

Introduction to Programming in C
Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 22

In the session, we will discuss operators and expressions. So, we have already used C expressions in our programs before.

(Refer Slide Time: 00:09)



And expressions in c are similar to expressions in mathematics and they follows tools, similar to what mathematical expressions also follow. They are a basic unit of evaluation and each expression has a value. Say, that an expression returns a value of a particular type.


(Refer Slide Time: 00:38)

C expressions

```
int a =3; int b = 4; int c;  
c = (a*a) + (b*b);
```

- The right hand side of the assignment operator =, that is, $(a*a) + (b*b)$ is an expression. It is made up of sub-expressions $a*a$ and $b*b$.
- Expressions are made up of constants and/or variables—these are the “atoms”. They then get combined using operators, unary (e.g., - or !) or binary (+ or - or * etc.).

-3 $!0$ $2+3$ $2*3$
 $2-3$



So, let us consider a few example expressions. For example, I have the following, a is 3, b is 4 and I have a variable c, which is just declared to be of type A. And then, say that c equal to a times a plus b times b. So, the right hand side of the assignment operator, this is a assignment operator. And the right hand side is an expression and that expression has sub expressions, a star a and b star b within parenthesis.

So, an expression can be made up of variables, it can be made up of constants. These are the atoms or the basic components of an expression. And sub expressions can be combined into bigger expressions, using operators. Now, operators can be unary that is, they take one argument operation. For example, on unary operators the examples can be minus, which is the unary minus. For example, minus 3 is a negative number. Similarly, NOT operator that we have seen in connection with logical operations so, NOT of zero, for example, the logical negation operator. Both of these operations take one argument. Now, there is also the binary operations like plus, minus, star, etcetera. So, plus takes two arguments. For example, an expression like 2 plus 3 and here is the binary minus. So, if I say 2 minus 3, this is actually a binary operator which takes two arguments, which are 2 and 3.

Similarly, the binary multiplication 2 star 3 would be the product of 2 and 3. So, notice the difference between... It is the same sign for the unary minus and the binary minus. But, the unary minus takes only one argument and the binary minus takes two arguments. We have used the assignment operation many times and let us understand that in, somewhat more detail.


(Refer Slide Time: 03:12)

Assignment Operator

- We have used the assignment operator many times.

```
int a;  
int b = 4;  
int c;  
b = b+1;
```
- E.g., `b = b+1`; above. If we remove the semicolon ; then we get the **expression**

```
b = b+1
```




For example, if you consider the expression b equal to b plus 1. Now, if you remove the semicolon at the end. So, the statement is b equal to b plus 1 semicolon. And if you omit the semicolon, what you get is an assignment expression, b equal to b plus 1 without the semicolon.

(Refer Slide Time: 02:28)

The Assignment Operator

- Here is an example expression involving = operator. Assume a and b are defined to be of type int.

```
a = (b = 10)
```
- Recall: assignment operator assigns to the left operand (**which must be a variable**) the value of the expression on the RHS **and then returns that value** (and type).



So, how does the assignment operation work? For example, consider an expression like a equal to and then, on the right hand side within parenthesis, b equal to 10. What does this do? So, assume that a and b are integer variables. Now, assignment assigns to the left hand variable, left hand operand, the value of the expression on the right hand side. For example, in this assignment operation there are two assignment expressions.


One is the expression b equal to 10. And the second is the expression a equal to b equal to 10. So, the first assignment expression is supposed to do the following, assign 10 to b. So, what it does is, it assigns the value of the right expression which is 10 in this case to the left hand side operand that is one thing, it does. And also, it returns the value after the assignments. So, 10 has been assigned to b. And the return value of this expression is 10.

(Refer Slide Time: 04:47)

The Assignment Operator

- Here is an example expression involving = operator. Assume a and b are defined to be of type int.

```
a = (b = 10)
```
- Evaluation of `a = (b = 10)` is done as follows.
 - First evaluate the expression in parenthesis.
 - This expression `(b=10)` has an = operator. Evaluate the expression on its RHS. Its value is 10 (constant) and of type int. So b gets the value 10. And the expression `(b=10)` evaluates to 10, this is returned.
 - The original expression now reduces to `a = 10`. This is evaluated as usual, a gets the value 10. The value 10 is returned.



So, we can now analyze a equal to b equal to 10, as follows. First, evaluate the expression in parenthesis. The expression b equal to 10 has an assignment operator. So, evaluate the expression on the right hand side and then assign it to b. Now, that operation returns. So, that expression returns a particular value, which is 10. Now, the original expression can be thought of as, just a equal to 10. This is evaluated as usual.

So, you take 10 and assign it to a. And the return value of the whole expression becomes 10. So, when executing this expression, when evaluating this expression, two variables are assigned their values. One is b, which is assigned the value 10 and the second is a, which is also assigned the value 10.


(Refer Slide Time: 05:49)

Assignment =

- Most commonly used to initialize a number of variables to the same value (similar to mathematics). Example

```
int a,b,c,d,e;  
a=b=c=d=e=0;
```
- There is ambiguity here since we did not put brackets. Should the assignment be treated as
 $((((a=b)=c)=d)=e)=0$
or should it be
 $a=(b=(c=(d=(e=0))))?$
 $a=0$
}

LHS = RHS
↑ ↑
var/const var/const
exp exp




The assignment operation can be used to initialize a number of variables, in one shot. For example, if I write a statement like a equal to b equal to c equal to d equal to e equal to 0. What does this mean? Now, here there is some ambiguity here, because we do not know which order to evaluate this. Should, we evaluate from left to right. Should, we evaluate from right to left. Does it matter?

So, should the assignment be treated as the following, where a equal to b is done first, then, equal to c, then, equal to d and. So, on. Or should it be the opposite way right to left, where e equal to 0 is first done. Then, d equal to that, then c equal to that,. So, on until a.

(Refer Slide Time: 06:39)

Evaluation from right for =

- Consider expression
 $a = (b = (c = (d = (e = 0))))$
- This assigns e,d,c,b,a successively to 0, which is the standard mathematical convention. Why?
 1. The expression in the inner most parenthesis is evaluated first. This is e=0. It assigns e to 0 and returns 0.
 2. The next outer expression is the expression d = (e=0). e=0 has been evaluated and the result is 0. So d is assigned to 0, and the expression returns 0. So on ...



So, the expression is evaluated from right to left, in the case of the assignment operation. For example, the above expression that we just saw will be done as, a equal to... So, e equal to 0, first and then backward, until a is assigned. Now, this is also the standard mathematical convention. We are not introducing a new strange rule, here. Why is this? First, we will evaluate the inner most expression, which is e equal to 0.

So, e will be assigned 0, then the return value of this sub expression. So, this sub expression will return the value 0. So, this becomes d equal to 0, d is assigned the value 0. And the return value of this sub expression becomes 0. So, then we have c equal to 0 and. So, on. So, finally, every variable here will be assigned the value 0. So, the reason for doing this is that, if you try to do it in the opposite way, you will see that uninitialized variables are initialized to other uninitialized variable.


For example, if you go from left to right, in the previous. This simply does not make any sense, because you have just declared a b c and. So, on. And when you say a equal to b, a and b are not initialized yet. So, this assignment hardly makes any sense. The basic rule of assignment is that, left hand side equal to right hand side. So, the left hand side is some value that can be assigned to.

For example, this is a variable. The right hand side can be anything, variable, constant or it can be an expression. So, all these are valid assignment. So, what is an invalid assignment? So, a equal to 0 can be a valid assignment but, 0 equal to a. So, the assignment operation is evaluated, right to left.

(Refer Slide Time: 09:11)

Associativity of Operators

- C defines the assignment operator = to be right-associative. That is,
$$a=b=c=d=0$$
has the same meaning as
$$a=(b=(c=(d=0)))$$
- Operator binary + is left associative. That is,
$$a + b + c + d$$
is evaluated by grouping from the left as
$$(((a+b)+c)+d)$$
- Associativity of an operator tells the order in which to evaluate operators if there are multiple occurrences of identical operators in an expression.

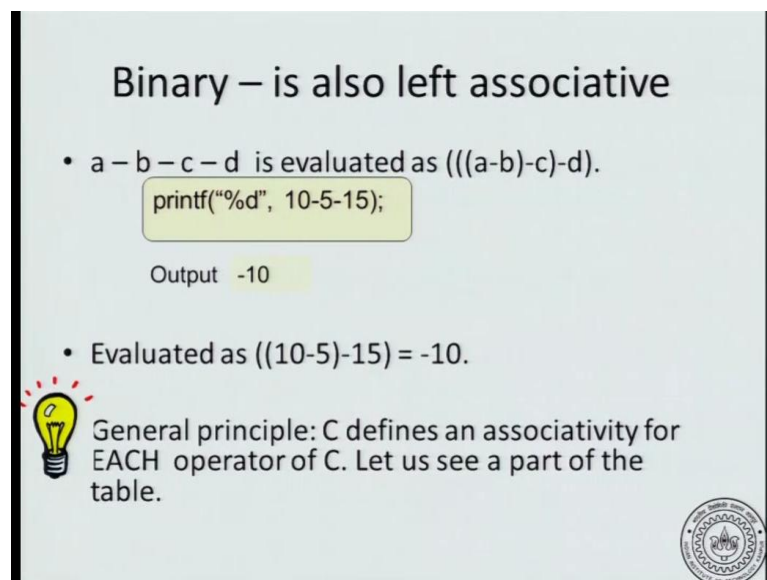


Now, we have the concept of associativity of operators. So, what does associativity mean? It is, we have just argued that, a equal to b equal to c equal to d equal to 0. An expression like that will be evaluated from right to left. So, it is as though, we have parenthesized the expression as d equal to 0, inner most. Then, c equal to that, then b equal to that and. So, on. So, on the other hand, if you take an operator like binary plus the addition symbol, then the usual custom is that you parenthesis from left to right.

So, the evaluation is done, a plus b first. Then, that sum is added to c. Then, that is added to d. So, the assignment operation goes right to left. The addition symbol operates left to right. So, this concept of associativity of an operator tells us, the order in which we evaluate the operations, if there are multiple occurrences of the same operator. So, the first there are multiple occurrences of the equal to sign. In the second, there are multiple occurrences of the addition symbol..

So, associativity rules tells you that, if there are identical operators in an expression, in which order do you evaluate them? Do you evaluate them from left to right? If you do, then it is called a left associative operator. If you evaluate from right to left, in the case of, for example, the assignment, then it is called a right associative operator.

(Refer Slide Time: 11:00)





Binary – is also left associative

- $a - b - c - d$ is evaluated as $((a-b)-c)-d$.

```
printf("%d", 10-5-15);
```


Output -10
- Evaluated as $((10-5)-15) = -10$.

 General principle: C defines an associativity for EACH operator of C. Let us see a part of the table.



Binary minus is also left associative. For example, a minus b minus c minus d is evaluated as a minus b, then c then minus c, then minus d. So, for example, if you say 10 minus 5 minus 15, what will be done is 10 minus 5 and then minus 15. So, this is 5 minus 15, which is minus 10. Whereas, if the parenthesisation had been in the opposite


way, it would be 10 minus 5 minus 15, which case it could be 10 minus, this is minus 10 which is 20.

Notice that, this is not how you are supposed to do it, even in mathematics. So, the way that c does handles the associativity of the binary operation, is correct. So, the correct parenthesisation is 10 minus 5 and then minus 15. In general, for every operator c defines an associativity. So, let us see the part of the associativity of operations in c .

(Refer Slide Time: 12:23)

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
Logical AND	&&	Left to right
Logical OR		Left to right
Assignment	=	Right to left

Keep this table handy while programming. Useful to know precedence among common ops.



There are several operations that we have seen,. So, far. The parenthesis, the Boolean naught, the logical naught and the unary minus, the binary multiplication, division and. So, on. Addition symbol, comparison less than, less than or equal to and. So, on. Equality, logical AND, logical OR and then the assignment operator. We have seen, all these operations,. So, far. And of this, the typical associativity is left to right.

There are couple of exceptions, one we have already seen. Assignment operation is right to left. The unary operations are also right to left. Most of the other operations are left to right. So, if you think for a little bit, you can see that the associativity for unary operations is also easily seen to be right to left. That makes more sense.

So, the idea is not that you should memorize this table but, you should understand. Given the table, can I understand, what will happen with an expression? How c will evaluate it? It is not that, you should remember this. But, rather if you are given the table and an expression, can you correctly calculate what the value of the expression will be.