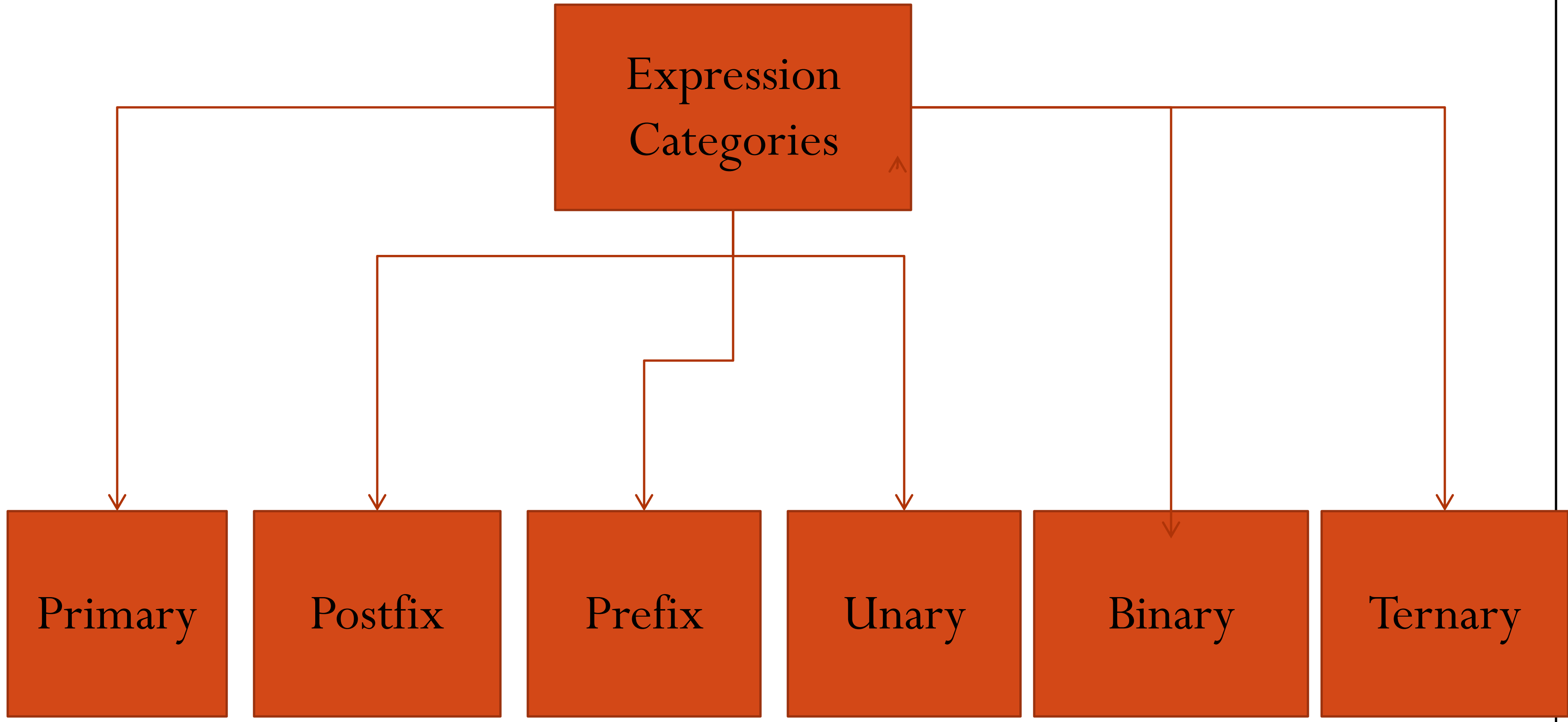
1.3a Expressions

Expressions

- An ***Expression*** is a sequence of operands and operators that reduces to a single value.
- An ***operator*** is a syntactical token that requires an action be taken
- An ***operand*** is an object on which an operation is performed; it receives an operator's action.
- A simple expression contains only one operator.
  - E.g.
    - 2+5 is a simple expression which gives 7
    - -a is a simple expression
- A complex expression – to evaluate we need to reduce it to a series of simple expressions.
- E.g.
- 2 + 5 * 7 =>2+ 35 => 37

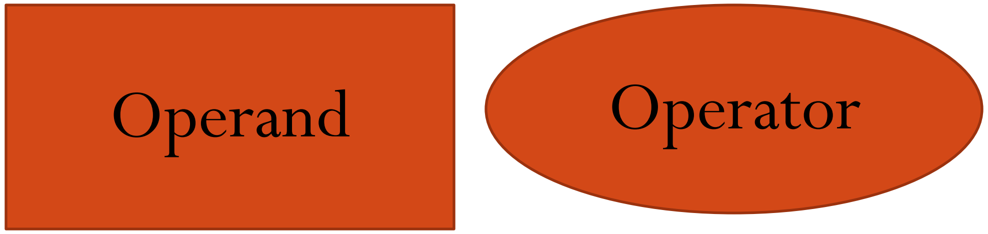
Expression Categories



Primary Expressions

- Most elementary type of expression
- A primary expression consists of only one operand with no operator
- Operands can be name, constant, a parenthesized expression
- Primary expression is evaluated first in case of precedence
- **Name**: is any identifier for a variable, function or any other object in the language. E.g.: a b12 int_max SIZE
- **Literal Constants**: It is a piece of data whose value can't be changed during the execution of the program
- E.g.: 5 123.98 'A' "Welcome"
- **Parenthetical Expressions**: Any value enclosed in parenthesis may be reducible to a single value
- E.g.: (2* 3+4) (a=23 + b * 6)

Postfix Expressions



- Postfix expression consists of one operand followed by one operator
- Postfix Increment: value is increased by 1.Thus a++ results in the variable a being increased by 1.The effect is the same as a=a+1.

a++ has the same effect as a=a+1

- The difference is that the value of the postfix increment expression is determined before the variable is increased

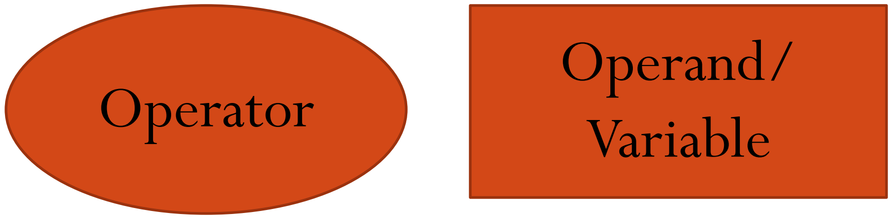
x++, where ++ appears after its operand, post-increment operator
x--, the decrement operator is called the post-decrement operator

Postfix Increment and decrement

- Postfix increment and decrement has a value and a side effect
- For instance, if a variable contains 4 before the expression is evaluated, the value of the expression is evaluated,i.e, the value of expression a++ is 4.As a result of evaluating the expression and it's side effect, a contains 5
- Postfix decrement (a--), the value of the expression is the value of a before the decrement; the side effect is the variable is decremented by 1

y = x++;
y is assigned the original value of x first,
then x is increased by 1.
y = x--; the assignment of y to the value of x takes place first,
then x is decremented.

Prefix Expressions



- Two prefix operators : Prefix increment and Prefix decrement
- The operand of a prefix expression must be a variable

++ a has the same effect as a =a+1

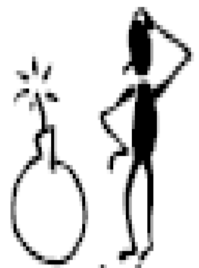
++x , where ++ appears before its operand, pre-increment operator
The operator first adds 1 to x, and then yields the new value of x

--x;, the pre-decrement operator first subtracts 1 from x and
then yields the new value of x

Example

C allows for a short hand notation that introduces side effects. This is done through the prefix- or postfix operators ++, --

If ++ or -- are used as prefixes, the variable is modified before it is used in the evaluation of an expression:



```
a = 3;
b = ++a + 3;     /* b = 4 + 3 = 7 and a = 4 side effect */
```

If ++ or -- are used as postfixes, the variable is first used to evaluate the expression and then modified.

```
a = 3;
b = a++ + 3;     /* b = 3 + 3 = 6 and a = 4 */
```

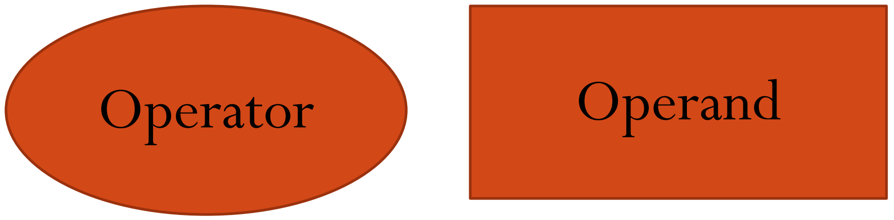
Example

```
int w, x, y, z, result;
 w = x = y = z = 1;
printf(" w = %d, x = %d, y = %d,  z = %d,\n", w, x, y, z);

result = ++w;
printf("++w evaluates to %d , w is now %d\n", result, w);

result = x++;
printf("x++ evaluates to %d , x is now %d\n", result, x);
```

Given w = 1, x = 1, y = 1, and z = 1,
++w evaluates to 2 and w is now 2
x++ evaluates to 1 and x is now 2

Unary Expression

- Consist of one operator and one operand
- Like the prefix expression, the operator comes before the operand
- But unlike the prefix expression, unary expression can have an expression or variable as the operand



sizeof

- Gives the size in bytes of a type or primary expression
- By specifying the size of an object during execution, we make our program more portable with other hardware

sizeof (int)

- sizeof -345.23
- sizeof x

Unary Plus /Minus

- If the expression's value is negative, it remains negative
- If the expression's value is positive, it remains positive

Expression	Contents of a before and after expression	Expression Value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

Cast Operator

- Cast operator converts one expression type to another

  (type_name) expression
- To convert an integer to a real number,

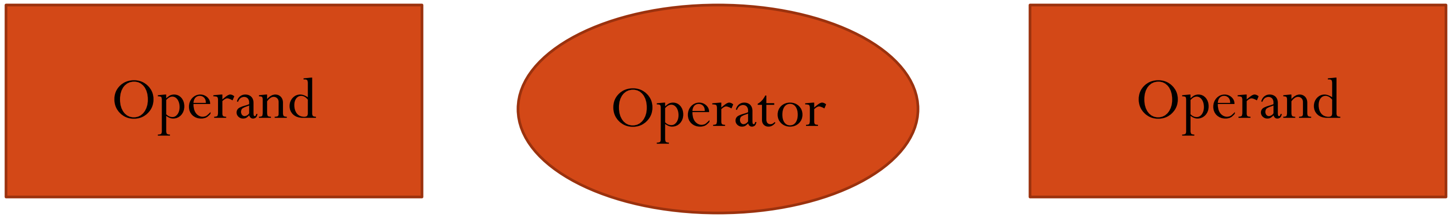
  float(x)
- Only the expression value is changed , the integer variable x is unchanged

Example :Cast operator

```
#include <stdio.h>
main()
{
 int sum = 17,
count = 5;
double mean;
 mean = (double) sum / count;
printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result −Value of mean : 3.400000

BinaryExpressions



- Binary expressions are formed by an operand-operator-operand combination
- Any two numbers added ,subtracted ,multiplied or divided are usually formed in algebraic notation, which is a binary expression
- a + b
- c - d

Multiplicative Expressions

- Which take its name from the first opearator,include the multiply, divide and modulus operator
- These operators have the highest priority among the binary operators and therefore evaluated first among them
- Result of a multiply operator (*) is the product of the two operands
- Operands can be any arithmetic type
- 10 * 3   //evaluates to 30
- true * 4   //evaluates to 4
- ‘A’ * 2   //evaluates to 130
- 22.3 * 2   //evaluates to 44.6
- (2 + 3 * I) * (1 + 2 * I)   //evaluates to -4 + 7 * I

Divide operator – Multiplicative Expressions

- Result of a divide operator (/) depends on the type of the operands.
- If one or both operands is a floating – point type, the result is a floating point quotient
- If both operands are integral type, the result is the integral part of the quotient

- 10 / 3 //evaluates to 3
- true / 4 //evaluates to 0
- 'A' / 2 //evaluates to 32
- 22.3 / 2 // //evaluates to 11.15

Modulus Operator (%) – Multiplicative Expressions

- This operator divides the first operator by second operator and returns the remainder rather than quotient
- Both operands must be integral type and the operator returns the remainder as an integer type

- 10 % 3 //evaluates to 1
- true % 4 //evaluates to 1
- 'A' % 2 //evaluates to 5
- 22.3 % 2 // Error: Modulo cannot be Floating Point

Both Operands of the modulo operator (%) must be integral types

Division and Modulus Operator

- The value of an expression with the division operator is the quotient and the value of the expression with the modulus operator is remainder
  - 3 / 5 //evaluates to 0
  - 3 % 5 //evaluates to 3
- If the integral operand is smaller than the second integral operand, the result of division is zero and the result of the modulo operator is the first operand
  - 3 / 7 //evaluates to 0
  - 3 % 7 //evaluates to 3

Additive Expressions

- Depending on the operator used, the second operand is added or subtracted from the first operand
- The operand in an additive expression can be any arithmetic types (integral or floating –point)
- Additive operators have lower precedence than multiplicative operators
- 3 + 7   //evaluates to 10
- 3 – 7   //evaluates to -4

Assignment Expressions

- Assignment Operator evaluates the operand on the right side of the operator (=) and places it's value in the variable on the left.
- The assignment expression has a value and a side effect
- The value of the total expression is the value of the expression on the right of the assignment operator (=)
- The side effect places the expression value in the variable on the left of the assignment operator

The left operand in an assignment expression must be a single variable

- There are two forms of assignment : simple and compound

Simple Assignment

- Found in algebraic expressions
- a = 5
- b = x + 1
- i = i +1

- Left variable must be able to receive it, that is , it must be a variable and not a constant

- If the left operand cannot receive a value and we assign one to it, we get a compile error

Compound Assignment

- A compound assignment requires that the left operand be repeated as part of the right expression
- Five compound assignment operators are *= ,/= ,%= ,+= ,-=
- To evaluate the compound assignment, first change it to a simple assignment, then perform the operation to determine the value of the expression

Compound Expression	Equivalent Simple Expression
x *= expression	x = x * expression
x /= expression	x = x / expression
x %= expression	x = x % expression
x += expression	x = x + expression
x -= expression	x = x - expression

Compound Expression Evaluation

- x *= y + 3

is evaluated as

x = x * (y + 3)

which, given the values x is 10 and y is 5 , evaluates to 80.

Short hand Assignment Operators

=	Assignment
* =	Multiply and assign
/ =	Divide and assign
% =	Modulo and assign
+ =	Add and assign
- =	Subtract and assign
<< =	Bitwise left shift and assign
>> =	Bitwise right shift and assign
& =	Bitwise AND and assign
^ =	Bitwise XOR and assign
| =	Bitwise OR and assign

= Assignment

- The left operand is the variable to be assigned
- The right hand side is evaluated and its type converted to the type of the variable on the left and

  stored in the variable

Assign multiple variables on a single line: This is

possible because , assignment operators return the value that was

stored in the variable

- Almost all operators can be combined with =

```
a  += b;    /* a = a + b */
a   *= b;   /* a = a * b */
```

```
a = c = 5;
 printf( "a = %dn", a );
```

c = 5 is done first,
then the value of c (which is now 5)
is assigned to a

```
a <<= c - 3;
  printf( "a = %dn", a );
```

a variable is 5 (101 in binary),
which when left shifted by 2 becomes
20 (10100 in binary)

```
a &= c;
 printf( "a = %dn", a );
```

a (20 decimal, 10100 binary)
bitwise and (keeps only the 1 bits that
are common to both numbers)
with c (5 decimal, 101 binary) results
in 4 (100 binary),
which is then stored in a.

```
int a = 8.3;
float b = 1.343f;
int c;
```

a = 8, b = 1.343000

```
printf( "a = %dn, b = %fn", a, b );
```

```
    a += 2;
    b *= a;
 printf( "a = %d, b = %fn", a, b );
```

a = 10, b = 13.430000

```
   a %= 4;
   b -= 0.43;
printf( "a = %d, b = %fn", a, b );
```

a = 2, b = 13.000000

```
int i;
i = 10;          /* Assignment */         printf("i = 10 : %d\n",i);
i++;             /* i = i + 1 */          printf ("i++ : %d\n",i);
i += 5;          /* i = i + 5 */          printf ("i += 5 : %d\n",i);
i--;             /* i = i = 1 */          printf ("i-- : %d\n",i);
i -= 2;           /* i = i - 2 */        printf ("i -= 2 : %d\n",i);
i *= 5;           /* i = i * 5 */        printf ("i *= 5 :%d\n",i);
i /= 2;           /* i = i / 2 */        printf ("i /= 2 : %d\n",i);
i %= 3;           /* i = i % 3 */                  printf   ("i   %=   3   :
   %d\n",i);
```

```
Assignment Operators
 i = 10 : 10              i++ : 11                          i += 5 : 16
 i-- : 15                 i -= 2 : 13                  i *= 5 :65
 i /= 2 : 32              i %= 3 : 2
```

Bit-Operators

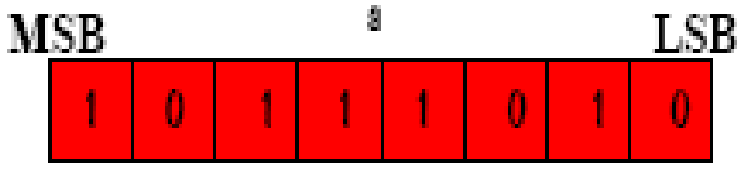
- The Bit-Operators & (AND), ^ (Exclusive-OR), and | (Inclusive-OR) manipulate bits according to standard two valued logic

Bit1	Bit2	Bit1 & Bit2	Bit1 ^ Bit2	Bit1 | Bit2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

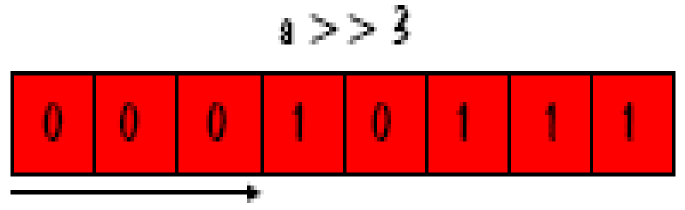
- With & one can set bits to 0.
- With ^ one can reverse the valu of bits (0 becomes 1 and 1 becomes 0)
- With | one can set bits to 1.

Shift Operators

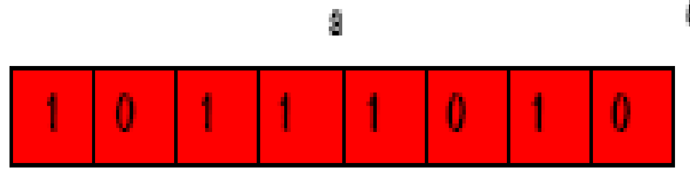
- << and >> are used to manipulate the position of the bits in a byte or a word
- With a >> b the bits in the variable a are displaced b positions to the right (the new bits are filled with 0).



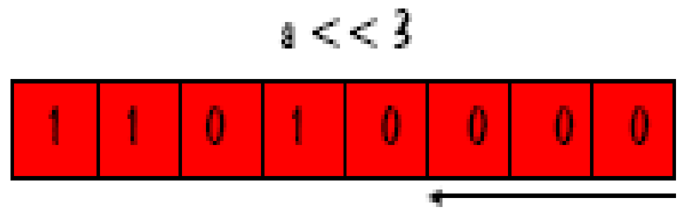
$$a / = 2^b$$



- With a << b the bits in the variable a are displaced b positions to the left (the new bits are filled with 0).

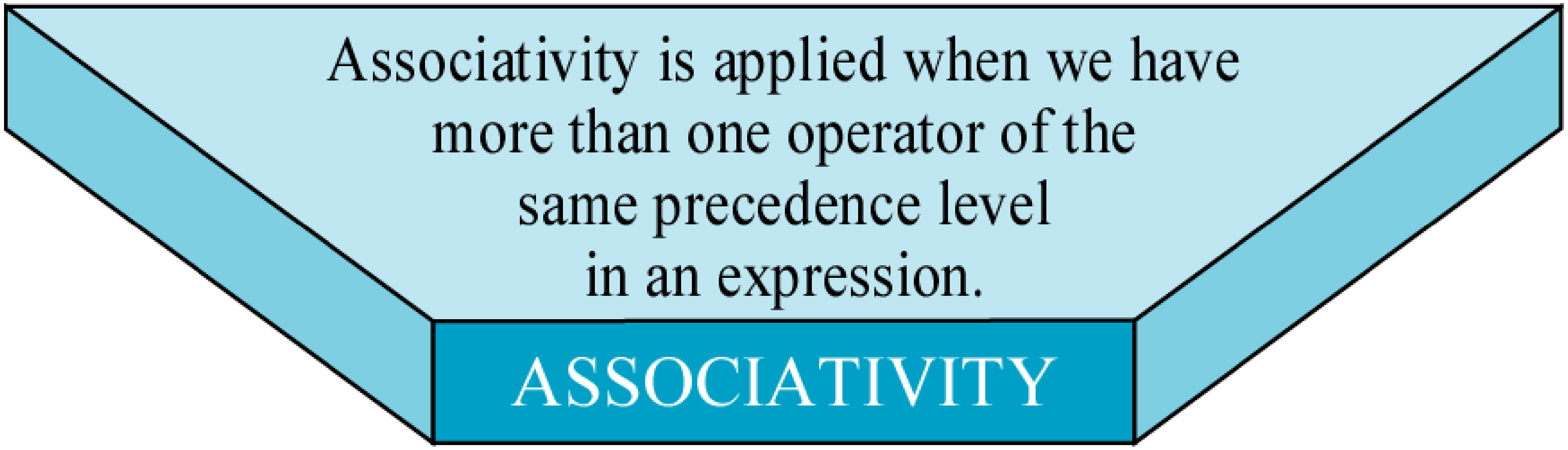


$$a * = 2^b$$



Associativity

- Associativity can be from left to right or right to left.
- Left to right associativity evaluates the expression by starting on the left and moving to the right
- Right to left associativity evaluates the expression by starting on the right and moving to the left
- Associativity is used only when the operators have same precedence.

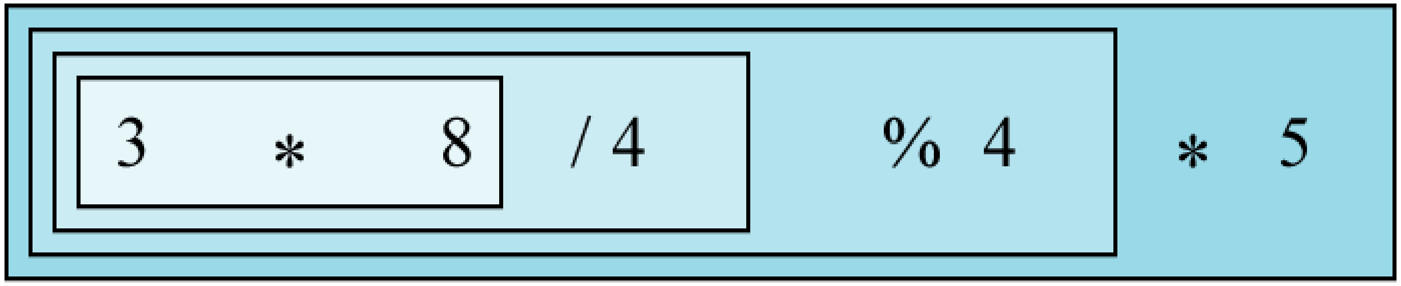


Associativity is applied when we have more than one operator of the same precedence level in an expression.

ASSOCIATIVITY

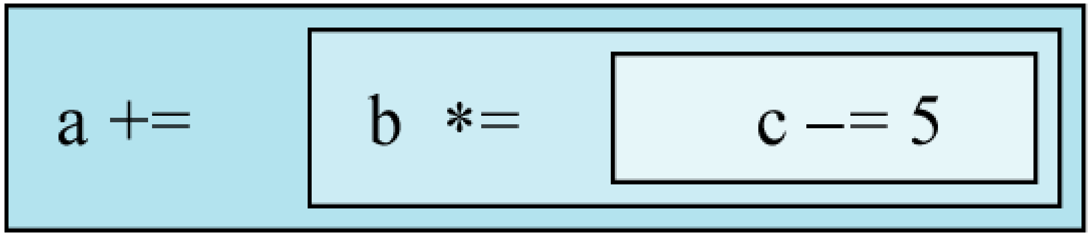
Left to Right Associativity

- (* / %) all have the same precedence
- 3 * 8 / 4 % 4 * 5
- Associativity determines how the sub expressions are grouped together. Thier associativity is from left to right



Right to Left Associativity

- E.g.- a += b *= c -= 5
- here more than one assignment operator occurs
- (a += (b *= (c -= 5))) which may be expanded to
- (a = a+ (b= b* (c =c-5)))
- (a = 3 + (b = (5 * (c = 8 – 5)))
- The value of complete expression is 18



- If we need to do a simple initializing then
- a=b=c=d=0

Example

If a programmer wishes to perform:

z = a + b / c;

it may be interpreted as ???????

z=(a+b)/c or   z=a+(b/c)

if a=3,b=6,c=2 then z=4.5 or z=6

In arithmetic expressions scanning is always done from left to right

Priority of operations :

First	Parenthesis or brackets()
Second	Multiplication & Division
Third	Addition & Subtraction
Fourth	Assignment i.e, =

An expression always reduces to a single value

Now what is the value of Z?   6

Associativity: left-to-right

Parentheses (function call)	()
Brackets (array subscript)	[]
Member selection via object name	.
Member selection via pointer	->
Postfix increment/decrement	++ --

Associativity: right-to-left

Prefix increment/decrement	++ --
Unary plus/minus	+ -
Logical negation/bitwise complement	! ~
Cast (change *type*)	(*type*)
Dereference	*
Address	&
Determine size in bytes	sizeof

Associativity: left-to-right

```
*     /     %     +     -
<<    >>    <     <=    >     >=    ==    !=
&     ^     |     &&    ||
,
```

Associativity: right-to-left

```
?:
=
+= -=
*= /=
%= &=
^= |=
<<= >>=
```

Example

- 5 + 3 * 2 is calculated as ????????

  5 + (3 * 2), giving 11,

  and not as (5 + 3) * 2, giving 16 :has higher "precedence" than + so the multiplication must be performed first

  PRECEDENCE/PRIORITY

- 8 - 3 - 2 is calculated as ????????

  (8 - 3) - 2, giving 3,

  and not as 8 - (3 - 2), giving 7:

- is "left associative", so the left subtraction must be performed first

ASSOCIATIVITY

Example

result=10+20%5-15*5/2

1. % is evaluated first
   The remainder is 0 when 5 divides 20

2. result=10+0-15*5/2
   Multiplication is performed
   15*5=75

3. result=10+0-75/2
   Division is performed

   75/2 results in 37 instead of 37.5
4. result=10+0-37
   Addition is performed because it comes before minus

5. result= 10 - 37
   Finally subtraction is performed
   and **–27** is stored in the variable **result**

Example

i= 2*3/4+4/4+8-2+5/8

i is ?????????????

i=6/4+4+8-2+5/8

i=1+4/4+8-2+5/8

i=1+1+8-2+5/8

i=1+1+8-2+0

i=2+8-2+0

i=10-2+0

i=8+0

i=8

Example

Add parentheses to the following expression to make the order of evaluation more clear:

```
year % 4 == 0 && year % 100 != 0 || year % 400 == 0
```

```
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Precedence and Associativity

- Precedence is used to determine the order in which different operators in a complex expression are evaluated.
- Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression
- Associativity determines how operators of same preference are group together to form complex expression
- Precedence is applied before associativity to determine the order in which expressions are evalaued,Associativity is then applied, if necessary

Hierarchy of operations

The preference in which arithmetic operations are performed in an arithmetic expression is called as Hierarchy of operations

Operator Precedence Chart- Highest to Lowest

Operators	Type
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

Operators higher in the chart have a higher precedence,meaning that the C compiler evaluates them first.
Operators on the same line in the chart have the same precedence.

Examples

- E.g.1:- 2+ 3 * 4
- This is actually 2 binary expressions with one addition and one Multiplication operator
- Multiplication is done first , followed by addition .The value of the complete expression is 14
- (2 + (3 * 4)) $\rightarrow$ 14
- E.g.2:- -b++
- First operation is unary minus followed by postfix increment where postfix increment has the highest precedence and is evaluated first followed by unary minus
- (- (b++))
- Assuming the value of b is 5 initially, the expression is evaluated to -5.

Side Effects

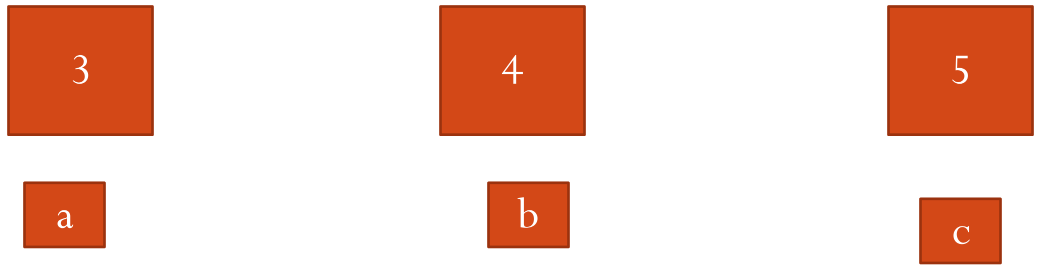
- A side Effect is an action that results from the evaluation of an expression.
- C first evaluates the expression on the right of the assignment operator and then places the value in the left variable. Changing the value of the left variable is a side effect.
- x = 4;
- Expression has 3 parts:
  - first ,on the right of the assignment operator is a primary expression that has the value 4
  - second ,the whole expression(x=4) has a value of 4
  - third , as a side effect, x receives the value 4

Example

- x = x + 4
- Assuming that x has the initial value of 3, the value of the expression on the right of the assignment operator has a value 7.The whole expression also has the value of 7.And as a side effect,x receives the value of 7
- int x =3;
- printf(“step 1–Value of x: %d\n”, x)
- printf((“step 2 - Value of x =x + 4: %d\n”,x=x+4);
- printf((“step 2 - Value of x now:%d\n”,x);

Evaluating Expressions

- Divided into Expression with side effects and Expressions without side effects
- Expressions without side effects:
- The first expression has no side effects, so the values of all of it's variables are unchanged
- a * 4 + b / 2 – c * b
- Assume that all the variables are



Evaluation of Expressions without side effects

1. Replace the variables by their values

   3 * 4 + 4 / 2 – 5 * 4

2. Evaluate the highest precedence operators and replace them with the resulting value

   (3 * 4) + (4/2) –(5* 4)

   12 + 2 – 20

3. Repeat step 2 until the result is a single value

Expression with Side Effects

--a * (3 + b) / 2 – c++ * b

1. Calculate the value of the parenthesized expression (3 + b) first

   --a * 7 / 2 – c++ * b

2. Evaluate the postfix expression (c++) next

   --a * 7 / 2 – 5 * b

3. Evaluate the prefix expression (--a)

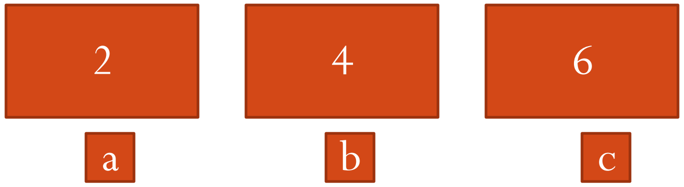
   2 * 7 / 2 – 5 * b

4. The multiply and division are now evaluated using their associativity rule, left to right

   14 /2 -5 * b -> 7 – 5 * 4 -> 7 -20

5. The last step is to evaluate the subtraction. The final expression value is -13 7-20 -> -13

After the side effects:→

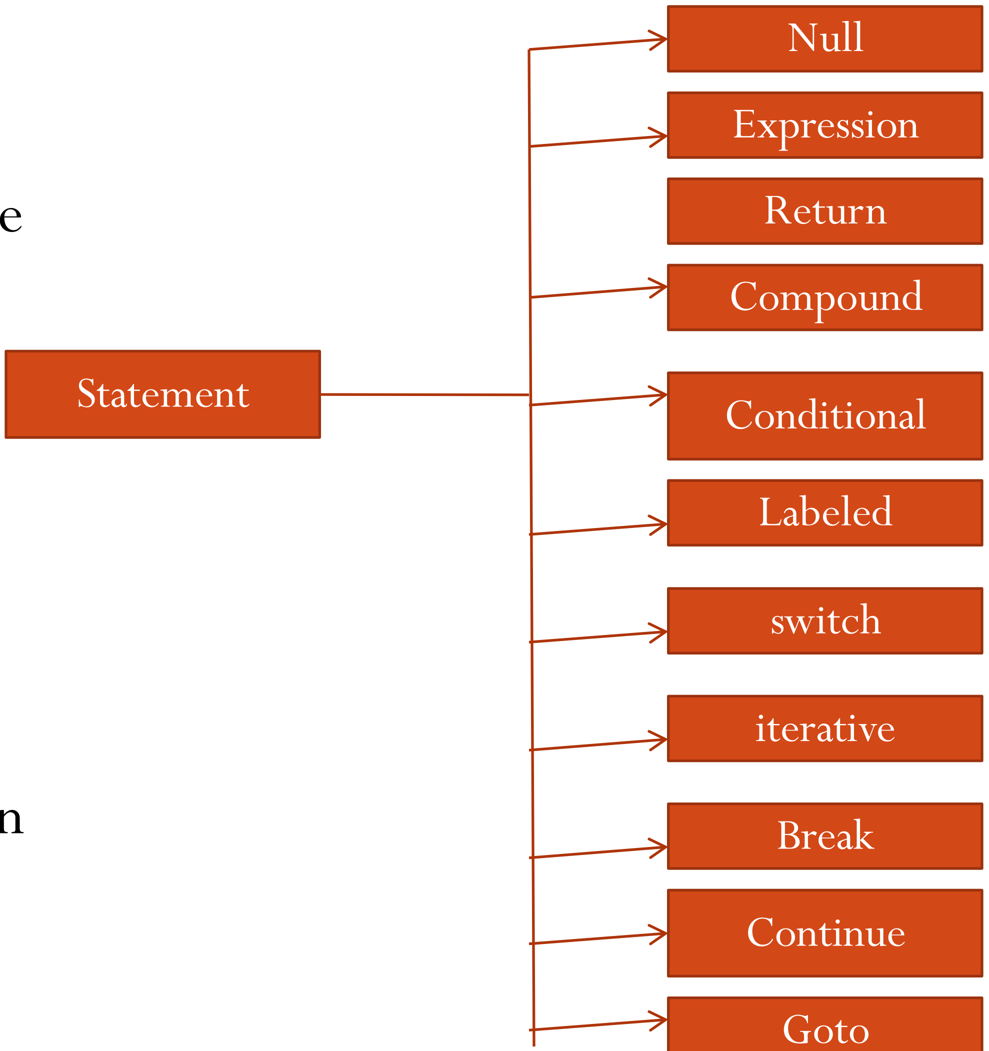


Remember….

In C,if a single variable is modified more than once in an expression, the result is undefined

Statements

- A statement causes an action to be performed by the program
- It translates directly into one or more executable computer instructions.
- Most statements need a semicolon at the end
- Types of statements are as follows:



Null statement

- It is just a semicolon (;)
- It can appear wherever a statement is expected.
- Nothing happens when a null statement is executed.

```
                ;
//null statement
```

- for (i = 0; i < 10; line[i++] = 0)

```
        ;
```

Expression Statement

- An expression is turned into a statement by placing a semicolon(;) after it
- expression;   //expression statement

- a = 2;

The effect of the expression statement is to store the value 2 in the variable a. The value of the expression is 2. After the value has been stored, the expression is terminated and the value is discarded. continues with the next statement.

Expression Statement

- a = b = 3;

The statement actually has two expressions.

a = (b =3)

The (b =3) has a side effect of assigning the value 3 to the variable b.The value of this expression is 3.Since the expression is terminated by the semicolon, it's value 3, is discarded.The effect of the expression statement, is that 3 has been stored in both a and b.

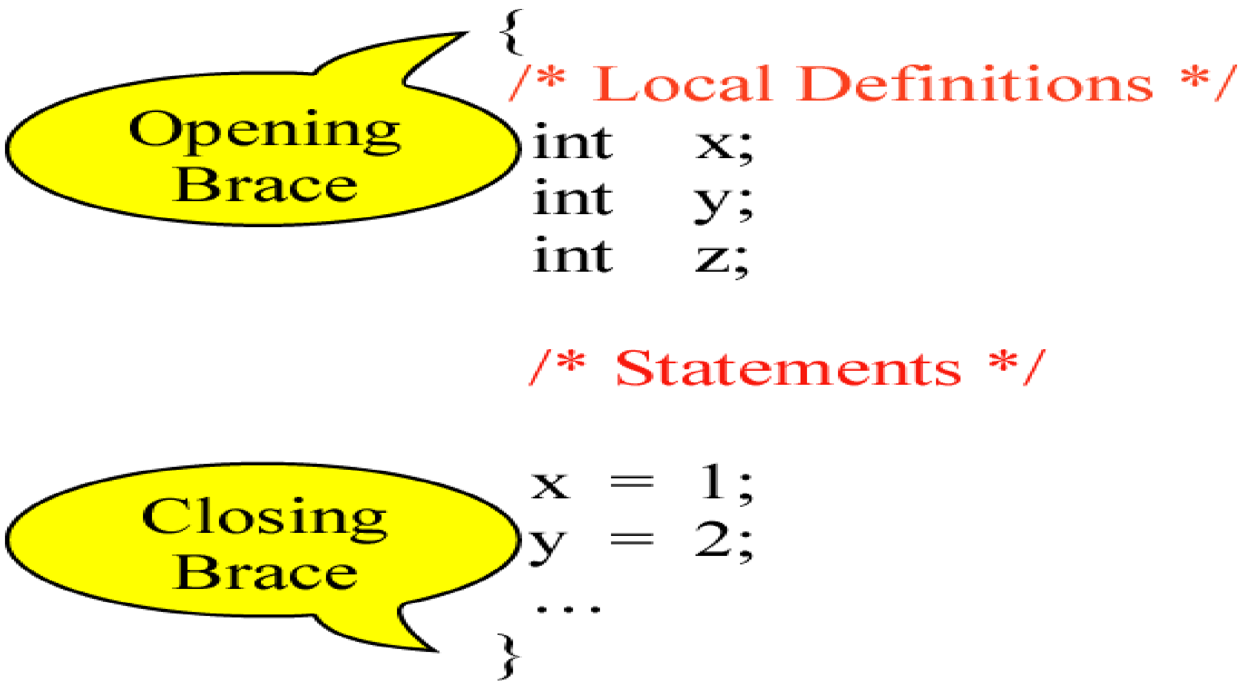
Return Statement

- A return statement terminates a function.
- All functions, including main, must have a return statement.
- When there is no statement at the end of the function, the system inserts one with a void return value.

- return statement ;   \\return statement
- return statement can return a value to the calling function
- In case of main, it returns a value to the operating system rather than to another function
- A return value of 0 tells the operating systems that the program executed successfully

Compound statements

- A compound statement is a unit of code consisting of zero or more statements. It is known as a block.
- The compound statement allows a group of statements to become one single entity
- All C functions contain a compound statement known as a function body.



```
{
/* Local Definitions */
int    x;
int    y;
int    z;

 /* Statements */

x  =  1;
y  =  2;
...
}
```

Compound Statement

- A compound statement consists of an opening brace, an optional declaration and definition section and an optional statement section, followed by a closing brace
- Compound statement does not need a semicolon.
- Both opening and closing brace acts as the delimiters
- If we put a semicolon, compiler thinks that it is an extra null statement.

Every declaration in C is terminated by a semicolon

Most statements in C are terminated by a semicolon

Any queries????