Chapter 3

Operators & Expressions

Dr. Niroj Kumar Pani
nirojpani@gmail.com
Department of Computer Science Engineering & Applications
Indira Gandhi Institute of Technology
Sarang, Odisha

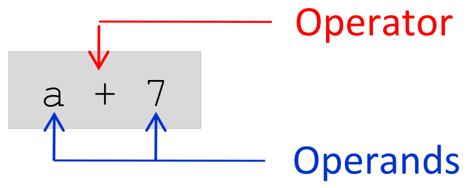
Chapter Outline…

- Operators & Expressions: An Overview
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment & Decrement Operators
- Conditional Operator
- Other Special Operators
- Operator Precedence & Associativity
- Assignments

Operators & Expressions: An Overview

- **Operators:** An operator in general, is a symbol that operates on operands (variables and constants) of certain data type.

e.g.,



- **Expression:**
  - An expression is a combination of one or more operands (variables & constants) and zero or more operators written according to the syntax of the language.
  - Every expression must evaluate to a single value of certain type that can be assigned to a variable.
  - **Examples of expressions:**

```
a + 7
y *((x < 10) && ( x > 5))
a /*Even a single variable or constant can be considered an expression. */
```

- **Types of Operators in C:** In C, operators can be classified into the following category based on their utility and action:
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Increment & Decrement Operators
  - Conditional Operator
  - Other Special Operators

We will discuss these one-by-one in the coming sections.

Arithmetic Operators

- **C has following arithmetic operators:**

Operator	Meaning	
+	Addition or Unary plus	
-	Subtraction or Unary minus	Unary / Binary
*	Multiplication	
/	Division	Binary
%	Modulus division	

- **The Operators +, -, *, /:**
  - **They are applicable to all basic data types, except `void`.**
  - **Working:**
    - **These operators works in the similar manner as that in arithmetic.**

- When these operators are applied to operands both of which have the same data type, then the result of the expression is also in that same data type.
- When these operators are applied to operands both of which have different data types, then 1st the operand in *lower data type* is promoted to *higher data type*, then the expression is evaluated.

➢ **Examples:**

Arithmetic Expression	Result
`8 + 6`	`14`
`8 - 6.0`	`2.0` /* 6.0 is double. So, 1st 8 is promoted double (8.0), then the expression is evaluated*/
`8 * 6`	48
`8 / 6`	1 /*Both operands are integers. So the decimal part is truncated*/
`8 / 6.0`	`1.333333`
`6 / 8`	0
`-6 / -8`	0
`-6 / 8`	`0 or -1` /*The result is machine dependent*/

- **The operator %:**
  - It is only applicable on integers.
  - **Working:**
    - It gives the reminder after division.
    - If there is a -ve sign in one or both operands, then the result takes the sign of the 1st operand.
  - **Examples:**

Arithmetic Expression	Result
`14 % 3`	2
`-14 % 3`	-2
`14 % -3`	2
`-14 % -3`	-2
`14 % 3.0`	Error

- **[NOTES]:**

  - **In C, there is no operator for exponentiation. However, to calculate exponents, C provides an in-built math function `pow()`.**

    `pow (x, y)` *means* x^y

  - **All mathematical functions, including `pow()` exists in `math.h`.**
  - **All mathematical functions, including `pow()` accepts parameters of type `double` only and also returns `double`.**
  - *For other standard math functions Refer: Any textbook (Balagurusamy) or Wikipedia.*

Programming Example:

```
/* PR3_1.c: Program to convert days to months & days */

#include <stdio.h>
#include <conio.h>

void main()
{
   int months, days;

   printf("\n Enter days: ");
   scanf("%d", &days);

   months = days / 30 ;
   days   = days % 30 ;
   printf("\n Months = %d\tDays = %d", months, days) ;

   getch();
}
```

Output

```
Enter days: 365
Months = 12     Days = 5
```

Relational Operators

- **C has following relational operators:**

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater then
>=	Greater than or equal to
==	Equal to
!=	Not equal to

All are Binary

- **The relational operators are applicable to all basic data types, except `void`.**

- **Working: Each relational operator compares its left-hand side with its right-hand side and produces the result as:**
  - **0, if the condition is evaluated to be false.**
  - **1, if the condition is evaluated to be true.**
- **Examples:**

Relational Expression	Result
4.5 <=10	1
10 < -7-5	0
5 == 5	1
7.0 != 7	0
6.35 != +6.35	0

- **[NOTE]: In C**
  - False ⇒ 0 (False is stored as 0) **and** 0 ⇒ False (0 is treated as false)**, while**
  - True ⇒ 1 (True is stored as 1) **and**

    Any non-zero value (including 1) ⇒ True (a non-zero value is treated as true)

Programming Example:

```
/* PR3_2.c: Program to demonstrate the relational operators*/

#include <stdio.h>
#include <conio.h>

void main()
{
    int a, b, c, d;

    a = (4.5 <=10);
    b = (4.5 < -10);
    c = (5 == 5);
    d = (6.35 != +6.35);

    printf(" a = %d \n b = %d \n c = %d \n d = %d", a, b, c, d);

    getch();
}
```

Output

```
 a = 1
 b = 0
 c = 1
 d = 0
```

Logical Operators

- C has following logical operators:

Operator	Meaning	
!	Logical NOT	Unary
&&	Logical AND	Binary
||	Logical OR	

- The logical operators are applicable to expressions each of whose operands are (or evaluates to) either true (non-zero) or false (zero).

- **Working: Like the relational operators, the logical operators also yield the result as either 0 or 1, depending upon the following rules:**

Op	!Op
False (zero)	1
True (non-zero)	0

Op1	Op2	Op1 && Op2
False (zero)	False (zero)	0
False (zero)	True (non-zero)	0
True (non-zero)	False (zero)	0
True (non-zero)	True (non-zero)	1

Op1	Op2	Op1 || Op2
False (zero)	False (zero)	0
False (zero)	True (non-zero)	1
True (non-zero)	False (zero)	1
True (non-zero)	True (non-zero)	1

- **Examples:**

Logical Expression	Result
5 &&6	1
0 || 10.11	1
!5	0
-11.6 && 10.9+1	1
!0	1

- **[NOTE]: Generally, the relational and logical operators are used to represent conditions in branch and loop control structures.**

  e.g., print your name if your roll number is between 3 and 10.

```
if (rollNo >= 3 && rollNO <= 10)
{
   printf("My Name is Don");
}
```

Bitwise Operators

- **C has following bitwise operators:**

Operator	Meaning	
~	Bitwise complement	Unary
&	Bitwise AND	Binary
|	Bitwise OR	
^	Bitwise XOR	
<<	Bitwise left shift	
>>	Bitwise right shift	

- **The bitwise operators are normally applicable to integral data types, i.e., integers and characters.**
- **These operators operate on individual bits of the data.**

- **Bitwise Complement (~):**
  - **Works as per the following rule:**

a	~a
0	1
1	0

  - **Example:**

```
int a = 10;
int b = ~a;  /* The value of b becomes -11 (Shown below) */
```

```
 a    0000  0000  0000  1010
-----------------------------
~a    1111  1111  1111  0101  (-11 in 2's complement form)
```

- **Bitwise AND (&):**
  - **Works as per the following rule:**

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

  - **Example:**

```
int a = 13;
int b = 7;
int c = a&b;  /* The value of c becomes 5 (Shown below) */
```

```
  a   0000 0000 0000 1101
  b   0000 0000 0000 0111
-------------------------
a&b   0000 0000 0000 0101
```

- **Use:**
  - Bit wise AND is often used to test whether a particular bit of a variable is 1 or 0. For example, the following code tests whether the 4th bit of a variable `flag` is 1 or 0.

```
#define TEST 8  /* 8 in binary form is 0000 ... 0000 1000 */

void main()
{
    int flag;
    ...
    ...
    if (flag & TEST) /* test 4th bit */
    {
         printf ("4th bit is set");
    }
}
```

  - Bit wise AND is also used for AND masking (covering by 0s):

```
     a    0000 0011 1011 1101
  mask    1111 1111 1111 0000
---------------------------------
a&mask    0000 0011 1011 0000
```

We want to AND mask (cover by 0s) the four least significant bits of 'a'. So we AND a 'mask' with 'a' whose four least significant bits are 0.

Bitwise OR (|):

- Works as per the following rule:

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

- Example:

```
int a = 13;
int b = 7;
int c = a|b;  /* The value of c becomes 15 (Shown below) */
```

```
  a   0000 0000 0000 1101
  b   0000 0000 0000 0111
-------------------------
a|b   0000 0000 0000 1111
```

- **Use:**
  - **Bit wise OR can also used be to test whether a particular bit of a variable is 1 or 0 (But this use is not so common).**
    For example, to test whether the 4th bit of a variable `flag` is 1 or 0 we OR this variable with 1111 1111 1111 <u>0</u>111.
  - **Bit wise OR is also used for OR masking (covering by 1s):**

```
      a    0000  0011  1011  1101
   mask    0000  0000  0000  1111
-------------------------------------
 a|mask    0000  0011  1011  1111
```

We want to OR mask (cover by 1s) the four least significant bits of ‘a’. So we OR a ‘mask’ with ‘a’ whose four least significant bits are 1.

Bitwise XOR (^):

- Works as per the following rule:

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

- Example:

```
int a = 13;
int b = 7;
int c = a^b;  /* The value of c becomes 10 (Shown below) */
```

```
  a   0000 0000 0000 1101
  b   0000 0000 0000 0111
-------------------------
a^b   0000 0000 0000 1010
```

- **Use:**
  - **Bit wise XOR is often used to determine "how many bits are different in two variables".**

```
  a    0000  0011  1011  1101
  b    0000  1101  0001  1100
 ----------------------------
a^b    0000  1110  1010  0001
```

There are six 1's in the result. So there are six bits in 'a' which are different in 'b'.

Bitwise Left Shift (<<):

- Works as per the following rules:
  - **Case 1 - If the number is unsigned:**

$00000011 \ll 2 = 00001100$ [Here, 00000011 refers to an unsigned number]

$\cancel{0}$	$\cancel{0}$	0	0	0	0	1	1	**0**	**0**

(The Rule: Remove *two* bits form the most significant place and append *two* <u>0s</u> at the least significant place)

  - **Case 2 - If the number is signed (+ve or -ve):**

$00100011 \ll 2 = 00001100$ [Here, 00100011 refers to a positive number]

0	$\cancel{0}$	$\cancel{1}$	0	0	0	1	1	**0**	**0**

(The Rule: Exclude the sign bit. Now remove *two* bits form the most significant place (after the sign bit) and append *two* <u>0s</u> at the least significant place)

$10100011 \ll 2 = 10001100$ [Here, 10100011 refers to a negative number]

1	$\cancel{0}$	$\cancel{1}$	0	0	0	1	1	**0**	**0**

(The Rule: Exclude the sign bit. Now remove *two* bits form the most significant place (after the sign bit) and append *two* <u>0s</u> at the least significant place)

- **Example:**

```
int a = 10;
int b = 3;
int c = a<<b;  /* The value of c becomes 80 */
```

- **Use:**
  - **Left shift is used to multiply a number by a power of 2.**

    `10 << 3` *means* 10×2^3

  - **Since shifting is faster than multiplication, considerable amount of execution time can be saved by using the left shift operator instead of the multiplication operator.**

- **Bitwise Right Shift (>>):**
  - **Works as per the following rules:**
    - **Case 1 - If the number is unsigned:**

00000111 >> 2 = 00000001 [Here, 00000111 refers to an unsigned number]

0	**0**	0	0	0	0	0	1	~~1~~	~~1~~

(The Rule: Remove *two* bits form the least significant place and append *two* 0s at the most significant place)

    - **Case 2 - If the number is signed (+ve or -ve):**

00100011 >> 2 = 00001000 [Here, 00100011 refers to a positive number]

0	**0**	**0**	0	1	0	0	0	~~1~~	~~1~~

(The Rule: Exclude the sign bit. Now remove *two* bits form the most significant place and append *two* 0s at the least significant place (after the sign bit))

10100011 >> 2 = 10001000 [Here, 10100011 refers to a negative number]

1	**0**	**0**	0	1	0	0	0	~~1~~	~~1~~

(The Rule: Exclude the sign bit. Now remove *two* bits form the most significant place and append *two* 0s at the least significant place (after the sign bit))

- **Example:**

```
int a = 10;
int b = 3;
int c = a>>b;  /* The value of c becomes 1 */
```

- **Use:**
  - **Right shift is used to divide a number by a power of 2.**

    10 >> 3 *means* 10 / 2^3

  - **Since shifting is faster than division, considerable amount of execution time can be saved by using the left shift operator instead of the division operator.**

Assignment Operators

- C has two types of assignment operators: simple and compound.
- Simple Assignment Operator:
  - We have already used this operator. It is the = sign.
  - Working: The operator '=' *assigns* the result of the expression on its right to the variable on its left.

  The following codes will clarify the concept of *assignment*:

```
int i = 12, j = 5, k;
float x = -3.1, y ;

k = i + j; /* RHS is evaluated to 17. It is then assigned to k (of type int).
              So, after the assignment  k will contain 17 */

y = i + j; /* RHS is evaluated to 17. It is then assigned to y (of type float).
              So, after the assignment  y will contain 17.000000 */
```

[Cont.]

```
k = x * j; /* RHS is evaluated to -15.5. It is then assigned to k (of type int).
              So, after the assignment  k will contain -15 */

y = i / j; /* RHS is evaluated to 2. It is then assigned to y (of type float).
              So, after the assignment  y will contain 2.000000 */

y = (float) i / j; /* RHS is evaluated to 2.4. It is then assigned to y (of
                      type float).  So, after the assignment  y will contain
                      2.400000 */
```

- **[NOTE]: The operator '=' allows only a single *variable* on its left.**

```
x = 10.9 + 2.1;  /* Legal. Assigns 13.0 to x*/
x + y = 10.9 + 2.1  /* Illegal. More than one variable in LHS of = */
10.9 = x + 2.1  /* Illegal. LHS of = is a constant */
```

Compound Assignment Operators:

- **The compound assignment operators are formed by combining the arithmetic and bitwise *binary* operators with the = operator.**

The Format (for combining) is: `op=`

The compound assignment operators in C are listed below:

Compound assignment operator	Working / Meaning
+=	a +=b *means* a = a+b
-=	a -=b *means* a = a-b
*=	a *=b *means* a = a*b
/=	a /=b *means* a = a/b
%=	a %=b *means* a = a%b
&=	a &=b *means* a = a&b
|=	a |=b *means* a = a|b
^=	a ^=b *means* a = a^b
<<=	a <<=b *means* a = a<<b
>>=	a >>=b *means* a = a>>b

■ Programming Example:

```
/* PR3_3.c: Find the square of a number by using the operator *= */

#include <stdio.h>
#include <conio.h>

void main()
{
    int num;

    printf("\n Enter an integer: ");
    scanf("%d", &num);
    num *= num;
    printf ("\n It\'s square is: %d", num);

    getch();
}
```

Output

```
Enter an integer: 6
It’s square is: 36
```

[NOTE]: Lvalues & Rvalues

- There are two kinds of expressions in C:
  1. lvalue: An expression that is a lvalue may appear as either the left-hand or right-hand side of an assignment.
  2. rvalue: An expression that is a rvalue may appear on the right- but NOT on the left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side or on the right-hand side of an assignment. But constants are rvalues. So, they can’t appear on the left-hand side of an assignment.

For Example:

```
int x = 10;  /* Valid statement*/
20 = 10;     /* Invalid statement
```

Increment & Decrement Operators

- **Increment Operator:** `++`
- **Decrement Operator:** `--`
- **Syntax:**

```
++variable;  or  variable++;  e.g., ++x;  or  x++;
--variable;  or  variable--;  e.g., --x;  or  x--;
```

- **Working / Meaning:**
  - **When these operators form statement independently, they mean the same thing:**

```
Both ++x;  and  x++;  means  x = x+1;
Both --x;  and  x--;  means  x = x-1;
```

- However, they behave differently when they are used in expressions on the right-hand side of an assignment statement:

```
y = ++x;  means x = x+1; y = x;  (i.e., 1st increment, then assignment)
y = x++;  means y = x; x = x+1;  (i.e., 1st assignment, then increment)

y = --x;  means x = x-1; y = x;  (i.e., 1st decrement, then assignment)
y = x--;  means y = x; x = x-1;  (i.e., 1st assignment, then decrement)
```

- **NOTES:**
  - `++` and `--` are not applicable to constants.

```
int x = 7;
x++;  /* Correct */
7++;  /* Illegal */
```

  - `++` and `--` are only applicable to integral values.

```
float x = 7.1;
x++;  /* Illegal */
```

■ Programming Example:

```
/* PR3_4.c: Use of ++ and -- operators*/

#include <stdio.h>
#include <conio.h>

void main()
{
   int x = 50, y = 100;

   printf ("\n %d", 10 + x++);
   printf ("\n %d", 10 + ++x);

   printf ("\n %d", 10 + y--);
   printf ("\n %d", 10 + --y);
   getch();
}
```

Output

```
60
62
110
108
```

Conditional Operator (? :)

- **The conditional operator "? : " is called a ternary operator since it takes three operands.**
- **Syntax:**

```
(expression1) ? (expression2) : (expression3);
```

- **Working:**
  - `expression1` **is evaluated 1st.**
  - **If the value** `expression1` **is true (non-zero), then the value of the *whole expression* becomes equal to the value of** `expression2`.
  - **If the value** `expression1` **is false (zero), then the value of the *whole expression* becomes equal to the value of** `expression3`.
- **The working is similar to:**

```
if (expression1)
{
    expression2;
}
else
{
    expression3;
}
```

■ Examples:

```
int a = 10, int b = 15, c = 1, x;

x = (a>b)?a:b; /* x will be assigned to 15 */

x = (a>=65 && a<=90)?1:0; /* x will be assigned to 0 */

(c==1)?printf(“Hi”):printf(“Bye”); /* “Hi” will be printed*/

x = (a>b ? (a>c?3:4) : (b>c?6:8)); /* x will be assigned to 6 */
```

Other Special Operators

- **C supports some special operators such as:**
  - **Comma operator (,)**
  - **sizeof operator**
  - **Pointer operators (& and *)**
  - **Member selection operators (. and ->)**

In this section, we will discuss the comma and sizeof operators. Pointer operators will be discussed when we study “Pointers” and member selection operators will be discussed when we study “Structures and Unions”

The Comma Operator (,)

- The comma operator can be used to link related expressions together.

  For Example:
  ```
  z = (x = 10, y = 5, x+y);
  for (n = 1, m = 10; n<=m; n++, m++)
  while (c = getchar(), c!= '10')
  ```

- Working: A comma linked expression is evaluated from *left to right* and the value of the *right-most* expression is the value of the combined expression.

  For Example: In the statement `z = (x = 10, y = 5, x+y);`

  - First 10 is assigned to x.
  - Then 5 is assigned to y.
  - Then x+y is calculated which is assigned to z.

  So, the value of z becomes 15.

The `sizeof` Operator

- The `sizeof` operator is used to determine the number of bytes occupied by an operand of certain data type.
- Syntax:

```
x = sizeof(operand);
```

Where,

- `x` is an integer variable
- `operand` may be a variable, constant, or a data type.

- Example:

```
int a;
double d;
a = sizeof (d); /* 'a' will be assigned to 8 */
a = sizeof (3.5); /* 'a' will be assigned to 8 */
a = sizeof (double); /* 'a' will be assigned to 8 */
```

Operator Precedence & Associativity

- Consider the following statement:

  ```
  z = x == 10 + 15 && y < 10;
  ```

  How to evaluate this?

  Which operator is evaluated 1st, which is at 2nd....which is at the last?

- The order of evaluation of operations in an expression is determined by their precedence and associativity.

- Operator Precedence:
  - Precedence determines how an expression involving more than one operator is evaluated.
  - Every operator belongs to a specific precedence level / group.
  - Operators with higher precedence are evaluated first before those with a lower precedence.

- **Operator Associativity:**
  - **Associativity determines how the operators that belong to the same precedence level / group are evaluated.**
  - **Associativity can be either “left to right (L→R)” or “right to left (R → L)”.**

The table given in next slide provides a complete list of C operators, their precedence levels, and their rules of association.

Operator	Description	Precedence level	Associativity
() []	Bracket / Function call Array element reference	1	L→R
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical NOT Bitwise complement Pointer reference Address Sizeof operand Type casting } All are Unary	2	**R → L**
* / %	Multiplication Division Modulus	3	L→R
+ -	Addition Subtraction	4	L→R
<< >>	Left shift Right shit	5	L→R

[Cont.]

Operator	Description	Precedence level	Associativity
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	L→R
== !=	Equal to Not equal to	7	L→R
&	Bitwise AND	8	L→R
^	Bitwise XOR	9	L→R
|	Bitwise OR	10	L→R
&&	Logical AND	11	L→R
||	Logical OR	12	L→R
?:	Conditional operator	13	**R → L**
= += -= *= /= %= &= |= ^= <<= >>=	Assignment operators	14	**R → L**
,	Comma operator	15	L→R

Assignments

Complete the experiments given in “Lab Manual - Section 3”**.**

End of Chapter 3