CONSTANTS

- Constants are also known as ***literals*** in C.
- Constants are quantities whose values do not change during program execution. There are three types of constants:
  1. Numeric constants
  2. Character constants
  3. String Constants
- The ***const*** keyword is used to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

- The keyword ***const*** can be declared before or after the data type.

NUMERIC CONSTANTS

- Numeric constants are of two types:

i. Integer constants

ii. Floating point constants

Rules for numeric constants

1.Commas and blank spaces cannot be included within the constant.

2. The constant can be preceded by a minus (-) sign if desired.(to change sign of a number)

3.The value of a constant cannot exceed specified minimum and maximum bounds. For each type of constant, these bounds **will** vary from one C compiler to another.

Integer Constants

- The whole numbers which have no fraction part (decimal point) or comma is called integer constants.
- An integer represents a value that is counted like the number of students in a class.
- Negative integer constant must be preceded by a negative sign but a plus sign is optional for non-negative integers.

Integer Constants

three different number systems (decimal (base 10), octal(base 8) and hexadecimal (base 16)

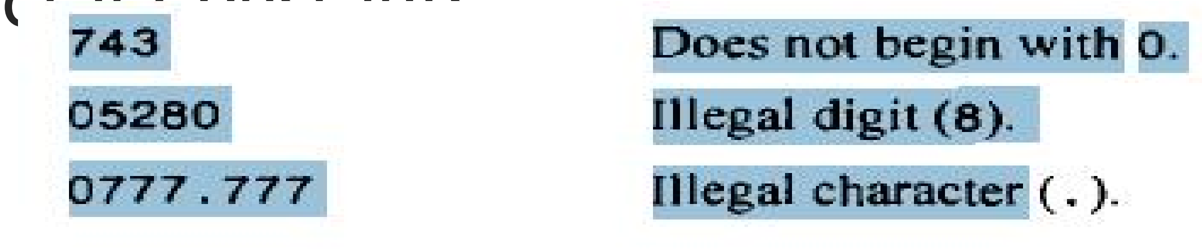
D*ecimal* integer constant - consist of any combination of digits taken from the set 0 through 9.

- Some examples of valid integer constants are:
- 0
- 765
- -987
- +4563
- Following are invalid integer constants for the reason indicated:
- 6.6758
- -56.8
- 18#3
- 082
- 1(blank) 9
- 34, 56
- In above example, character (#), blank space,(.) , (,),(-+) are illegal characters and 0 cannot be the first digits of decimal integer constants.

Octal constant

- any combination of digits taken from the set ***0*** through 7.
- However the first digit must be ***0***,in order to identify the constant as an octal number.
  - Eg:- 0270, 0743, 077

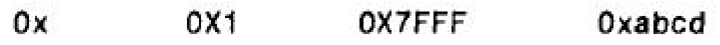
Invalid Octal Constants :



743	Does not begin with 0.
05280	Illegal digit (8).
0777.777	Illegal character (.).

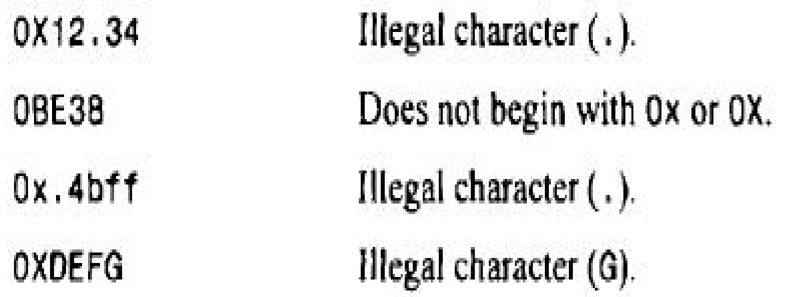
Hexadecimal integer constant

- must begin with either **Ox** or **OX** followed by any combination **of** digits taken from the sets 0 through 9 and a through f (either upper- or lowercase). EG:-

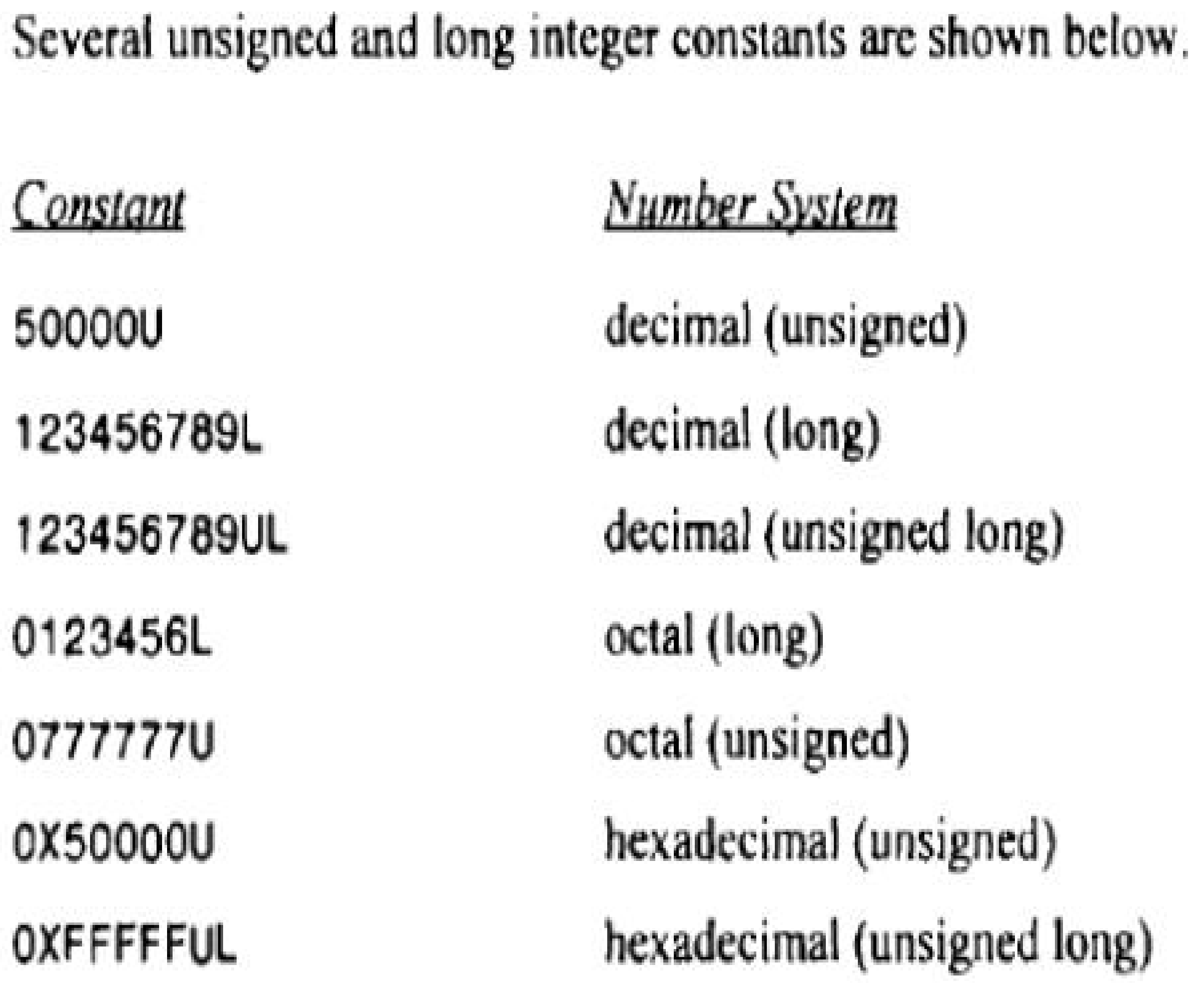


```
0x        0X1        0X7FFF        0xabcd
```

Invalid c



0X12.34	Illegal character (.).
0BE38	Does not begin with 0x or 0X.
0x.4bff	Illegal character (.).
0XDEFG	Illegal character (G).



Several unsigned and long integer constants are shown below.

Constant	*Number System*
50000U	decimal (unsigned)
123456789L	decimal (long)
123456789UL	decimal (unsigned long)
0123456L	octal (long)
0777777U	octal (unsigned)
0X50000U	hexadecimal (unsigned)
0XFFFFFUL	hexadecimal (unsigned long)

– maximum permissible values of unsigned and long integer constants will vary from one computer (and one compiler) to another.

Floating Point Constants

- Floating point constants are also called ***real constants***.
- Floating point constants are used to represent values that are measured like the height of a person which might have a value of 166.75 cm, as opposed to integer constants which are used to represent values that are counted.

- A floating point constant consists of either a fraction form or the exponent or the both. It may have either sign (+,-).
- Some valid floating points are as given;

  0.5, 11.0, 8905.2, 66668e2, -52.34,-0.123
- The following are some invalid floating point constant;
- 85———————- Missing decimal point (.)
- -1/2———————- Illegal characters (/)
- .59, 45.4——————Illegal character (.,,) and no digits to left of the decimal point.
- Some valid floating pint constants in the exponent forms are given below.
- 8.85e5——is equivalent to 885000.
- 6.6E2——-is equivalent to 660.
- 5.3e-6——-is equivalent to -530000.

The quantity 3×10^5 can be represented in C by any of the following floating-point constants.

```
300000.     3e5       3e+5     3E5       3.0e+
.3e6        0.3E6     30E4     30.E+4    300e3
```

Represent the following numbers

1.2×10^{-3}

5.026×10^{-17}

`1.2E-3` or `1.2e-3`. This is equivalent to `0.12e-2`, or `12e-4`, etc.

```
5.026E-17     .5026e-16     50.26e-18     .0005026E-13
```

Floating-point constants contd..

the magnitude might range from a minimum value of approximately 3.4E-38 to a maximum of 3.4E+38.

Some versions cover a wider range, such as 1 .7E-308 to 1 .7E+308.

0.0 is also valid

are normally represented as double-precision quantities in C. (8 bytes)

precision (the number of significant figures) will vary from one version **of** C to another. (all versions permit at least six significant figures)

Character Constants

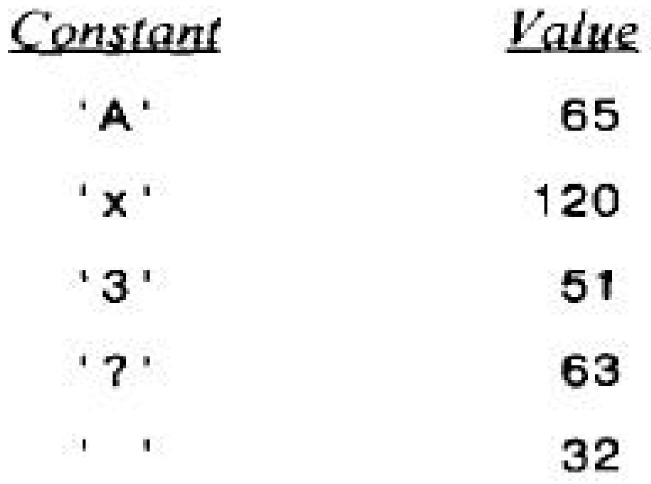
- is a single character, enclosed in apostrophes (i.e., single quotation marks).are stored as char.

```
'A'          'x'          '3'          '?'          ' '
```

- Character constants have integer values that are determined by the computer's particular character set.
- PCs use ASCII character set – each character is encoded with 7 bit combinations (128 characters)

- IBM Mainframe uses EBCDIC - 8 bit combinations (256 characters)

Character constants and their values(ASCII character set)



Constant	Value
'A'	65
'x'	120
'3'	51
'?'	63
' '	32

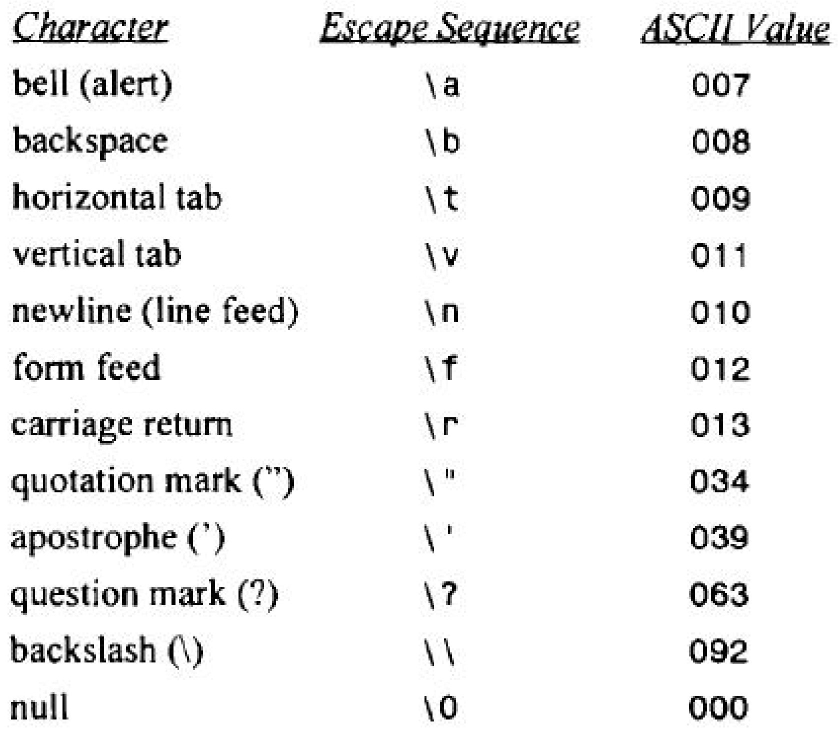
Escape sequences

Certain nonprinting characters, as well as the backslash (\) and the apostrophe (‘), can be expressed in terms of ***escape sequences***.

An escape sequence always begins with a backward slash and is followed by one or more special characters.

For example, a line feed **(LF)**, which is referred to as a ***newline*** in C, can be represented as \n.

Commonly used Escape sequences



Character	*Escape Sequence*	*ASCII Value*
bell (alert)	\a	007
backspace	\b	008
horizontal tab	\t	009
vertical tab	\v	011
newline (line feed)	\n	010
form feed	\f	012
carriage return	\r	013
quotation mark (")	\\"	034
apostrophe (')	\\'	039
question mark (?)	\\?	063
backslash (\\)	\\\\	092
null	\0	000

String constants

any number of consecutive characters (including none), enclosed in (double) quotation marks.

For example

```
“Charu”
“A”
“3/9”
“x = 5”
```

EXAMPLE 2.14 Several string constants are shown below.

```
"green"      "Washington, D.C. 20005"      "270-32-3456"

"$19.95"     "THE CORRECT ANSWER IS:"      "2*(I+3)/J"

"     "      "Line 1\nLine 2\nLine 3"      ""
```

"Line 1\nLine 2\nLine 3" would be displayed as

```
Line 1
Line 2
Line 3
```

The compiler automatically places a null character **(\O)** at the end of every string constant, as the last character within the string (before the closing double quotation mark).

This character is not visible when the string **is** displayed.

Count the characters ??

```
"\tTo continue, press the \"RETURN\" key\n"
```

38 characters !

HOW TO USE CONSTANTS IN A C PROGRAM?

.

- We can define constants in a C program in the following ways.
- By “const” keyword
- By “#define” preprocessor directive

    (Symbolic constants)

EXAMPLE PROGRAM USING CONST KEYWORD IN C

```
#include <stdio.h>
void main()
{
const int height = 100; /*int constant*/
const float number = 3.14; /*Real constant*/
const char letter = 'A'; /*char constant*/
const char letter_sequence[10] = "ABC"; /*string constant*/
const char backslash_char = '\?'; /*special char cnst*/
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);
printf("value of backslash_char : %c \n", backslash_char);
}
```

output

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

- #include <stdio.h>
-
- int main() {
-   int const RED=5, YELLOW=6, GREEN=4, BLUE=5;
-
-   printf("RED = %d\n", RED);
-   printf("YELLOW = %d\n", YELLOW);
-   printf("GREEN = %d\n", GREEN);
-   printf("BLUE = %d\n", BLUE);
-   getch();
- }
-
- This will produce following results
-
-   RED = 5
-   YELLOW = 6
-   GREEN = 4
-   BLUE = 5

SYMBOLIC CONSTANTS

- is a name that substitutes for a sequence of characters.
- The characters may represent a numeric constant, a character constant or a string constant.
- When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.
- Symbolic constants are usually defined at the beginning of a program

SYMBOLIC CONSTANTS

- **A** symbolic constant is defined by writing
    - **#define *name text***
- where ***name*** represents a symbolic name, typically written in uppercase letters.
- ***text*** represents the sequence of characters that is associated with the symbolic name.

SYMBOLIC CONSTANTS

#define P I 3.141593

:

:

area = PI * **radius * radius;**

During the compilation process, each occurrence of **PI** will be replaced by its corresponding text.Thus,the above statement will become

area = **3.141593 * radius * radius;**

SYMBOLIC CONSTANTS

Note that ***text*** does not end with a semicolon, since a symbolic constant definition is not a true C statement.

SYMBOLIC CONSTANTS

```
#define CONSTANT 6.023E23
int c =3;
. . . . .
p r i n t f ( "CONSTANT = %f", c );
Output:
CONSTANT =3
```

SYMBOLIC CONSTANTS

- The substitution of text for a symbolic constant will be carried out anywhere beyond the **#define** statement, ***except*** within a string. Thus, any text enclosed by (double) quotation marks will be unaffected by this substitution process.

- **printf ("CONSTANT = %f ”, CONSTANT) ;**
- **printf** statement would become
- **printf("CONSTANT = %f “, 6.023E23) ;**
- during the compilation process

WRITE THE ABOVE PROGRAM USING #DEFINE PREPROCESSOR DIRECTIVE IN C:

```
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
void main()
{
printf("value of height : %d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n",letter_sequence);
printf("value of backslash_char : %c \n",backslash_char);
}
```

output

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

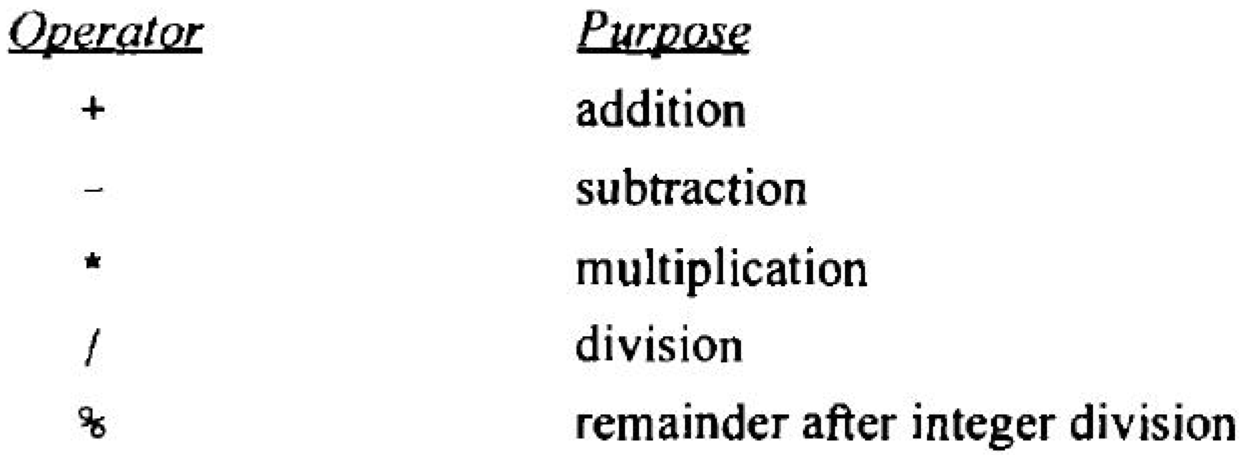
Operators

- C contains large no. of operators
- fall into different categories.

- **ARITHMETIC OPERATORS**
- **UNA RY OPERATORS**
- **RELATIONAL AND LOGICAL OPERATORS**
- **ASSIGNMENT OPERATORS**
- **THE CONDITIONAL OPERATORS**.

Arithmetic Operators

There are five ***arithmetic operators*** in C. They are



Operator	Purpose
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after integer division

There is no exponentiation operator in C. However, there is a ***library function*** **(POW)** to carry out exponentiation

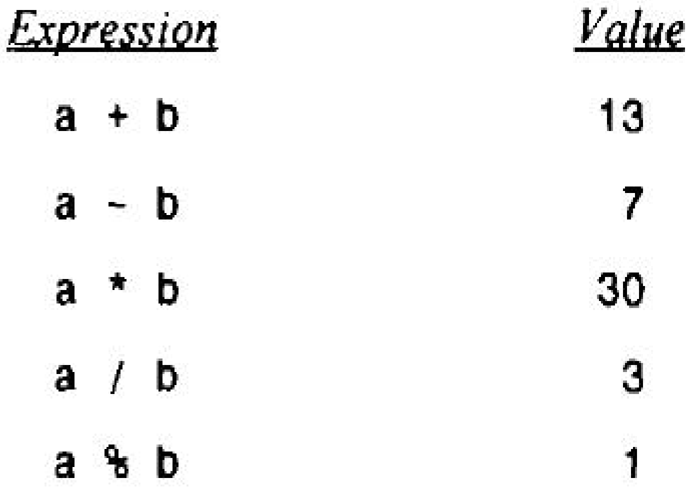
Operators

Arithmetic Operators

- % is a modulus operator. x % y results in the remainder when x is divided by y and is zero when x is divisible by y.
- Cannot be applied to float or double variables.
- Example

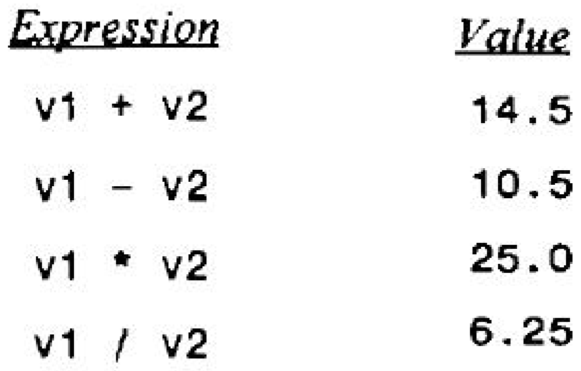
```
if ( num % 2 == 0 )
 printf(“%d is an even number\n”,
   num)’;
else
 printf(“%d is an odd number\n”,
   num);
```

```
int a,b;
a=10;
b=3;
```



Expression	*Value*
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

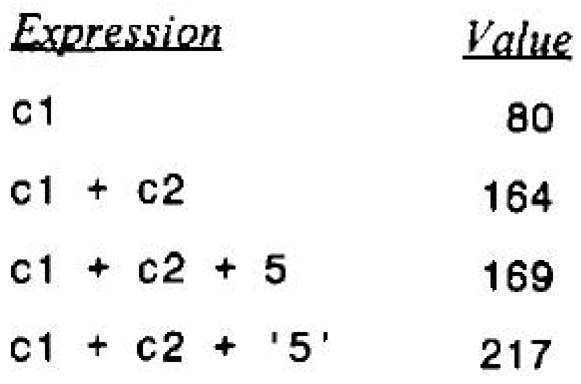
```
float v1,v2;;
v1=12.5;
v2=2.0;
```



Expression	*Value*
v1 + v2	14.5
v1 - v2	10.5
v1 * v2	25.0
v1 / v2	6.25

```
char c1,c2;;
c1='P';     (80)
c2='T';     (84)

('5'= 53)
```



Expression	*Value*
c1	80
c1 + c2	164
c1 + c2 + 5	169
c1 + c2 + '5'	217

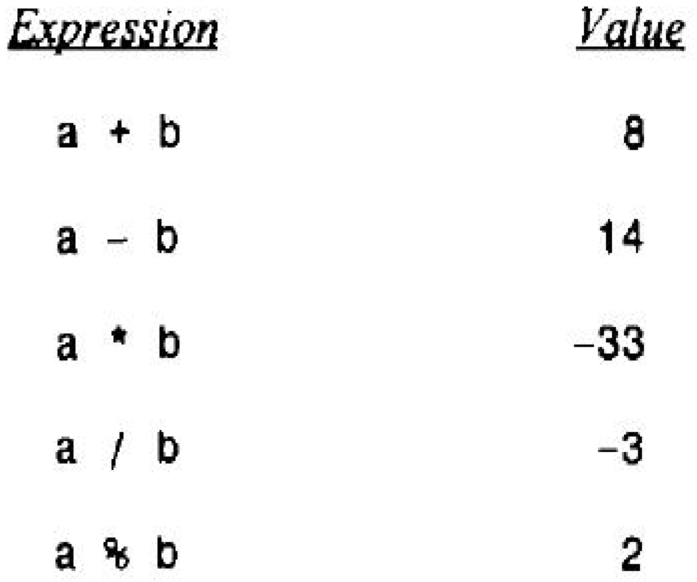
Arithmetic Operators

- The operands must represent numeric values.
- can be integer quantities, floating-point quantities or characters.
- The remainder operator (%) requires that both operands be integers and the second operand be nonzero.
- Similarly, the division operator (/) requires that the second operand be nonzero.
- Division of one integer quantity by another is referred to as ***integer division.*** This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped).
- 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 floating-point quotient.

Negative operands

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra

Let a=11, b=-3



Expression	*Value*
a + b	8
a - b	14
a * b	-33
a / b	-3
a % b	2

1.a.What would be the value of a/b when a= -11 and b=3 ?

Ans: -3

b. value of **a** % **b** = ?? Ans: -2

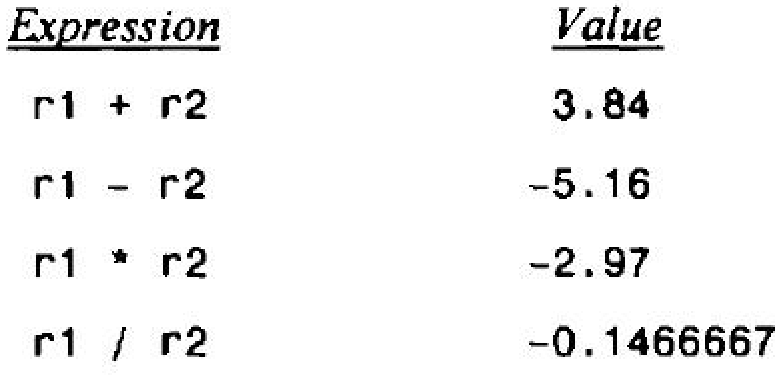
2.a.What would be the value of a/b when a= -11 and b= -3 ?

Ans: 3

b. value of **a** % **b** = ?? Ans: -2 (sign of the first operand to the remainder)

Floating-point operands(different signs

Let rl and r2 be floating-point variables whose assigned values are -0.66 and **4.50.**



Expression	*Value*
r1 + r2	3.84
r1 - r2	-5.16
r1 * r2	-2.97
r1 / r2	-0.1466667

Type Conversions

- The operands of a binary operator must have a the same type and the result is also of the same type.
- Integer division:

```
c = (9 / 5)*(f - 32)
```

The operands of the division are both int and hence the result also would be int. For correct results, one may write

```
c = (9.0 / 5.0)*(f - 32)
```

- In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called Automatic Type Casting.

```
c = (9.0 / 5)*(f - 32)
```

Automatic Type Casting

1. char and short operands are converted to int
2. Lower data types are converted to the higher data types and result is of higher type.
3. The conversions between unsigned and signed types may not yield intuitive results.
4. Example

   `float f; double d; long l;`
   `int i; short s;`
   `d + f` f will be converted to `double`
   `i / s` s will be converted to `int`
   `l / i` i is converted to `long`; `long` result

Hierarchy

Double

float

long

Int

Short and char

CONVERSION RULES

The following rules apply when neither operand is **unsigned**

1. If one of the operands is long double, the other will be converted to long double and the result will be long double.
2. Otherwise, if one of the operands is double, the other will be converted to double and the result will be double.
3. Otherwise, if one of the operands is float , the other will be converted to float and the result will be float
4. Otherwise, if one of the operands is unsigned long int , the other will be converted to unsigned long int and the result will be unsigned long int
5. Otherwise, if one of the operands is long int and the other is unsigned int , then:

(a) If unsigned int can be converted to long int , the unsigned int operand will be converted **as** such and the result will be long int .

(*b*) Otherwise, both operands will be converted to unsigned long int and the result will be unsigned long int.

6. Otherwise, if one of the operands is long int, the other will be converted to long int and the result will be long int.

7. Otherwise, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int

8. If none of the above conditions applies, then both operands will be converted to int (if necessary), and the result will be int

Note that some versions of C automatically convert all floating-point operands to double-precision.

ASSIGNMENT RULES

If the two operands in an assignment expression are of different data types, then the value of the right-hand operand will automatically be converted to the type of the operand on the left. The entire assignment expression will then be of this same data type. In addition,

1. A floating-point value may be truncated if assigned to an integer identifier.
2. A double-precision value may be rounded if assigned to a floating-point (single-precision) identifier.
3. An integer quantity may be altered (some high-order bits may be lost) if it is assigned to a shorter integer identifier or to a character identifier .

Explicit Type Casting

- The general form of a type casting operator is
- (type-name) expression
- Example

  `C = (float)9 / 5 * ( f - 32 )`
- `float` to `int` conversion causes truncation of fractional part
- `double` to `float` conversion causes rounding of digits
- `long int` to `int` causes dropping of the higher order bits.

Precedence

The operators within C are grouped hierarchically according to their ***precedence*** (i.e., order of

evaluation). Operations with a higher precedence are carried out before operations having a lower precedence.

The natural order of evaluation can be altered, however, through the use of parentheses

Precedence

The **+, -** group has lower precendence than the ***, / %** group.

```
3 - 5 * 7 / 8 + 6 / 2
  3 - 35 / 8 + 6 / 2
  3 - 4.375 + 6 / 2
  3 - 4.375 + 3
  -1.375 + 3
  1.625
```

2.UNARY OPERATORS

class of operators that act upon a single operand

UNARY MINUS (-)

INCREMENT OPERATOR(++)

DECREMENT OPERATOR(--)

SIZEOF

(TYPE) //CAST

UNARY OPERATORS

```
printf ("i = %d\n" 1);
printf("i = %d\n", i++);
printf("i = %d\n", i);OUTPUT:
1
1   //value will be altered after its use
2
```

UNARY OPERATORS

- **i=1;**
- **printf("i = %d\n", 1);**
- **printf ("1= %d\n", ++i);**
- **printf("i = %d\n", i);**
- **OUTPUT:**
- **1**
- **2 //value will be altered before its use**
- **2**

SIZEOF

Suppose that i is an integer variable, **x** is a floating-point variable, d is a double-precision variable,

and c is a character-type variable. The statements

```
p r i n t f ( "integer: %d\n" sizeof i ) ;
p r i n t f ( " f l o a t : %d\n" sizeof x ) ;
p r i n t f ("double: %d\n" sizeof d) ;
p r i n t f ( "character: %d\n" , sizeof c) ;
```

Increment and Decrement Operators

- `a++` and `++a` are equivalent to `a += 1`.
- `a--` and `--a` are equivalent to `a -= 1`.

- `++a op b` is equivalent to `a++; a op b;`
- `a++ op b` is equivalent to `aopb; a++;`
- `Example`

```
Let b = 10 then
(++b)+b+b = 33
```

SIZEOF

```
p r i n t f ( "integer: %d\n" , sizeof ( i n t e g
  e r ) ) ;
p r i n t f ( " f l o a t : %d\n", s i z e o f ( f l
  o a t ) ) ;
p r i n t f ("double: %d\n" sizeof (double));
p r i n t f ("character: %d\n" s i z e o f ( c h
  a r ) ) ;
```

OUTPUT

```
integer: 2
float: 4
double: 8
character: 1
```

3.Relational Operators

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

Closely associated with the relational operators are the two equality operators

Operator	*Meaning*
==	equal to
!=	not equal to

Relational Operators

- The expression

```
operand1 relational-operator operand2
```

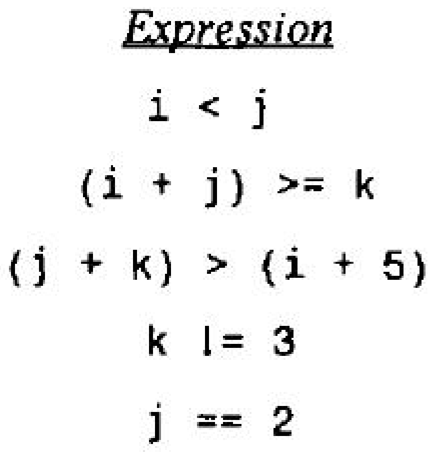
  takes a value of 1(int) if the relationship is true and 0 (int) if relationship is false.

- Example

```
int a = 25, b = 30, c, d;
c = a < b;
d = a > b;
```

  value of c will be 1 and that of d will be 0.

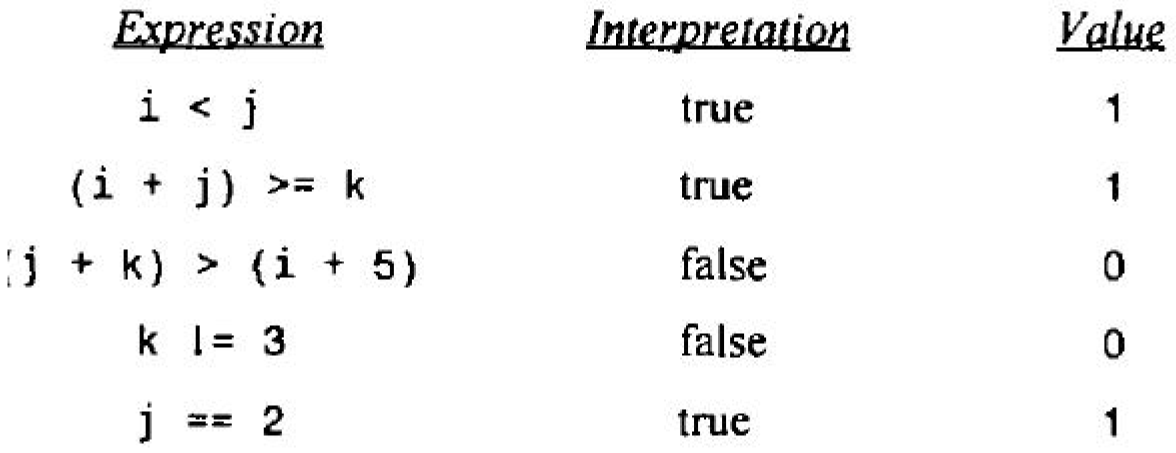
i=1,j=2,k=3



Expression

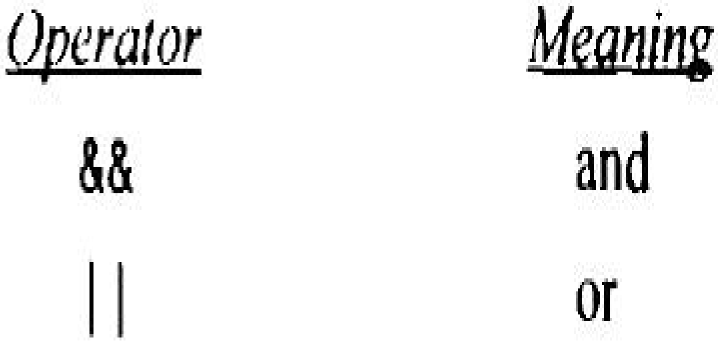
```
i < j
(i + j) >= k
(j + k) > (i + 5)
k != 3
j == 2
```

Result



Expression	*Interpretation*	*Value*
`i < j`	true	1
`(i + j) >= k`	true	1
`(j + k) > (i + 5)`	false	0
`k != 3`	false	0
`j == 2`	true	1

Logical Operators



Operator	*Meaning*
&&	and
||	or
!	**not**

Logical Operators

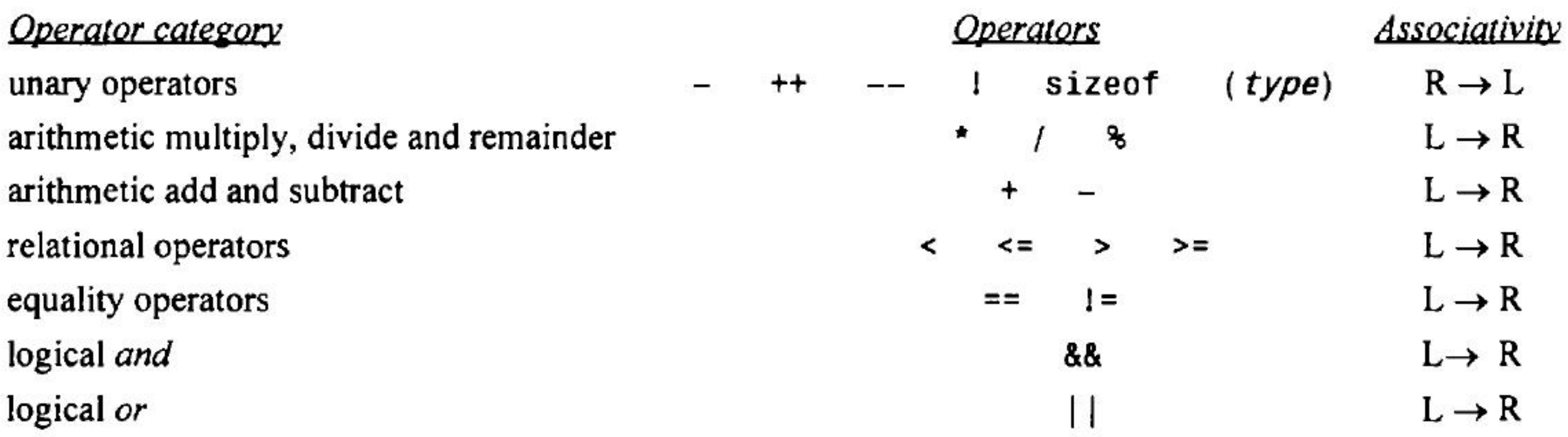
- `expr1 && expr2` has a value 1 if `expr1` and `expr2` both are true
- `expr1 || expr2` has a value 1 if either `expr1` or expr2 true or both expr1 and `expr2` are true
- `!expr1` has a value 1 if expr1 is zero else 0.
- Example
- `if ( marks >= 45 && attendance >= 85 ) grade = 'P'`
- `If ( marks < 45 || attendance < 85 ) grade = 'N'`

4.Assignment operators

- The general form of an assignment operator is
- `v op= exp`
- Where v is a variable and op is a binary arithmetic operator. This statement is equivalent to
- `v = v op (exp)`

`a = a + b`	can be written as	`a += b`
`a = a * b`	can be written as	`a *= b`
`a = a / b`	can be written as	`a /= b`
`a = a - b`	can be written as	`a -= b`

Precedence of Operators

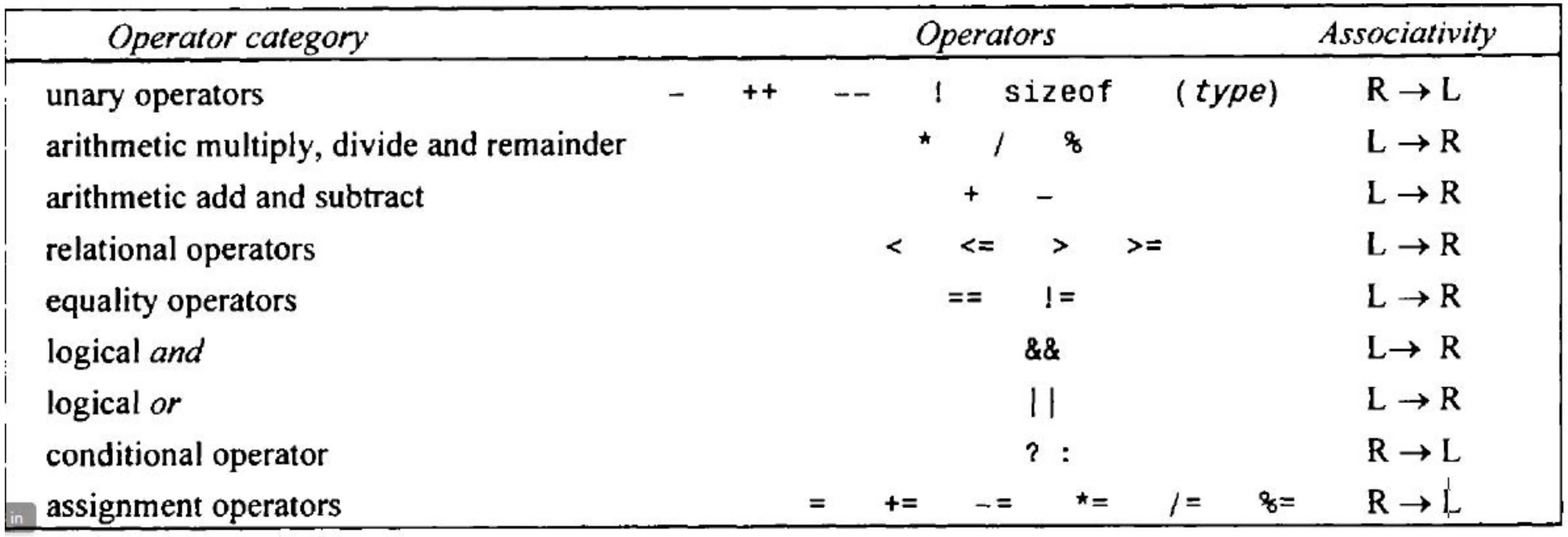


Operator category	*Operators*	*Associativity*
unary operators	`-` `++` `--` `!` `sizeof` `(`*type*`)`	R → L
arithmetic multiply, divide and remainder	`*` `/` `%`	L → R
arithmetic add and subtract	`+` `-`	L → R
relational operators	`<` `<=` `>` `>=`	L → R
equality operators	`==` `!=`	L → R
logical *and*	`&&`	L → R
logical *or*	`||`	L → R

5. CONDITIONALOPERATOR

- ***conditional operator*** **(? :)**
- ***conditional expression:***
- expression that makes use of the conditional operator
- General form:
- ***expression 1*** ? ***expression 2*** : ***expression 3***
- ***Eg:-***
- **flag** = (i < 0) ? ***0*** : **100**

Operator Precedence Groups



Operator category	*Operators*	*Associativity*
unary operators	`-` `++` `--` `!` `sizeof` `(`*type*`)`	R → L
arithmetic multiply, divide and remainder	`*` `/` `%`	L → R
arithmetic add and subtract	`+` `-`	L → R
relational operators	`<` `<=` `>` `>=`	L → R
equality operators	`==` `!=`	L → R
logical *and*	`&&`	L → R
logical *or*	`||`	L → R
conditional operator	`?` `:`	R → L
assignment operators	`=` `+=` `-=` `*=` `/=` `%=`	R → L

Suppose a, b and c are integer variables that have been assigned the values a = 8, b = 3 and c = –5. Determine the value of each of the following arithmetic expressions.

(*a*) `a + b + c`

(*b*) `2 * b + 3 * (a - c)`

(*c*) `a / b`

(*d*) `a % b`

(*e*) `a / c`

(*f*) `a % c`

(*g*) `a * b / c`

(*h*) `a * (b / c)`

(*i*) `(a * c) % b`

(*j*) `a * (c % b)`

Suppose `x`, `y` and `z` are floating-point variables that have been assigned the values `x` = 8.8, `y` = 3.5 and `z` = −5.2. Determine the value of each of the following arithmetic expressions.

(*a*) `x + y + z`

(*b*) `2 * y + 3 * (x - z)`

(*c*) `x / y`

(d) `x % y`

(*e*) `x / (y + z)`

(*f*) `(x / y) + z`

(*g*) `2 * x / 3 * y`

(*h*) `2 * x / (3 * y)`

A C program contains the following declarations:

```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

Determine the data type of each of the following expressions.

(*a*) `i + c`

(*b*) `x + c`

(*c*) `dx + x`

(*d*) `((int) dx) + ix`

(*e*) `i + x`

(*f*) `s + j`

(*g*) `ix + j`

(*h*) `s + c`

(*i*) `ix + c`