

(iii) b = 15; c = 10; d = e = 20;

(iv) b = c = 9; d = 20; e = 19;

*****