Unit - 2

Chapter

6 C Operators and Expressions

Chapter Outline

- Operators in C
- Arithmetic Operators
- Unary Operators
- Relational Operators
- Logical Operators
- Assignment Operator
  - Compound Assignment Operators
- Conditional Operator (Ternary Operator)
- Bitwise Operators
- Special Operators
  - Comma Operator
  - Size of Operator
- Operator Precedence and Associativity
- Precedence of Arithmetic Operators
- Arithmetic Expressions
- Evaluation of Arithmetic Expressions
- Relational Expressions
- Logical Expressions
- Type Conversion
  - Implicit Type Conversion
  - Explicit Type Conversion
- Library Functions
- Mathematical Functions
- Sample Programs
- Review Question

6.1 Operators in C

In order to perform different kinds of operations, C uses different kinds of **operators.** An operat indicates an operation to be performed on data that yields a value. C is very rich in the use of differe operators. An **operand** is an entity on which operators perform the operations.

Example : a+b

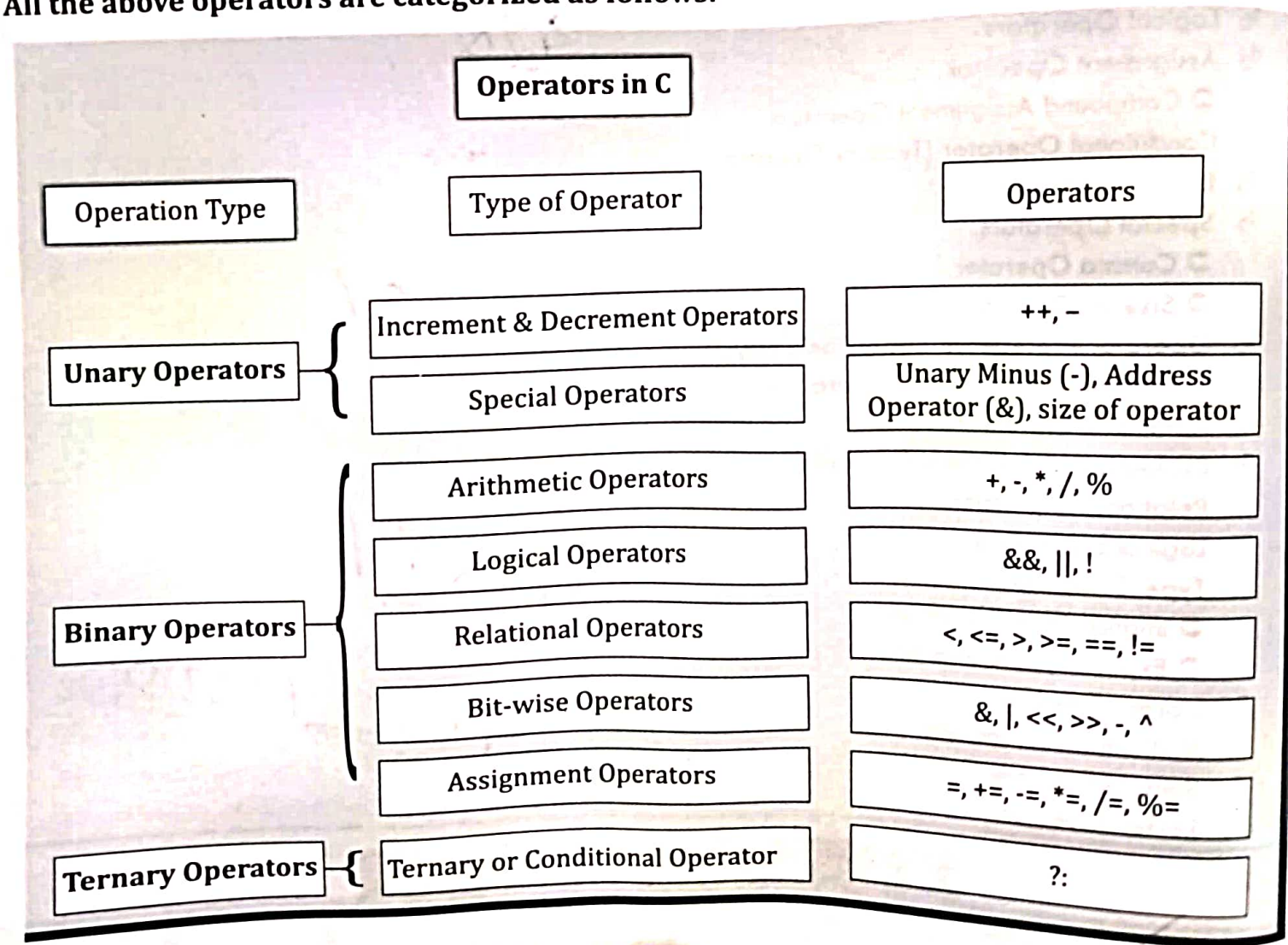
Here, a and b are operands and '+' is an operator. The '+' operator requires two operands to perfo the addition.

Types of C Operators

The different types of operators in C are listed below:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Increment and Decrement Operators
7. Conditional Operator
8. Special Operators (comma, &, *, size of)

All the above operators are categorized as follows:



1. **Unary Operator** : Requires one operand
   Example: ++a
2. **Binary Operator** : Requires two operands
   Example: a+b
3. **Ternary Operator** : Requires three operands
   Example: (a == 5) ? b : c

6.2 Arithmetic Operators

There are five arithmetic operators in C for the purpose of arithmetic operations. In simple words we can say arithmetic operators are used for calculations. The following table shows different arithmetic operators that are used in 'C'.

Arithmetic Operator	Operator Description	Examples
+	Addition	(1) 2 + 3 (2) a + b
-	Subtraction	(1) 5 – 2 (2) c – d
*	Multiplication	(1) 5 * 3 (2) e * f
/	Division	(1) 8 / 2 (2) x / y
%	Modular Division	(1) 8 % 3 (It gives remainder 2) (2) a % b

The above arithmetic operators are used for numerical calculations between two constant values. Arithmetic operators are also called as **Binary Arithmetic Operators.** All these operators requires two operands, i.e., one operand left and another operand right of the operator.

1. **Addition Operator (+):** This operator is used to add the two data values (or) operands that appear on either side of operator (+).

   Example: Sum = a + b;

   The above statement adds the values of two variables **a** and **b** and assigns the result to the another variable called **Sum.**

2. **Subtraction Operator (–):** This operator is used to subtract the one data. value from the other.

   Example: Sub = a – b;

   The above statement subtracts the value of b from a and assigns the result to the variable **Sub.** If a = 10, and b = 6, then sub will have the value 4.

3. **Multiplication Operator (*):** This operator is used to multiply the two data values.

   Example: Mul = a * b;

The above statement multiplies the values of a and b and assigns the result to the variable **Mul.**

4. **Division operator (/):** This operator is used to divide the two data values. The value to the left of '/' is divided by that on the right.

   Example : Div = a / b;

   which divides the value of a by b. If a = 8 and b = 2, then 8/2 = 4 is assigned to the variable **Div.**

5. **Modulus operators (%) :** This operator is used to get the remainder of an integer division.

   Example: Mod = a % b ;

   The variable Mod contains the remainder of a/b. For example 10%4 yields the value 2.

   Example: Mod = 13 %4

   It assigns the result 1 to the variable Mod, which is the remainder of the integer division of 13 by 4.

Note

We cannot use % operator on float or double. For example, 13.5% 4.2 is not possible.

The following program demonstrates the usage of arithmetic operators.

Program 1 **To demonstrate Arithmetic Operators**

```
#include <stdio.h>
main()
{
   int a , b
   int sum, sub, mul, div, mod;
   printf(" Enter two numbers: ");
   scanf("%d %d",&a,&b)'
   sum = a+b;
   sub = a-b;
   mul = a*b;
   div = a/b;
   mod = a %b;
   printf("\n The sum of a and b = %d", sum);
   printf("\n The subtraction of b from a = %d", sub);
   printf("\n The product of a and b = %d", mul);
   printf("\n The division of a by b = %d", div);
   printf("\n The remainder of a/b = %d", mod);
}
```

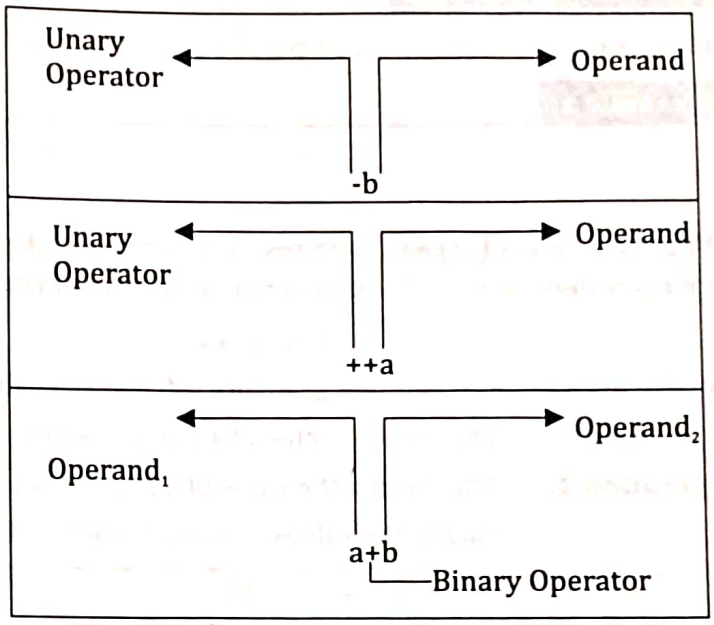
Output

```
Enter two numbers: 10 20
The sum of a and b = 30
The subtraction of b from a = 10
The product of a and b = 200
The division of a by b = 2
The remainder of a/b = 0
```

6.3 Unary Operators

The operators requiring only one operand are known as **Unary operators.** There are three unary operators in C. They are listed below.

Unary Operator	Operator Description	Examples
-	Unary minus	- 10 ;
		- b ;
++	Increment	a++ ;
		++a ;
--	Decrement	a- - ;
		- -a ;



We know that binary operator requires two operands. But unary operator requires only one operand.

1. **Unary Minus(–)**

   This operator is used to represent a negative number. Any number or variable preceded with a sign(–) is considered as negative number. Placing the plus(+) sign before the number is an optional. For example

   b = 5

   a = – b;

   The above statement assigns – 5 to the variable **a.**

2. **Increment and Decrement Operators (++, – –):**

   These operators are used to increase or decrease the value of an operand by one. ++ adds 1 to its operand and – – subtracts 1 from its operand. Two mostly commonly used assignment statements for incrementing and decrementing the value of an operand (suppose x is an operand) by 1 are x = x+1 and x = x–1 respectively. C provides fast and efficient way of doing this by (++) and (– –) operators.

```
x = x +1  is same as x++ (or) ++x.
x = x - 1 is same as x-- (or) --x.
```

Example

```
if a = 5,
   a++ ;          // Now a becomes 6
   ++a ;          // Now a becomes 7
   --a ;          // Now a becomes 6
   a-- ;          // Now a becomes 5
```

An increment operator or a decrement operator and an operand together form an increment or decrement expression. Depending on the order in which operator and operand are placed, these expressions are classified as **postfix** and **prefix expressions**.

Postfix Expression

In this ++ or – – operator are placed after the operand.

Example 1

```
a++;
a-- ;
```

When we use a++ (or) ++a, both result in increment of value by 1. Therefore both postfix and prefix expressions are equivalent to a = a+1. An assignment statement consisting of postfix expression such as

```
b = a ++;
```

results in two operations being performed. If initial value of a is 5, then the two operations are.

Operation 1: The present value of a is assigned to b.

Operation 2: After which the value of a is incremented by 1.

Finally the values of a and b are 6 and 5 respectively.

```
int a = 5, b;
b = a++;
operation 2 (incrementing a value by 1)
operation 1 (assigning value of a to b)
```

Operation1 is performed first and then operation2 is performed. Accordingly, the value of a (a = 5) is assigned to b and then a value is incremented by 1 (a = 6).

Example 2

Consider postfix decrement expression shown below.

```
int a = 5, b;
b = a--;
```

First it assigns the value of a (a = 5) to b, then the value of a is decremented by 1. Finally a and b contains the values 4 and 5 respectively.

Prefix Expression

In this ++ or – – operator are placed before the operand. For example, ++a or – –a; when prefix expression is present in an assignment statement, then again two operations are carried out.

Example 1

```
int a = 5, b;
b = ++a;
```

Operation 1: First increment the value of a by 1 (a becomes 6).

Operation 2: After which, the updated value of a is assigned to b.

Finally, the values of a and b are 6 and 6 respectively.

Example 2

```
int a = 5, b;
b = --a;
```

Operation 1: First decrement the value of a by 1 (a becomes 4).

Operation 2: After which, the updated value of a (a = 4) is assigned to b.

Finally, the values of a and b are 4 and 4 respectively.

Program 2 To demonstrate the post-increment and pre-increment

```
main()
{
    int a, b, x = 10, y = 20;
    a = x*y++;
    b = x*++y;
    printf("a = %d, b = %d \n", a, b);
}
```

Output

```
a = 200, b = 220
```

Explanation

The statement a = x*y++ gives the result 200 because first x*y is performed which is 200. After x*y, y gets increased to 21. So, 'a' contains 200. The next statement b = x*++y, first it increases the values of y to 22 and then it perform x*y, so, the result is 10*22 = 220 is assigned to the variable b.

6.4 Relational Operators

In the term relational operators, **relational** refers to the relationships that expressions can have with one another. These operators are used to distinguish between two expressions depending on their relations. If the relation is true then it returns the value 1 otherwise 0 for false relation. In simple words, a relational operator compares two relational expressions or quantities and the result of which is either True (1) or False (0). The relational operators along with their description and examples are shown below.

Relational Operator	Description	Example	Result
>	Greater than	8 > 5	True (1)
<	Less than	8 < 5	False (0)
<=	Less than or equal to	8 <= 8	True (1)
>=	Greater than or equal to	8 >= 10	False (0)
==	Equal to	5==5	True (1)
!=	Not equal to	3! = 3	False (0)

Syntax:

The syntax of relational expression consists of two expressions separated by a relational operator.

exp1 <relational operator> exp2

Where, **exp1** and **exp2** are two expressions and it can also be a variable or a constant.

An increment operator or a decrement operator and an operand together form an increment or decrement expression. Depending on the order in which operator and operand are placed, these expressions are classified as **postfix** and **prefix expressions**.

Postfix Expression

In this ++ or -- operator are placed after the operand.

Example 1

```
a++;
a-- ;
```

When we use a++ (or) ++a, both result in increment of value by 1. Therefore both postfix and prefix expressions are equivalent to a = a+1. An assignment statement consisting of postfix expression such as

```
b = a ++;
```

results in two operations being performed. If initial value of a is 5, then the two operations are.

Operation 1: The present value of a is assigned to b.

Operation 2: After which the value of a is incremented by 1.

Finally the values of a and b are 6 and 5 respectively.

```
int a = 5, b;
b = a + +
    ↑   ⟶ operation 2 (incrementing a value by 1)
operation 1 (assigning value of a to b)
```

Operation1 is performed first and then operation2 is performed. Accordingly, the value of a (a = 5) is assigned to b and then a value is incremented by 1 (a = 6).

Example 2

Consider postfix decrement expression shown below.

```
int a = 5, b;
b = a--;
```

First it assigns the value of a (a = 5) to b, then the value of a is decremented by 1. Finally a and b contains the values 4 and 5 respectively.

Prefix Expression

In this ++ or -- operator are placed before the operand. For example, ++a or --a; when prefix expression is present in an assignment statement, then again two operations are carried out.

Example 1

```
int a = 5, b;
b = ++a;
```

Operation 1: First increment the value of a by 1 (a becomes 6).

Operation 2: After which, the updated value of a is assigned to b.

Finally, the values of a and b are 6 and 6 respectively.

Example 2

```
int a = 5, b;
b = --a;
```

Operation 1: First decrement the value of a by 1 (a becomes 4).

Operation 2: After which, the updated value of a (a = 4) is assigned to b.

Finally, the values of a and b are 4 and 4 respectively.

Program 2 To demonstrate the post-increment and pre-increment

```
main()
{
    int a, b, x = 10, y = 20;
    a = x*y++;
    b = x*++y;
    printf("a = %d, b = %d \n", a, b);
}
```

Output

```
a = 200, b = 220
```

Explanation

The statement a = x*y++ gives the result 200 because first x*y is performed which is 200. After x*y, y gets increased to 21. So, 'a' contains 200. The next statement b = x*++y, first it increases the values of y to 22 and then it perform x*y, so, the result is 10*22 = 220 is assigned to the variable b.

6.4 Relational Operators

In the term relational operators, **relational** refers to the relationships that expressions can have with one another. These operators are used to distinguish between two expressions depending on their relations. If the relation is true then it returns the value 1 otherwise 0 for false relation. In simple words, a relational operator compares two relational expressions or quantities and the result of which is either True (1) or False (0). The relational operators along with their description and examples are shown below.

Relational Operator	Description	Example	Result
>	Greater than	8 > 5	True (1)
<	Less than	8 < 5	False (0)
<=	Less than or equal to	8 <= 8	True (1)
>=	Greater than or equal to	8 >= 10	False (0)
==	Equal to	5==5	True (1)
!=	Not equal to	3! = 3	False (0)

Syntax:

The syntax of relational expression consists of two expressions separated by a relational operator.

exp1 <relational operator> exp2

Where, **exp1** and **exp2** are two expressions and it can also be a variable or a constant.

Example 1

```
(a+b) < (c*d)
a <= (c - d)
a == 10
```

In the above example, == , <= and < are relational operators, which tests the expression and result will be either true(1) or false (0).

The expression a==10 tests whether a is equal to 10?. If it is true it returns the value 1, otherwise it returns the value 0. Let us consider the below expression.

```
i = (a == 10)
```

if a=10 then value of i is 1 (1 indicates true)

if a=20 then value of i is 0 (0 indicates false)

Example 2

If a = 10, b = 5, c = 15, then

1. a == b → returns '0'
2. a >= b → returns 1
3. (a+b) > c → (10+5) > 15 → returns 0
4. (a+b) >= c → (10+5) > = 15 → returns 1
5. c < = (b*2) → 15 < = 10 → returns 0

The following program demonstrates the above example.

Program 3 To demonstrate the relational operators

```
#include <stdio.h>
void main()
{
        int a = 10, b = 5, c = 15;
        printf("a==b        : %d \n", a==b);
        printf("a>=b        : %d \n", a>=b);
        printf("(a+b)>c     : %d \n", (a+b)>c);
        printf("(a+b)>=c    : %d \n", (a+b) >=c);
        printf("c<=(b*2)    : %d \n", c<=(b*2));
}
```

Output

```
a==b        : 0
a>=b        : 1
(a+b)>c     : 0
(a+b)>=c    : 1
c<(b*2)     : 0
```

6.5 Logical Operators

Logical operators are used to combine two or more relational expressions. A logical relationship between the two expressions are checked with logical operators. Using these operators, two expressions can be joined. After checking the conditions it gives logical **true** (1) or **false** (0) status. The operands could be expressions, constants, and variables. The following table describes the three logical operators together with examples and their return values.

Logical Operator	Description	Example	Return value
&&	Logical AND	8>5 && 5<10	True (1)
||	Logical OR	8>5 || 5<2	True (1)
!	Logical NOT	! (8<15)	False (0)

- The expression such as
  (a>b) && (x>b) is termed as a logical expression.
- Like the simple relational expression, a logical expression also yields a value of **one** (or) **zero.**

Expression	What it Evaluates to
(exp1 && exp2)	True (1) only if both exp1 and exp2 are true; False (0) otherwise;
(exp1 || exp2)	True (1) if either exp1 or exp2 is true; False (0) if both are false
! (exp1)	True (1) of exp1 is false False (0) if exp1 is true.

- The truth table for Logical AND, Logical OR and Logical NOT are shown below.

Logical AND

Exp1	Exp2	Exp1 && Exp2
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Logical OR

Exp1	Exp2	Exp1 || Exp2
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Logical NOT

Exp1	! (Exp1)
TRUE	FALSE
FALSE	TRUE

The following program demonstrates the usage of logical operators.

Program 4 **To demonstrate the logical operators**

```
#include <stdio.h>
void main()
{
      printf(" 5>3 && 3<10      : %d \n", (5>3) && (3<10));
      printf(" 7<4 || 3<8       : %d \n", 7<4 || 3<8);
      printf(" !(8==8)          : %d \n", !(8==8));
}
```

Output

```
5>3 && 3<10   :  1
7<4 || 3<8    :  1
! (8==8)      :  0
```

Explanation

In the above program first and second expressions are true, hence the program returns 1. The third expression is false. Hence result returned will be 0.

6.6 Assignment Operator

Assignment operators are used to assign the result of an expression to a variable. The general syntax of the assignment statement is as follows. The assignment operator is represented by the symbol '='.

Syntax

```
variable = value (or) expression;
```

Example 1

```
a = 20;
```

Here, 20 is the value and it is assigned to the variable **'a'** using an assignment operator **'='.**

Example 2

```
b = (2 + 5) * 3;
```

Here, (2 + 5) * 3 is an arithmetic expression and whose result is 7 * 3 = 21 and the result 21 is assigned to the variable **'b'** using an assignment operator **'='.**

Example 3

```
x = (a - b)*(c % d);
```

Here, the expression (a – b)*(c % d) is evaluated based on the values of a, b, c and d, and the result of that expression is assigned to the variable 'x'.

6.6.1 Compound Assignment Operators

Sometimes we may require to modify the value of a variable by incrementing, decrementing, multiplying or dividing it with the value of an expression or a constant using assignment statements such as x = x+a, x = x – a, x = x*a, and x = x/a.

But these assignment statements can be simplified using **compound assignment operators.** These operators are combinations of an assignment operator and other operators such as +, -, *, /, %, |, &, ^, >>, <<, etc.

Operator	Description	Example	Meaning
=	Assignment	Number = 6;	Assigns 6 to Number.
+=	Sum and assign	Number += 10;	adds 10 to Number
-=	Subtract and assign	Number -= 5;	subtract 5 from Number
*=	Multiply and assign	density_1 *= Number;	same as density_1= density_1*Number;
/=	Divide and assign	density_1/=4;	divides density_1 by 4

Note

When a compound operation is used, there must not be any space between the operators.

6.7 Conditional Operator (Ternary Operator)

The **conditional operator** contains a condition followed by two statements (or) values. If the condition is true, then the first statement is executed; otherwise the second statement is executed. The conditional operator is represented by the symbol **(? :).** The conditional operator is also called as **ternary operator** because it requires three arguments /operands. The syntax of conditional operator is given below.

Syntax

```
(expr1)? expr2 : expr3;
```

Where
- **expr1** is an expression evaluated to true or false.
- if **expr1** is evaluated to true, **expr2** is executed.
- if **expr1** is evaluated to false, **expr3** is executed

Note : One of two statements either **expr1** or **expr2** is executed.

Example 1

```
int x = 10, y = 5, z;
z = (x > y) ? x : y;
```

Since (x>y) is true, the expr2 is executed. i.e., the value of x is assigned to the variable z. Therefore 'z' contains the value 10.

Example 2

```
printf ("Result = %d", 2 == 3 ? 5 : 10);
```

It prints the output as

```
Result = 10
```

Because the condition 2 == 3 is false, so the second value 10 gets printed.

Example 3

```
void main()
{
    8 > 4 ? printf ("True") : printf ("False");
}
```

In the above example, we have used full statement using the conditional operator. The condition 8>4 is true, hence, first **printf()** statement is executed. It prints True on the screen

6.8 Bitwise Operators

One of the unique features of C-language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. C has distinction of supporting several special operators known as **bit-wise operators** for manipulation of data at bit-level. Bit-level manipulation are used for testing the bits, or shifting them right, or left. They are also used to perform certain numerical computations faster. The bit-level manipulations may not be applied to **float** and **double.** That is, bit-wise operators work only on integer type operands. The following table gives some of bitwise operators and their work.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
|	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. Every 0 is converted to 1 and every 1 is converted to 0.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.

The bitwise operations provide an efficient way of interacting with hardware and for carrying out arithmetic operations.

Program 5 To demonstrate the usage of Bitwise operators.

```
#include<stdio.h>
void main()
{
        printf("12&8    = %d \n", 12&8);
        printf("8|4     = %d \n", 8|4);
        printf("8^4     = %d \n", 8^4);
        printf("~16     = %d \n", ~16);
        printf("10>>2   = %d \n", 10>>2);
        printf("10<<3   = %d \n", 10<<3);
}
```

Output

```
12&8    = 8
8|4     = 12
8^4     = 12
~16     = -17
10>>2   = 2
10<<3   = 80
```

6.9 Special Operators

C supports some special operators such as

(1) Comma Operator
(2) Size of Operator
(3) Pointer Operators (& and *)
(4) Member Selection Operators (• and →)

We will discuss comma and size of operators in this section and remaining operators will be discussed in next chapters.

6.9.1 Comma Operator

The **comma** operator can be used to link the related expressions together. Two or more expressions may be combined into a single expression using **comma** operator. The **comma** operator makes the statement compact. The general form is,

```
(expr1, expr2, expr3, … exprn);
```

Example 1

Suppose if we have the following three statements in our program

```
temp=a;
a=b;
b=temp;
```

Then, we can combine these three statements into a single statement using comma operator as shown below.

```
temp=a, a=b, b=temp;
```

Example 2

The comma operator has the lowest precedence when compared to all operators in C and has left to right associativity. Therefore the expressions will be evaluated from left to right.

```
int a, b=4, c, d;
d=(a=b+3, c=a+2);
```

Here, first a = b+3 is evaluated and whose result is a = 7. After which c = a+2 is evaluated and whose result is c = 7+2 = 9. The value of second expression is assigned to the variable d. So, the variable **d** contains the value 9.

Example 3

```
int a=4, b=5, c;
c=(++a, b+a);
```

On execution of first expression ++a, a gets value 5. Then the next expression b+a is evaluated to 5+5 = 10. Finally the value 10 of the second expression is assigned to the variable c.

6.9.2 Size of Operator

size of operator returns the number of bytes required / used for the operand specified. Using this operator, we can get the size of any data type, or variable, or constant. Syntax of **sizeof** operator is

Syntax

```
sizeof(operand)
```

where

- Operand can be a data type, or variable, or a constant.

Example

sizeof(char)	returns 1 byte	sizeof(int)	returns 4 bytes
sizeof(float)	returns 4 bytes	sizeof(double)	returns 8 bytes
int i; sizeof(i)	returns 4 bytes		

6.10 Operator Precedence and Associativity

Operator Precedence in C

Operator precedence determines which operator is evaluated first when an expression has more than one operators. The precedence of C operators dictates the order of calculation within an expression.

Associativity in C

Associativity is used when there are two or more operators of same precedence is present in an expression. Associativity can be either Left to Right or Right to Left.

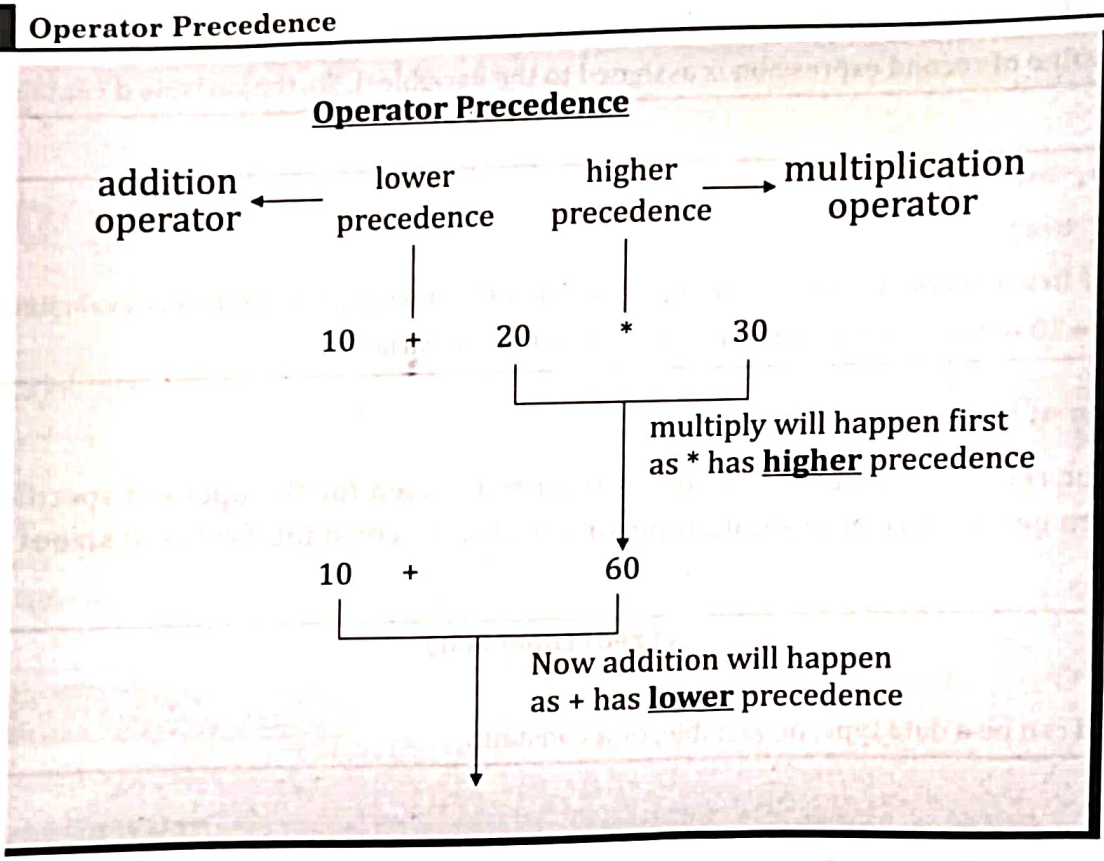
While evaluating the expression 5+3*2/6 consisting of *and/operators with equal precedence, the question that arises is whether to evaluate 3*2 or 2/6 as the first operation. This problem is solved by using **associativity of operators.**

Example 1 Operator Precedence

100 - 2 * 30 would yield 40,

because it is evaluated as 100 – (2 * 30) and not (100 - 2) * 30. The reason is that multiplication * has higher precedence than subtraction(-).

Example 2 Operator Precedence



In the above example. 10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30

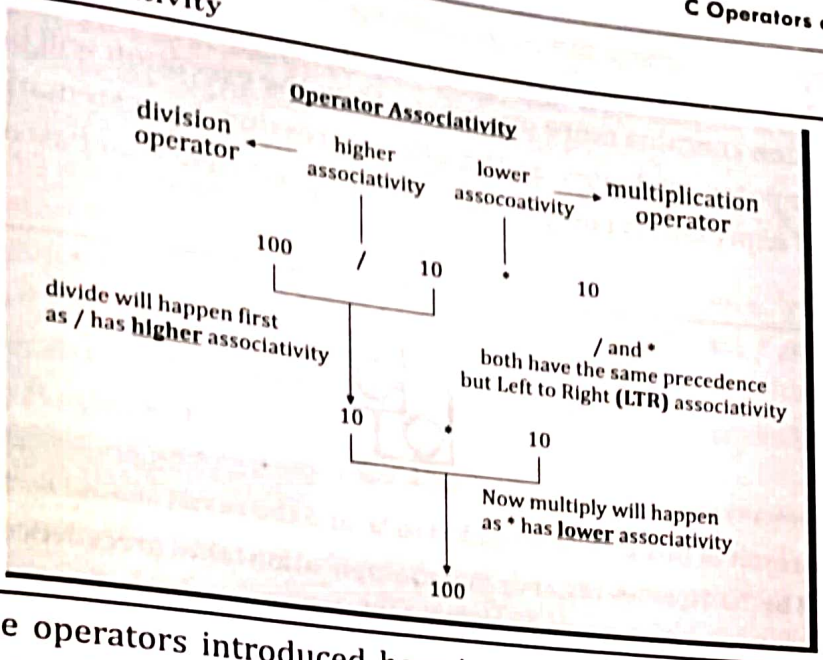
Example 3 Operator Associativity

Multiplication (*) and division(/) arithmetic operators have same precedence,

Lets say we have an expression 5*2/10, this expression would be evaluated as (5*2)/10 because the associativity is left to right for these operators. The multiplication of 5*2 would happen first then division by 10. The result of the expression would be 1.

Similarly 20/2*5 would be calculated as (20/2)*5. The result of the expression would be 50.

Example 4 Operator Associativity



The precedence of the operators introduced here is summarized in the table below. The highest precedence operators are given first.

Operators								Associativity
()	->	.						left to right
!	~	+	-	++	--	&	*	right to left
*	/	%						left to right
+	-							left to right
<	<=	>	>=					left to right
==	!=							left to right
&								left to right
|								left to right
&&								left to right
||								right to left
=	*=	/=	%=	+=	-=			right to left

Here the same operator appears twice (for example *) the first one is the unary version.

Note on Precedence and Associativity

- Associativity is only used when there are two or more operators of same precedence
- All operators with the same precedence have same associativity
- Precedence and associativity of postfix ++ and prefix ++ are different. Precedence of postfix ++ is more than prefix ++, their associativity is also different. Associativity of postfix ++ is left to right and associativity of prefix ++ is right to left.
- Comma has the least precedence among all operators.
- All binary operators have left to right associativity. Whereas unary plus (+) ++, - -, and unary (-) have right to left associativity.
- When two operators of the same priority are found in the expression, precedence is given to the extreme left operator.

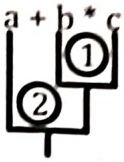
6.11 Precedence of Arithmetic Operators

If an arithmetic expression contains more operators, then the execution will be performed according to their priorities. Computer evaluates arithmetic expression in systematic manner. During the process of evaluation of expressions computers consider precedence and associativity of operators.

Example

Suppose a = 20, b = 10, c = 5

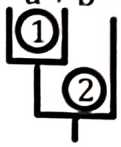
d = a + b * c



First it evaluates **b*c,** the result of this should be added to **'a'** and the result should be stored in the variable **'d'**. Finally the value of **'d'** will be 70. Here we observe that multiplication takes precedence over addition. Similarly every operator in C has its place in the order of precedence.

Consider one more statement.

d = a + b – c



Here, we have 2 cases: We can add a+b first then subtract c from the result or subtract c from b and add the result to a. In both the cases the result is same. Here we observe that + and – have same precedence and so, whichever the symbol appears first from left to right, that operation will be performed first.

The sequence in which arithmetic operators in an expression are processed is known as **Precedence of Arithmetic Operators.**

The **associativity** determines the order in which the operators having equal precedence are to be processed. Associativity is applied only when an expression has operator with equal precedence. Associativity of operators can be either from **left to right** or from **right to left.**

Procedence Level	Operators	Associativity
1	( )	Left to right
2	Unary +, Unary – , ++, --	Right to left
3	*, /, %	Left to right
4	+, –	Left to right

- Expressions enclosed in the parenthesis will get first priority.
- Unary plus (+) and unary minus (–) having equal precedence get second priority.
- Multiplication (*), division (/) and modulus (%) operators having equal precedence get third **priority.**

- Addition (+) and subtraction (–) operators having equal precedence and they get last priority.
- If an expression has nesting of parenthesis, the expression will be evaluated from the innermost parenthesis to the outermost parenthesis.
- All binary operators have left to right associativity. Whereas unary plus (+) ++, – –, and unary (–) have right to left associativity.
- In the expression 3+4*5/2. both * and / have equal precedence. Then compiler uses left to right associativity of these operators and consequently * and / are processed from left to right. So, first it performs 4*5 = 20. Then it performs 20/2 = 10. Finally it evaluates 3+10 = 13.
- When two operators of the same priority are found in the expression, precedence is given to the extreme left operator.

6.12 Arithmetic Expressions

A group of constants, variables and operators arranged as per the syntax of the language is referred to as **Arithmetic Expression.** Depending on data types of operands (variables and constants) present, arithmetic expressions are classified into 3 categories, which are

1. Integer Arithmetic Expressions
2. Real Arithmetic Expressions
3. Mixed Mode Arithmetic Expressions

1. **Integer Arithmetic Expressions:** In an expression, if all the values are integers, then the expression is said to be **Integer Arithmetic Expressions**. The integer expression always yields in an integer value. If evaluated value of expression of type **int** is **float**, such as 5/2 = 2.5, the fraction part will be removed and finally the integer value 2 will be the evaluated value of expression.

Integer Expression	Evaluated Value	Remarks
5 / 2	2	2.5 is truncated to 2
5 * 2	10	
(5 + 8) / 2	6	6.5 is truncated to 6
8 - 3 * 2	2	
–100 + 8	–92	

We can observe from the above table that evaluated value of all the integer expressions are integers.

2. **Real Arithmetic Expressions:** In an expression, if all the values are in float form, then the expression is said to be **Real Arithmetic Expression.** The result of the real arithmetic expression is always **float** value. The following table gives typical floating point expressions.

Real Expressions	Evaluated value
5.0/2.0	2.5
2.0+10.5	12.5
10.5+2.0*1.0	12.5
3.0 – 10.5	–6.5
2.0 – (–9.5)	11.5

3. Mixed Mode Arithmetic Expressions

In mixed mode arithmetic expressions, the expression is composed of **float** and **integer** values. The result of mixed mode arithmetic expression is always a **float** value. In this mode, if an expression has operands of type **float** and **int**, operand of lower type **int** is converted to higher type **float.**

Mixed Mode Expressions	Evaluated Value	Remarks
5.0/2	2.5	2 is converted to 2.0
2.0/5	0.4	5 is converted to 5.0
3.5+4	7.5	4 is converted to 4.0
2.5*4/2	5.0	value of 4/2 is 2, further 2 is converted to 2.0
2*(1.5*2.5)	8.0	2 is converted to 2.0

Any mathematical expression can be converted into C equivalent expression. Mathematical functions such **sqrt(), log(), sin()** etc., can also be the part of the expressions. C can handle any complex mathematical expressions.

NOTE

The expression (a+b) (c+d) is converted into C expression(a+b)*(c+d)

Example **The following table shows different arithmetic expressions and their equivalent C expressions.**

Arithmetic Expression	C Expression
ab – c	a*b – c
$\left[\frac{ab}{c}\right]$	a*b/c
$3x^2+2x+5$	3*x*c+2*x+5
$\left(\frac{a}{b}\right)+c$	a/b+c
$root = \frac{-b+\sqrt{b^2-4ac}}{2a}$	root = (–b+sqrt (b*b – 4*a*c)) / (2*a)
$\frac{ax+b}{ax-b}$	(a*x+b) / (a*x –b)
$s = \frac{a+b+c}{2}$	s = (a+b+c) / 2
$\sqrt{a+b+c}$	sqrt (a+b+c);
$e^{\alpha}+ab$	exp (c*x)+a*b

6.13 Evaluation of Arithmetic Expressions

All the expressions will be evaluated based on their precedence and associativity which is discussed in the previous section. The following rules are required to follow for evaluation of expressions.

- When parenthesis are used, the expression within parenthesis will be executed first.
- If two (or) more parenthesized expressions are used, the left most parenthesized expression is evaluated.

  For example, In 3+(9*2)+(8–2)*5 expression, First (9+2) is evaluated and then (8–2) is evaluated.
- If parenthesis are nested, the evaluation begins with the inner most expression.

Example 1

(2 + ((5 * 2) + 3) * 4)

Here, the innermost expression (5*2) is evaluated first, then ((5*2)+3) is evaluated and finally (2+((5*2)+3) *4) is evaluated.

∴ (2+((5*2)+3)*4) ⇒ (2 + (10 + 3) * 4)
⇒ (2 + 13 * 4)
⇒ (2+52)
⇒ 54.

- The precedence rule is applied in determining the order of applications of operators in evaluating expression.

Example 2

2+13 * 4

Here, * has higher precedence than+and hence we evaluate 13*4 first and then we add 2 with the result of 13 * 4 = 52. Hence we get the value 54.

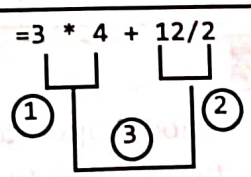
- The associativity rule is applied when two or more operators of the same precedence appear in an expression.

Example 3

5 + 6 – 2.

Here, both + and – are having same precedence. So, we apply associativity rule to evaluate this expression. These operators have left to right associativity and hence first operator which comes from left is evaluated first. In this example + comes first and it adds 5+6 = 11 and then it evaluates 11 – 2 = 9.

Example 4



Here, 3 * 4 is solved first. Though * and / have the same precedence, the operator * comes before '/'.

x = 3*4 + 12/2 — 1st Operation (3 *4)
= 12 + 12/2 — 2nd Operation (12/2)
= 12 + 6 — 3rd Operation (12 + 6)
= 18

Example 5

x = 8 - 24 / (3+3) * (3-1)

Step 1: Here, two parenthesized sub expressions are available. They are (3+3) and (3-1). Fir evaluate the left most parenthesized subexpression (3+3)

x = 8 - 24 / 6 * (3 - 1)

Step 2: Now evaluate (3 - 1)

x = 8 - 24 / 6 * 2

Step 3: Here,/and * are having the same precedence. Evaluate 24/6 first because/occurs before *.

x = 8 - 4 * 2

Step 4: Now * has higher precedence than -. Therefore evaluated 4 * 2 first

x = 8 - 8.

Step 5: Now evaluate the simple expression x = 0.

Example 6

x = 2* ((8% 5) * (4 + (15 - 3) / (4 + 2)))

= 2 * ((8% 5) * 4 + 12/(4 + 2)))

= 2 * ((8% 5) * (4 + 12/6)

= 2 * (3 * (4 + 12/6)

= 2 ((3 * 4 + 2))

= 2 * (3 * 6)

= 2 * 18

x = 36

Example 7

x = 10* 8/(3 + 4 / 2 - 1) * 2) / (2 + 3)

= (10 * 8/3 + 2 - 1) *2) / (2 + 3)

= (10 * 8 / (5 - 1) * 2) / (2 + 3)

= (10 * 8) / 4 * 2) / (2 + 3)

= (80 /4 * 2) / 2 + 3)

= (20 * 2) / (2 + 3)

= 40 / (2 + 3)

= 40 / 5

x = 8

6.14 Relational Expressions

A Relational Expression consists of relational operators, constants, variables and other operat
A relational expression consists of relational operators which compares two expressions and outcome of which is either TRUE (1) or FALSE (0). The general form of relational expression is

```
exp1 <relational operator> exp2
```

Here, exp1 and exp2 can also be the variables or constants. The evaluated value of a relati expression is either '0' or '1'.

Example

Relational Expression	Evaluation
(20+15) > 20/5	= 35 > 20/5 = 35 > 4 = TRUE (1)
30/10 ! = (5 * 2 – 1)	= 30/10 ! = (10 – 1) = 30/10 ! = 9 = 3 ! = 9 = TRUE (1)
3 * 7 – 2 < = 5 * 3+4/2 – 1	= 21 – 2 < = 5 * 3+4/2 – 1 = 21 – 2 < = 15+4/2 – 1 = 21 – 2 < = 15+2 – 1 = 19 < = 15+2 – 1 = 19 <= 17 – 1 = 19 <= 16 = FALSE (0)

6.15 Logical Expressions

Logical operator AND, OR and NOT are used to connect two or more relational and arithmetic expressions, thus resulting in a **logical expression.** The symbols used for logical operators AND, OR and NOT are &&, || and ! respectively.

Example

Logical Expression	Evaluation
5 > 2 && 6 < = 8	5 >2 is true and 6 < = 8 is also true. Therefore 1&&1 = True (1).
(10+20) > 50 || 2 < 3	= 30 > 50 || 2 < 3 = False || True = True (1)
! (8 = = 8) || 3 < 2	= ! (True) || 3 < 2 = False || False = False (0)
0 && 1 && 1	= False && True && True = False (0)
(8 <= 10) && (3 == 3)||(2>10)	= True && (3 = = 3) || (2>10) = True && True || (2 > 10) = True && True || False = True || False = True (1)

6.16 Type Conversion

Type conversion helps to convert a value of one type to another. The type conversions are classified into two types. They are

1. Implicit Type Conversion
2. Explicit Type Conversion

6.16.1 Implicit Type Conversion

We have already studied the evaluation of mixed mode arithmetic expressions. In mixed mode arithmetic expression it contains operands of different data types. Certain type conversions are automatically carried out by the computer itself and these automatic type conversions are called **implicit type conversions.** For example, while evaluating 5.0/2, the constant 2 of type int and 5.0 of type float and the integer decimal point at its end as 2.0, the result is in float form.

- When the variable or constant of lower type (having lower bytes) is converted to higher type (having higher bytes), it is called **promotion.**

Example 1

```
int i = 5;
float f ;
f = i;
```

The above example assigns an integer variable i to float variable f. Here i is lower type and f is higher type. An integer variable i is automatically converted to float since f is a float variable.

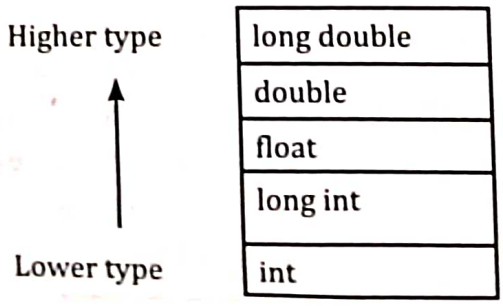
- When the variable or constant of higher type is converted to lower type, then it is called demotion.

Example 2

```
float f = 3.142;
int i;
i = f;
```

Here f is higher type and i is lower type. Before assigning the variable f to an integer variable i, the float value 3.142 is converted to integer by truncating the fraction part, so the value of i is 3.

- Hierarchy of data types is shown below:



6.16.2 Explicit Type Conversion

In this method, we can explicitly convert the value of one type to another. The casting operators are used to accomplish this task.

Syntax

(type) expression

Where

- **type** is any data type available in C

Example 1

```
x = (float) (5+6) ;
```

Here, the expression (5+6) is explicitly converted to float by using the casting operator (float).

Example 2

```
(Float) (6/4)
```

value of 6/4 is 1. After which 1 is converted to float constant as 1.0

Example 3

`(float) 6/4.` Here, 6 is converted to 6. and the value of 6.0/4 will be 1.5.

The following program demonstrates the type conversion.

Program 6 To demonstrate the type conversions

```
#include <stdio.h>
main( )
{
    printf ("5/2      = %d\n", 5/2);
    printf ("5.0/2    = %f\n", 5.0/2);
    printf ("5/2      = %f \n", (float) 5/2);
}
```

Output

```
5/2    = 2
5.0/2  = 2.5
5/2    = 2.5
```

6.17 Library Functions

C Standard library functions or simply C Library functions are inbuilt functions in C programming.

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.
- Each library function in C performs specific operation.
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.
- These library functions are created by the persons who designed and created C compilers.
- All C standard library functions are declared in many header files which are saved as filename.h. Function prototype and data definitions of these functions are written in their respective header file

- We should include the header files in our program using "#include<filename.h>" to make use of the functions those are declared in the header files.
- When we include header files in our C program using "#include<filename.h>" command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.
- **Example**: If we want to use printf() function, the header file <stdio.h> should be included.

```
# include <stdio.h>
void main()
{
      /* If we write printf() statement without including header file, this program
      will not compile. */
      printf("Skyward Publishers");
}
```

- There is at least one function in any C program, i.e., the main() function (which is also a library function). This program is called at program starts. There are many library functions available in C programming to help the programmer to write a good efficient program.
- Suppose, if we want to find the square root of a number. We can write our own code to find square root but, this process is time consuming. We can find the square root by just using sqrt() function which is defined under header file "math.h".

```
#include <stdio.h>
#include <math.h>
void main()
{
      float num,sqroot;
      printf("Enter a number to find square root :");
      scanf("%f",&num);
      sqroot=sqrt(num);     /* Computes the square root of num and stores  in root. */
      printf("Square root of %.2f=%.2f",num,sqroot);
}
```

6.18 Mathematical Functions

The C language has number of library functions that carry out various operations. Let us see now only some mathematical functions. The definitions of all mathematical functions are included in the header file **<math.h>**. Some of the mathematical functions are given in the following table.

Mathematical Function	Operation Performed
int abs(int i)	Returns an absolute value of an integer i
double acos(x)	Returns the arc cosine of x in the range 0 to π. The input parameter x should be in the range –1.0 to 1.0

double asin(x)	Returns the arc sine of x in the range $-\pi/2$ to $\pi/2$. The input parameter x should be in the range -1.0 to 1.0.
double atan(x)	Returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.
double atan2(y,x)	Returns the arc tangent of y/x in the range $-\pi$ to π
double ceil(x)	Returns the smallest whole number not less than x. For example, ceil (3.1) = 4.0, ceil (-3.1) = 3.0
double cos(x)	Returns the cosine of an angle. x should be in radians.
double exp(x)	Returns the value e^x
double fabs(x)	Returns the absolute value of floating point number x
double floor(x)	Returns the largest whole number not greater than x. For example, floor (2.9) = 2, floor (-2.9) = -3.0
double fmod(x,y)	Returns the floating point remainder of x/y.
double log(x)	Returns the natural logarithm of x
double log10(x)	Returns the logarithm of base 10
dobule pow(x,y)	Returns the value of x^y
double sin(x)	Returns the sine of an angle. x is in radians
double sqrt(x)	Returns the squre root of x
double tan(x)	Returns the tangent of an angle. x is in radians.

Program 7 **To demonstrate Mathematical Functions**

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

void main()
{
      printf("\n ceil(3.6)  = %4.2f",ceil(3.6));
      printf("\n ceil(3.3)  = %4.2f",ceil(3.3));
      printf("\n floor(3.6)  = %4.2f",floor(3.6));
      printf("\n floor(3.2)  = %4.2f",floor(3.2));
      printf("\n sqrt(16)  = %4.2f",sqrt(16));
      printf("\n sqrt(7)  = %4.2f",sqrt(7));
      printf("\n pow(2,4)  = %4.2f",pow(2,4));
      printf("\n pow(3,3)  = %4.2f",pow(3,3));
      printf("\n abs(-12)  = %d",abs(-12));
      printf("\n abs(85)  = %d",abs(85));
```

Output

```
ceil(3.6)  = 4.00
ceil(3.3)  = 4.00
floor(3.6)  = 3.00
floor(3.2)  = 3.00
sqrt(16)  = 4.00
sqrt(7)  = 2.65
pow(2,4)  = 16.00
pow(3,3)  = 27.00
abs(-12)  = 12
abs(85)  = 85
```

6.19 Sample Programs

Program 8 Write a program to find the square of the given number.

```
#include <stdio.h>
main()
{
        int num, result;
        printf("Enter any integer number :");
        scanf("%d", &num);
        result = num*num;
        printf("\n Square of given number = %d", result);
}
```

Output

```
Enter any integer number : 4
Square of given number = 16
```

Program 9 Write a program to find area and circumference of the circle. [Accept radius value through keyboard].

We know that, formula to find area of the circle is

area = πr^2
= π * radius * radius
= 3.142 * radius * radius

Formula to find circumference of the circle is

circumference = 2 π r = 2 * 3.142 * radius

```
#include <stdio.h>
void main()
{
        float radius, area, circum;
        printf ("\n Enter the radius of the circle:");
        scanf ("%f", &radius);
        area = 3.142 * radius * radius;
        circum = 2 * 3.142 * radius;
        printf ("\n Area of the circle = %f", area);
        printf ("\n circumference is = %f", circum);
}
```

Output

```
Enter the radius of the circle : 12
Area of the circle = 452.448000
Circumference is = 75.408000
```

Program 10 Write a program to find the simple interest and compound interest.

The formula to find simple interest and compound interest is

```
SI=  p*t*r/100;
CI =  p*pow ((1+r/100), t);
```

Where P is principle, T is time and R is rate of interest. In the following program, we have declared **p** and **t** as variables of **int** and **r** as variable of **float** type. The variables **SI** and **CI** are used to store the values of simple interest and compound interest respectively. **pow( )** is the built in function to find the power. To use this function, we should include <math.h>

```
#include <stdio.h>
#include <math.h>
void main()
{
    int p, t;
    float r, SI, CI;
    printf ("\n Enter principle, time and rate of interest:");
    scanf ("%d %d %f", &p, &t, &r);
    SI = p*t*r/100;
    CI = p*pow((1+r/100), t);
    printf ("\n Simple interest = %f", SI);
    printf ("\n Compound interest = %f", CI);
}
```

Output

```
Enter principle, time and rate of interest 500 5 14
Simple interest = 350.000000
Compound interest = 962.767291
```

Program 11 Write a program to convert Fahrenheit to Celsius degree.

The formula is C = (F – 32) / 1.8

```
#include <stdio.h>
void main()
{
    int f;
    float c;
    printf ("\n Enter the Fahrenheit temperature :");
    scanf ("%d", &f);
    c = (f-32)/1.8;
    printf ("\n The converted Celsius degree = %f", c);
}
```

Output

```
Enter the Fahrenheit temperature : 98
The converted Celsius degree = 36.666666
```

6.20 Review Questions

1. What is an operator?
2. What is an operand?
3. What are logical operators?
4. What are relational operators?
5. What are Arithmetic operators?
6. What are Bitwise operators?
7. What do you mean by unary operator? Mention unary operators.
8. What is the purpose of relational and logical operators?
9. What is the result of the expression (5>2 && 5<4) and (20 + 05) > 30 && 60.
10. Which operators are used both as binary operator and unary operator?
11. What is the difference between = and == ?
12. What is the difference between prefix and postfix increment operator?
13. Write the general syntax of conditional operator or ternary operator.
14. What is the use of bit-wise operators?

15. What is the main use of bit-wise AND (&)?
16. What is the main use of bit-wise OR (|)?
17. What is the value of (5 + 8) /2?
18. What is the value of 2.5*4/2?
19. Write the C expression for $(a^2 + b^2 + c^2)/8$
20. Write the C expression for $\sqrt{a^2 + b^2 + c^2}$.
21. What do you mean by type conversion?
22. What is the value of (float) (6/4)?
23. Mention different types of type conversion.
24. What is the result of (35 < 30 ? 35 : 30) ?
25. Give the precedence of Arithmetic operators.
26. Give the precedence of relational and logical operators.
27. Operators are divided into how many categories? What are they?
28. Give a demonstration program for arithmetic operators.
29. Differentiate between postfix and prefix expression.
30. Write a program to demonstrate relational operators.
31. Give the truth tables for logical AND and logical OR.
32. Write the truth table for logical NOT operation.
33. Write a simple program to demonstrate the logical operators.
34. What is sizeof( ) operator with syntax?
35. Explain comma operator with a suitable example.
36. What are Arithmetic expressions?
37. Explain any 4 mathematical functions.
38. Give the 'C' expression for the following.
    $\sqrt{a+b+c}$, $(a + b)^2 / (c + d^4)$
39. Give the value for the expression.
    (a) (2 + ((3 * 6) + 8) * 4)
    (b) 3*4+12/2
    (c) (6 – 2 / (3 + 3) * (4 – 2))
    (d) 8-24 / (3 +3) * (3 – 1)
40. Give the precedence of logical operators.
41. What is implicit type conversion?
42. What is promotion and demotion?
43. Give the hierarchy for the data type, int, long int, float, double, long double.
44. Give the syntax for explicit type conversion.
45.
```
# include <stdio.h>
main( )
{
        printf ("5/2 = %d/n", 5/2);
        printf ("5.0/2 = %f/n", 5.0/2);
        printf ("5/2 = %f/n", (float) 5/2);
}
```
    What is the output for the above program?