Sl No.	Operator	Meaning
1	<	Less Than
2	>	Greater Than
3	==	Equal to
4	!=	Not Equal to
5	<=	Less Than or Equal to
6	>=	Greater Than or Equal to

Following are the examples of the relational operators.

```
int i = 5;
i < 4;      /* false or 0 */
i > 2;      /* true or 1 */
i < 5;      /* false */
i == 6;     /* false */
i != 10;    /* true */
i <= 5;     /* true */
i >= 4;     /* true */
```

1.6.5 Logical Operators

There are three logical operators in C namely - ! (NOT), && (AND) and || (OR) which when used with operands, return *true* or *false*. These operators can be used for combining several decision making expressions together.

Out of these three, the first one ! (NOT) is a unary logical operator and the other two are binary operators.

The ! operator returns negation of the given operand. These operators can be used for integer and *float* variables.

1.6.6 Assignment Operator

The assignment operator = is used to assign the value of a variable or constant or result of an expression which appears on the right-hand-side of an assignment statement to the variable appearing on the left-hand-side. For example,

```
a = 5;      /* assign 5 to a       */
b = a;      /* b gets value of a   */
c = 2 + 6;  /* assign 8 to c       */
```

When the arithmetic operators are combined with = operator, we get composite operators. This is shown in Table 1.6.

These operators help the programmers to write efficient code, as otherwise they would use the conventional assignment statements (inefficient programs). By using these built in operators they work at hardware level rather than source level.

Truth Table for ! (NOT)

Operand	! (NOT)
false	true
true	false

Truth table for binary Logical Operators

Operand 1	Operand 2	Result (&&)	Result (||)
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Table 1.6 Composite Operators

Sl. No.	Operator	Example	Meaning	When a = 5
1	+=	a += 2	a = a + 2	a = 7
2	-=	a -= 2	a = a - 2	a = 3
3	*=	a *= 2	a = a * 2	a = 10
4	/=	a /= 2	a = a / 2	a = 2
5	%=	a %= 2	a = a % 2	a = 1

1.6.7 Conditional Operator (? :)

This is a very interesting operator as it not an unary or binary operator, but it is called as a **ternary operator**. This means that there will be three operands which generally finds its application in the replacement for *if-then-else* statement (will be discussed later). The Syntax is,

```
expression-1 ? expression-2 : expression-3;
```

An example for this operator is given below:

```
int a = 10;
int b = 5;
int big;
big = a >= ? a : b;
```

If a is greater than b, then assign a to big, else assign b to big. In this example, the output will be, big = 10;

1.6.8 Comma Operator

The related expressions can be combined together using a , (**comma**) operator and its syntax is,

```
expression-1, expression-2, ........... , expression-n;
```

A comma expression is evaluated from left-to-right and the value of the expression is the value of the last expression. For example,

```
length = 2.5, breadth = 4.6, area = length * breadth;
```

We can assign the final value (i.e. `area`) to a left-hand side variable, as shown below:

```
A = (length = 2.5, breadth = 4.6,
area = length * breadth);
```

1.6.9 Bit-wise Operators

The operators we have seen so far operate on variables as a unit (`int`, *float, long,* `double`, etc.). However, bit-wise operators work on bits (binary digits). Following Table 1.7 gives the bit operators allowed in C.

Table 1.7 Bit-wise operators

Sl. No.	Operator	Meaning
1	&	AND
2	|	OR
3	^	EXCLUSIVE-OR
4	~	Complement
5	<<	Left Shift
6	>>	Right Shift

Program 1.11
Logical and shift operators - demo

```
#include <stdio.h>
void main()
{
    int a = 9;
    int b = 5;
    printf("Bit-wise AND = %d\n", a & b); /* 1 */
    printf("Bit-wise OR = %d\n", a | b); /* 13 */
    printf("Bit-wise Complement = %d\n", ~a); /* -10 */
    printf("Bit-wise Left Shift = %d\n", a << b); /* 36 */
    printf("Bit-wise Right Shift = %d\n", a >> b); /* 2 */
}
```

1.6.10 Cast Operator

The cast operator converts the data type of a constant/variable/expression in to a type specified within a bracket. This is called as the **explicit type conversion**. The syntax and example is given below:

```
(type) expression;

float f = 10.2;
int i;
i = (int) f;      /* i gets 10 */
```

The casting does not affect the original variable, it only return the result that temporarily serves the purpose of the expression in which it appears.

Program 1.12
Type casting – demo

```
#include <stdio.h>
void main()
{
     int total;
     float a = 2.7;
     float b = 1.4;
     total = (int)a + (int)b;
     printf("Total : with casting = %d\n", total);
     total = a + b;
     printf("Total : without casting = %d\n", total);
}
```

Sample Run

```
Total : with casting = 3
Total : without casting = 4
```

1.6.11 sizeof Operator

The **sizeof** operator returns the size of a data type (either standard or derived type) in bytes. The syntax and examples are shown below:

```
sizeof(data-ype);
sizeof(int);     /* returns 2 bytes */
sizeof(float);   /* returns 4 bytes */
sizeof(char);    /* returns 1 bytes */
```

This operator is useful in finding the size of a more complicated *structures*. As the size of the data types vary from one machine to the other, one can find the size practically using this operator. Another advantage is that the user need not calculate the size manually instead he can use the *sizeof* operator (more details can be found in later chapters).

1.6.12 Arithmetic Expression

An arithmetic expression comprises of operand and operators for a specific operation. The expression generally consists of *int* or *float* variables or *constants* along with valid arithmetic operators. For example,

```
x = a + b + c;
disc = b * b - 4 * a * c;
celsius = (5.0 / 9.0) * (fahrenheit - 32);
```

The rvalue and lvalue

The purpose of an assignment statement is to evaluate the expression on the right side of = operator and assign that value to the left hand side variable. The expression on the right hand side of = is called as **rvalue** and left hand side is called as **lvalue**.

The result of an *rvalue* expression should always yield a value and the *lvalue* should be a data object (i.e. a variable) capable of holding the value produced by the *rvalue*.

For instance, in the below example the *lvalue* is a variable of type integer and the *rvalue* is a value.

```
temp = 0;
```

The right hand side can be an expression as in the second example shown below:

```
count = 0;
count = count + 1;
```

Below are few illegal *lvalue* expressions (assuming t is an integer)

```
100 = t; /* 100 can't hold value of t */
t + t = 2;
t + 1 = t; /* left hand can't hold t */
```

You can write multiple assignments in a single assignment statement. For example,

```
        a = b = c = 0;

Step 1: c = 0;
Step 2: b = 0;
Step 3: a = 0;
```

The effect of this multiple statement is to assign the *rvalue* (0 in this case) to *lvalue* expression(s).

1.6.13 Operator Precedence

The precedence is the hierarchy of evaluation of an arithmetic or logical expression. In other words, the compiler follows a particular order in the evaluation of a sub-expression. In the below mentioned example, what would be the order of evaluation?

```
y = a + b * c;
```

Assuming a = 10, b = 20, and c = 30,

```
y = 10 + 20 * 30;
```

There are two possibilities in evaluating the above expression. First, evaluate 20 * 30 and then 10 is added to it to obtain 610. Secondly, evaluate 10 + 20 and then multiply 30 with it which yields 900.

Now you can easily see that whenever the order of evaluation is changed, the result differs (this happens when different operators are involved). Therefore, a common procedure is required in evaluating an expression. This is based on what is known as **operator precedence**. That is, which operator must be evaluated first and which one second, and so on. Also when two or more operators of the same precedence are written in an expression we require a particular direction of evaluation – left-to-right or right-to-left. This is called as the **associativity rule**. For example,

```
int a = 5;
int b = 7;
int c = 2;
int y;
y = a * b / c;
printf("%d\n", y); /* y = 17 and not 15 */
```

In the above example, you get the answer for `y` as 17 by following left-to-right assiciativity.

The C compiler assigns highest precedence to the parentheses. This indicates that the sub-expression in the parenthesis is evaluated first and in case if you write nested parentheses, then the innermost expression in the brackets will be evaluated first.

1.6.14 Type Conversion

In an expression, it is obvious that you may mix up variables or constants of various data types and in such cases, how does the compiler evaluate that expression?

This is done by making all the variables and constants into one common type and evaluate. This is called as **type conversion**. Type conversion is a change in the data type of a variable or expression (done either by the user or the compiler) to match up different data types. There are two types of type conversions namely:

- Implicit type conversion
- Explicit type conversion