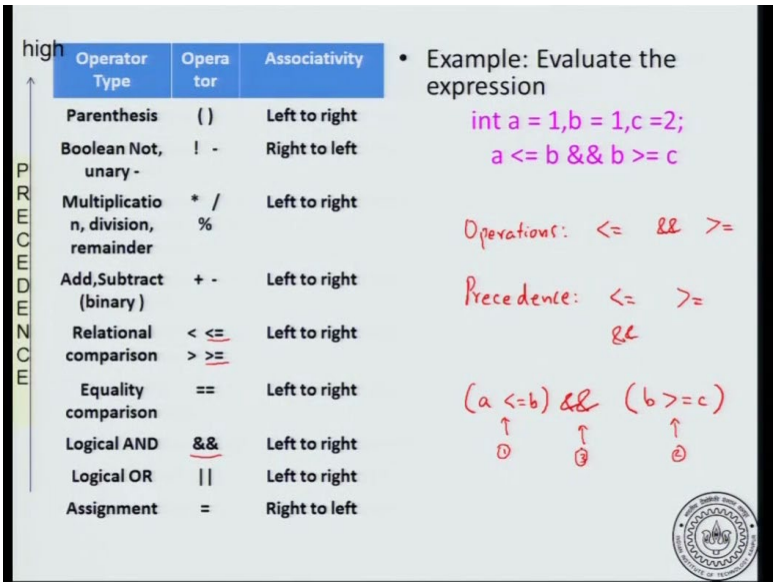
Introduction to Programming in C
Department of Computer Science and Engineering

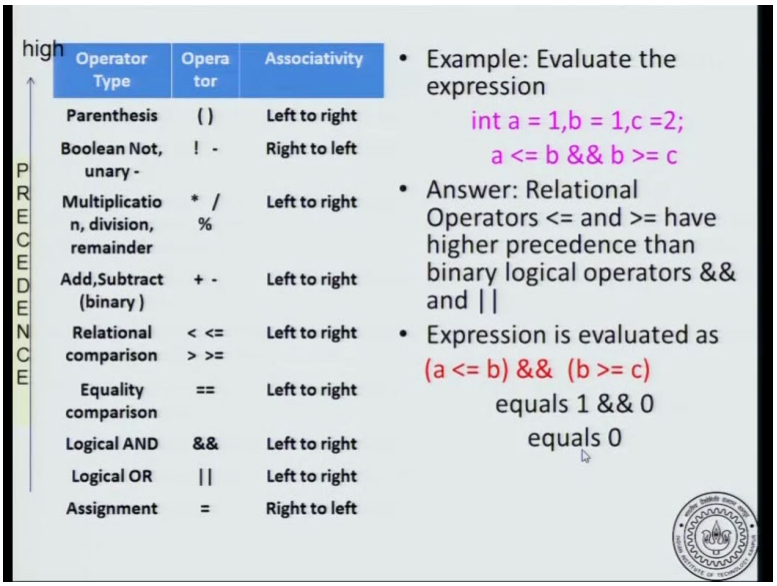
Let us see a few more examples of expression evaluation in C; what kinds of expressions are allowed, what kind of errors do people usually make, and so on.

(Refer Slide Time: 00:14)



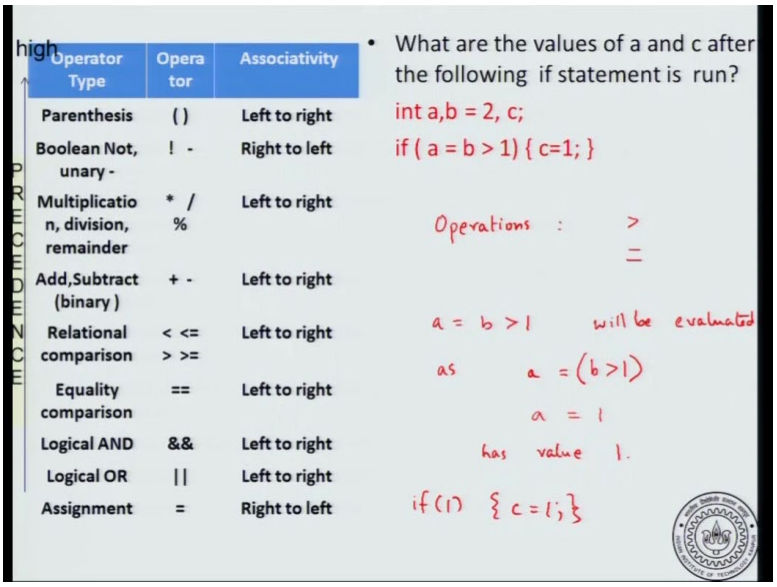
Let us say that we have given an expression a = 1, b = 1, c = 2. And then we have an expression a < b and then b >= c. So, this is the expression that we want to see how it will be evaluated. So, let us just go through it systematically. The operations on are <=, then we have the logical AND operation the >= symbol. Of these, the relational comparison operations <= and >= – have greater precedence over the logical AND. So, the precedence will be AND. And among operations of the same precedence level, we have left to right. So, whatever happens first when looking from left to right will be evaluated first. So, these two operations have the same precedence. So, we will have (a <= b); then (b >= c); these have to be done first and then AND. So, this will be done first, this will be done second, and this is the third operation. Conceptually, using just precedence and associativity rules, this is how the expression should be evaluated.

(Refer Slide Time: 01:51)



So, when we evaluate it, a < are = b is 1 < are = 1. So, that is 1. b >= c is 1 >= 2. So, that is 0. So, this becomes 1 and 0; in which case, it is 0. Now, let us look at a few tricky examples.

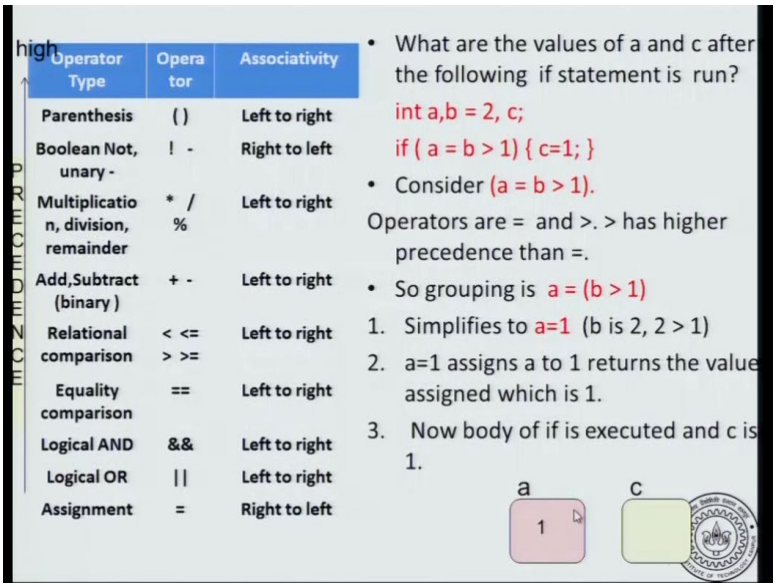
(Refer Slide Time: 02:14)



So, if you have an expression of the following form, if a = b > 1; then c = 1. So, let us see what happens here. We will do the same thing; operations sorted by precedence is… There is greater-than symbol, which has a higher precedence over the equal-to symbol. So, the expression a = b > 1 will be evaluated as b > 1, because that has higher

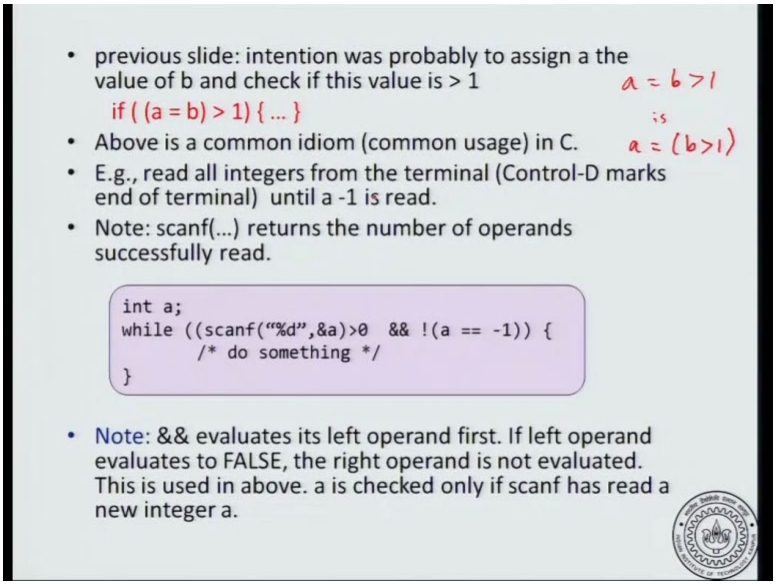
precedence. So, this goes first. And then a = b > 1. Now, b is 2. So, b > 1 is 1. So, you have a = 1. And a = 1 is an assignment expression. It assigns the value 1 to a. And the return value is 1 because a is assigned to 1.

(Refer Slide Time: 03:44)



So, then this whole if expression becomes if 1 – c = 1; in which case, we know that, c = 1; that statement will be executed.

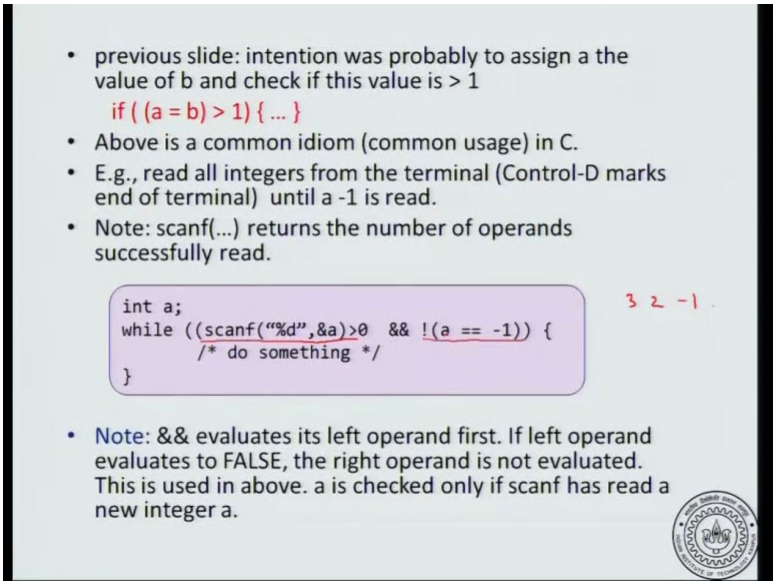
(Refer Slide Time: 03:59)



Now, typically, what is expected… The typical programming style is to say something like a assigned to b; and if that result is > 1. So, we may want to deliberately violate the

precedence. How do you do that? So, C does it some way; if you do not parenthesize it, you can always change the order of evaluation in C by introducing parenthesis, so that the meaning is very clear. So, if you do not parenthesize it, then a = b > 1 is the same as a = b > 1. But, what if you really want to do a = b and then that > 1? So, that case, you parenthesize it. Why? Because parenthesis has the highest precedence. So, whatever is within parenthesis will be evaluated first. So, = b will be evaluated first and b is 2. So, a will get the value 2. So, the assignment a = b will have returned the value 2. And 2 is > 1. So, it will execute ((Refer Time: 05:17)) One particular way in which such an expression can be seen; we have already seen such an example is – you read all integers from the terminal until a -1 is read.

(Refer Slide Time: 05:36)

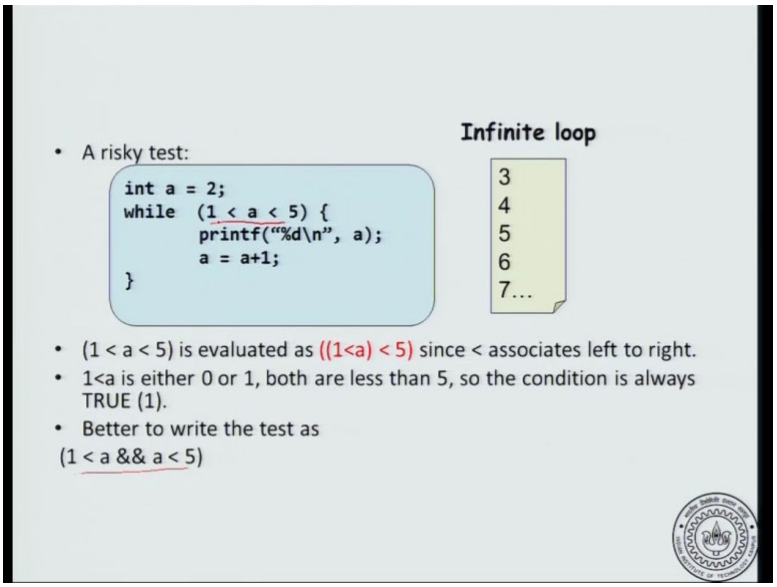


So, suppose the input is of the form 3 2 -1; and then let us say dot or something of that sort. So, what this expression does is scanf returns a value, which is the number of tokens that – number of inputs that, it was successfully able to read. So, if you try to read a character as an integer, it may not succeed. And so, as long as you have correctly written the integer and the integer is not -1, then you do a particular ((Refer Time: 06:19)) So, this is the kind of expression that is often used; where, you assign some value to a using the assignment statement. Or, maybe you want to check the return value of a function whether it is positive or not. And based on that, you want to write a condition. So, the logical and operation does operates in the following way. It evaluates the left operand first. If this condition is false, then you know that, the whole expression is going to be

false. If at least one of the terms is false, then you know that, the whole thing is false. So, it will not even evaluate the second operand.

On the other hand, if the operation is true, then it will check whether the second operand is true. If the second operand is also true, then the whole expression is true. If the second operand is false, then the whole expression is false. This method of evaluation is also called short-circuiting because it may not evaluate the whole expression in order to get the result. So, if I know that, this expression is false; then there is no need to evaluate this, because I know that, the whole expression is going to be false.

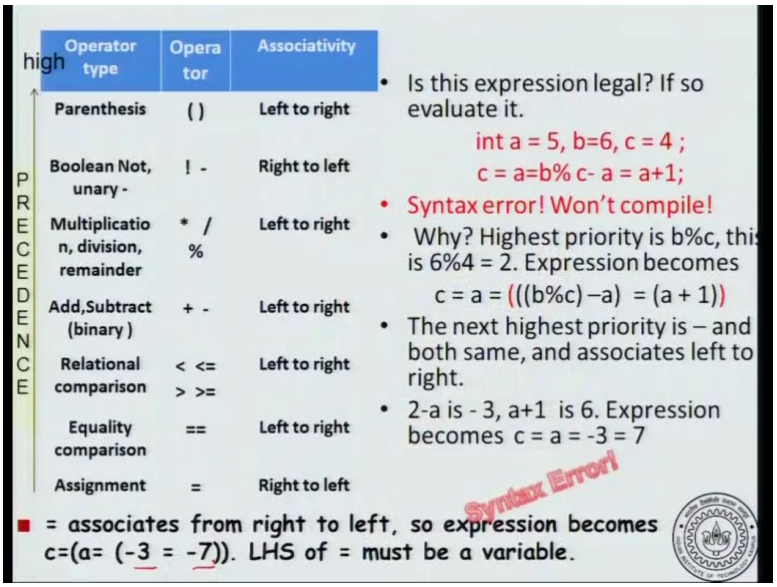
(Refer Slide Time: 07:44)



Here is a common mistake that people do, because this is similar to mathematical notation. When you want to check a condition that a is between 1 and 5; what happens if you right 1 < a < 5? Because this is the way we do it in mathematics. C will apply the precedence and the associativity. In this case, it is the same operation. So, only associativity applies. And according to associativity, it is left to right. So, this will be evaluated as 1 < a < 5. Now, a is 2. So, 1 < a is false. So, this becomes 0. So, the whole thing is 0 < 5. So, it is true. So, if you execute this code, it will eventually become an infinite loop, because this is an expression that always evaluates to true. Now, what you probably mean is that, I want to check that, a is between 1 and 5; a is 2. So, the correct way to write such an expression would be 1 < a and a < 5; that will check the between-ness condition. So, notice that, this is different from the way we normally write in

mathematics. This is how we would write such a test in mathematics. But, that will cause an infinite loop. This is because C will apply the precedence and the associativity rules and not what you think it should do.

(Refer Slide Time: 09:35)

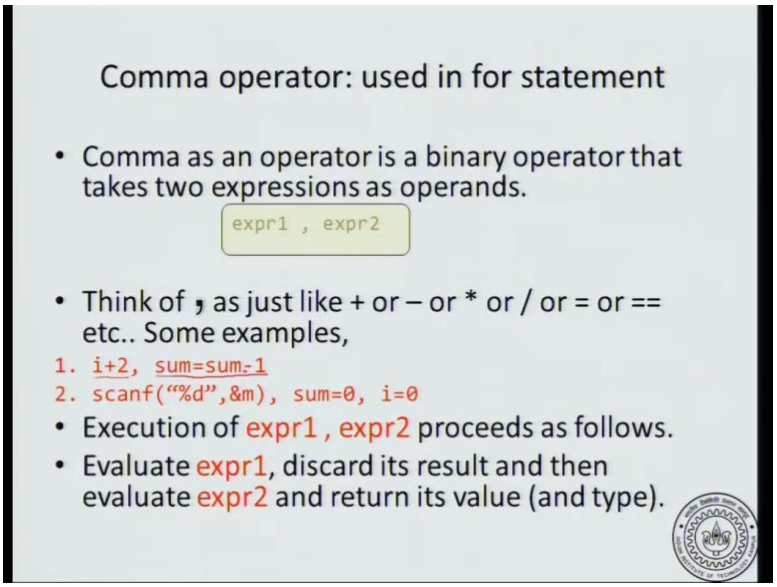


Now, let us look at can there be expressions, which make no sense? We have seen several examples, where you can always make sense out of it. So, let us take this expression. Again, list out the operations; see you have = =; then you have the % operation, which is highest precedence; then you have minus; then you again have an =; and then you have a +. So, these are the operations in the expression. So, what needs to be done first? b % c. And then you have -a; and then you have a + 1. This is by following precedence and associativity rules.

Now, we come to the assignment statement. Assignment statements are done right to left. So, the first thing that you would try to do is the following. So, you try to do the… So, here is a sub expression; here is a sub expression; here is a sub expression; and here is a sub expression. So, it is like assigning four terms. And the innermost thing will be done first; the rightmost thing will be done first. So, the rightmost assignment is b % c -a is assigned to a + 1. Now, this is a syntax error. So, what happen is as we just discussed if you work out the whole assignment; if you workout the whole expression, it becomes something like this. And somewhere when you work out the assignment from right to left, you will see that, it is trying to assign a number -3 to -7. That does not make any

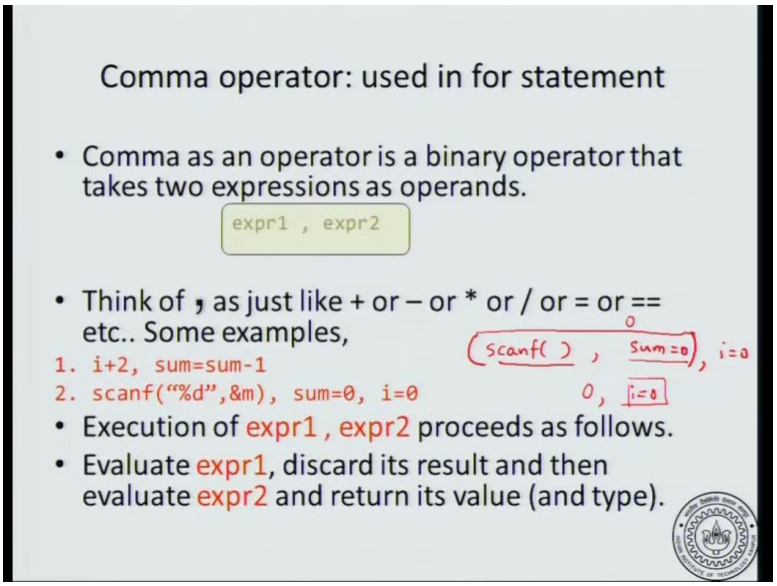
sense. The left-hand side of an assignment statement should be an assignable value, which is essentially a variable. And in this case, you are trying to assign a number to another number, which does make sense. So, here is a syntax error.

(Refer Slide Time: 11:57)



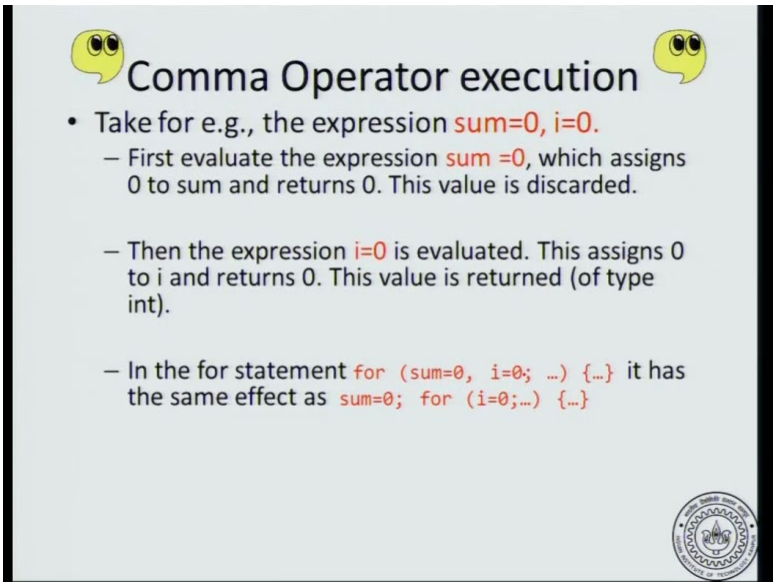
We will conclude the discussion on operations with one more operation, which is quite common in C; which is the ; operator. Now, this is not very common in mathematics. But, let us just discuss what does it mean in C. So, let us say that, we have two expressions: expression 1 and expression 2 separated by a ;. Now, think of the ; as an operation just like any other operation like + or minus. So, it must have a precedence it must have an associativity and so on. So, what will happen when we have an expression like i + 2 ; sum = sum -1. So, how does it follows? First, you evaluate the expression 1. So, first, in this case, you evaluate i + 2; then you evaluate sum = sum -1; and return the value of the lost expression. So, the whole – the ; operation is involved in an expression called the ; expression. Every expression has a value and the value of the ; expression will be expression 2.

(Refer Slide Time: 13:22)



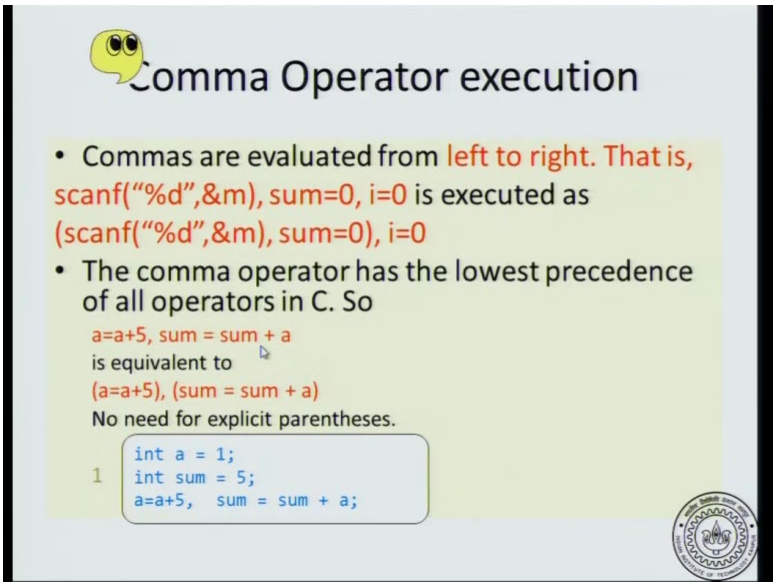
So, what if you have multiple expressions? You figure out what is the associativity of the ; expression. The ; expression associates left to right. So, this expression will become scanf and so on; sum = 0; i = 0. So, this… For the first ;, this is expression 1 and this is expression 2. So, this expression evaluates to the result of sum = 0; which is 0 as we know. Now, the second level is you have 0 ; i = 0. So, the first ; expression is evaluated and its result is expression 2 of that expression, which is a value of sum = 0, which is 0. So, the outer expression becomes 0 ; i = 0. The value of that expression is the value of expression 2 in that bigger expression, which is the value of i = 0. So, here is how you will apply the rule that, it is the value of the second expression for a more general expression involving multiple commas.

(Refer Slide Time: 14:46)



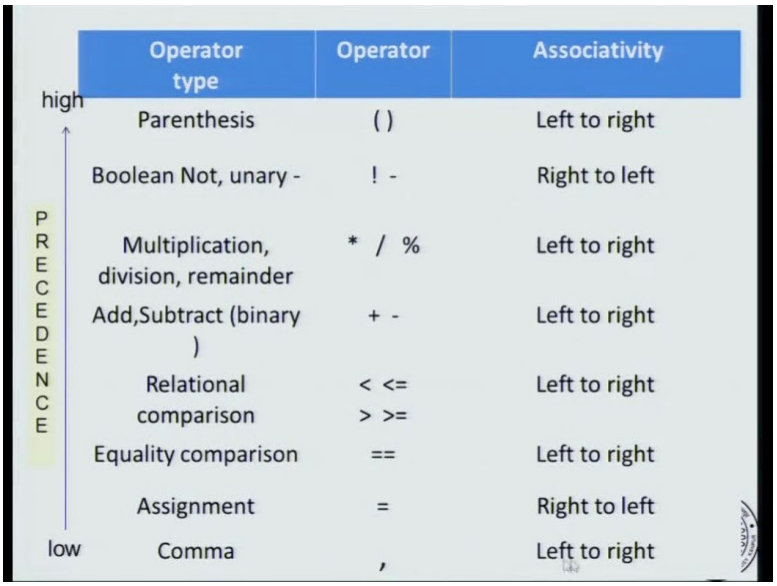
So, what you do is – first, evaluate the first expression and it has some value. For example, in this case, it is an assignment expression. So, it will have value 0. And then the second expression is evaluated. And the value of ; expression is the value of the second expression. Note that, you may… At first sight, you may see multiple commas in the same expression; but the way you do it is that, you group them using associativity rules into a sequence of ; expressions, where each ; expression has exactly two terms. This is what we did in the previous example. Now, ; expression is very convenient, because you can do things like when you want to initialize multiple variables in a for loop for example, you can just say sum = 0, ; i = 0. It will initialize both values at the same time; both variables at the same time.

(Refer Slide Time: 15:49)



So, ; are evaluated left to right. This is what I just worked out an example of the following form. So, if you have multiple sub expressions in a ; expression; if we have multiple ;, what you do is you associate them just like you did with + and star; you have multiple ; expressions. And then group them two at a time. So, it becomes two ; expressions. And then evaluate them. Now, the ; expression has the lowest precedence of any operator in C. So, if you have an operation like a = a + 5 ; sum = sum + a, what will happen is you do this expression a = a + 5; then do this expression sum = sum + a. And then evaluate the ; expression. And therefore, when you have a ; expression, you do not need explicit parenthesis, because the precedence takes care of it; it has the lowest precedence. So, it will never get swallowed into a bigger expression, which involves other operations. So, it will always be evaluated at the end.

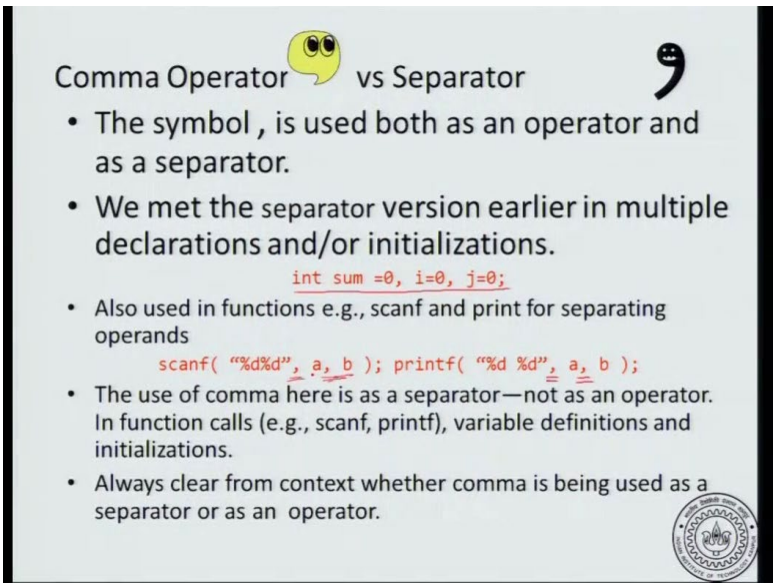
(Refer Slide Time: 17:01)



Operator type	Operator	Associativity
Parenthesis	( )	Left to right
Boolean Not, unary -	! -	Right to left
Multiplication, division, remainder	* / %	Left to right
Add,Subtract (binary)	+ -	Left to right
Relational comparison	< <= > >=	Left to right
Equality comparison	==	Left to right
Assignment	=	Right to left
Comma	,	Left to right

So, just to remind you, here is the table once again. And notice that, as we discussed the ; operation is the lowest precedence and the associates left to right.

(Refer Slide Time: 17:10)



This is also a slightly different meaning of the ; in C. We will just mention that in passing. There is also the normal separator. So, the separator can be seen in multiple occasions in C. When you initialize an expression; when you say sum = 0, ; = zero, ; j = 0; this is not the ; expression; it is just a separator as in English. So, similarly, when you call a function, you have ; to separate out the arguments. That does not mean that, the

arguments are inside a ; expression. Here ; is just a separator as in English. And it is always clear from the context whether a ; is a separator or an operator. As an operator, it has a particular value; as a separator, it does not do anything other than saying that, this first and then this. So, we have seen several operators in C and discussed the concepts of precedence and associativity. And what is important is – given the precedence and the associativity tables, can you understand an expression; see whether it is a valid expression, and if it is a valid expression, what will be its value.