

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

**Lecture - 24**

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)

PRECEDENCE	Operator Type	Operator	Associativity
high	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

• Example: Evaluate the expression


```
int a = 1, b = 1, c = 2;
a <= b && b >= c
```

Operations: <= && >=

Precedence: <= >= &&

(a <= b) && (b >= c)

↑            ↑            ↑  
①            ②            ③




Let us say that we have given an expression a equal to 1, b equal to 1, c equal to 2. And then we have an expression a less than b and then b greater than or equal to 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 less than or equal to, then we have the logical AND operation the greater than or equal to symbol. Of these, the relational comparison operations less than or equal to and greater than or equal to – 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 less than or equal to b within parenthesis; then b greater than or equal to c within parenthesis; 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)

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

- Example: Evaluate the expression  
`int a = 1, b = 1, c = 2;`  
`a <= b && b >= c`
- Answer: Relational Operators <= and >= have higher precedence than binary logical operators && and ||
- Expression is evaluated as  
`(a <= b) && (b >= c)`  
 equals 1 && 0  
 equals 0




So, when we evaluate it, a less than are equal to b is 1 less than are equal to 1. So, that is 1. b greater than or equal to c is 1 greater than or equal to 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)

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

- What are the values of a and c after the following if statement is run?  
`int a, b = 2, c;`  
`if (a = b > 1) { c = 1; }`
- Operations : > =
- `a = b > 1` will be evaluated  
 as `a = (b > 1)`  
`a = 1`  
 has value 1.
- `if (1) { c = 1; }`



So, if you have an expression of the following form, if a equal to b greater than 1; then c equal to 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)

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

high

- What are the values of `a` and `c` after the following if statement is run?
 

```
int a,b = 2, c;
if ( a = b > 1 ) { c=1; }
```
- Consider `(a = b > 1)`. Operators are `=` and `>`. `>` has higher precedence than `=`.
  - So grouping is `a = (b > 1)`
  - 1. Simplifies to `a=1` (`b` is 2, `2 > 1`)
  - 2. `a=1` assigns `a` to 1 returns the value assigned which is 1.
  - 3. Now body of if is executed and `c` is 1.

a 1      c

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

(Refer Slide Time: 03:59)

- previous slide: intention was probably to assign a the value of b and check if this value is > 1

```
if (a = b) > 1 { ... }
```

*a = b > 1*

*is*

*a = (b > 1)*

- Above is a common idiom (common usage) in C.
- E.g., read all integers from the terminal (Control-D marks end of terminal) until a -1 is read.
- Note: scanf(...) returns the number of operands successfully read.

```
int a;
while ((scanf("%d",&a)>0 && !(a == -1)) {
    /* do something */
}
```

- Note: && evaluates its left operand first. If left operand evaluates to FALSE, the right operand is not evaluated. This is used in above. a is checked only if scanf has read a new integer a.



Now, typically, what is expected... The typical programming style is to say something like `a = b`; and if that result is greater than 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 greater than 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 > 1` 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 greater than 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 minus 1 is read.


(Refer Slide Time: 05:36)

- previous slide: intention was probably to assign `a` the value of `b` and check if this value is `> 1`  
`if ( (a = b) > 1) { ... }`
- Above is a common idiom (common usage) in C.
- E.g., read all integers from the terminal (Control-D marks end of terminal) until `a -1` is read.
- Note: `scanf(...)` returns the number of operands successfully read.

```
int a;
while ((scanf("%d",&a)>0 && !(a == -1)) {
    /* do something */
}
```

3 2 -1 .

- Note: `&&` evaluates its left operand first. If left operand evaluates to FALSE, the right operand is not evaluated. This is used in above. `a` is checked only if `scanf` has read a new integer `a`.



So, suppose the input is of the form 3 2 minus 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 minus 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)

The slide contains the following content:

- A risky test:

```
int a = 2;
while (1 < a < 5) {
    printf("%d\n", a);
    a = a+1;
}
```
- Infinite loop**  
3  
4  
5  
6  
7...
- (1 < a < 5) is evaluated as ((1 < a) < 5) since < associates left to right.
  - 1 < a is either 0 or 1, both are less than 5, so the condition is always TRUE (1).
  - Better to write the test as (1 < a && a < 5)

A circular logo of Anna University is visible in the bottom right corner of the slide.

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 less than a less than 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 less than a less than 5. Now, a is 2. So, 1 less than a is false. So, this becomes 0. So, the whole thing is 0 less than 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 less than a and a less than 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)

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


PRECEDENCE ↑

high

- Is this expression legal? If so evaluate it.
  - `int a = 5, b=6, c = 4 ;`
  - `c = a=b%c - a = a+1;`
  - Syntax error! Won't compile!**
  - Why? Highest priority is `b%c`, this is `6%4 = 2`. Expression becomes `c = a = (((b%c) - a) = (a + 1))`
  - The next highest priority is `-` and both same, and associates left to right.
  - `2-a` is `-3`, `a+1` is `6`. Expression becomes `c = a = -3 = 7`

**Syntax Error!**

■ = associates from right to left, so expression becomes `c=(a= (-3 = -7))`. LHS of = must be a variable.



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 equal to twice; then you have the modulo operation, which is highest precedence; then you have minus; then you again have an equal to; and then you have a plus. So, these are the operations in the expression. So, what needs to be done first? b modulo c. And then you have minus a; and then you have a plus 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 modulo c minus a is assigned to a plus 1. Now, this is a syntax error. So, what happens 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 minus 3 to minus 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)

### Comma operator: used in for statement

- Comma as an operator is a binary operator that takes two expressions as operands.  

`expr1 , expr2`
- Think of `,` as just like `+` or `-` or `*` or `/` or `=` or `==` etc.. Some examples,
  1. `i+2, sum=sum-1`
  2. `scanf("%d",&m), sum=0, i=0`
- Execution of `expr1, expr2` proceeds as follows.
- Evaluate `expr1`, discard its result and then evaluate `expr2` and return its value (and type).




We will conclude the discussion on operations with one more operation, which is quite common in C; which is the comma 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 comma. Now, think of the comma as an operation just like any other operation like plus 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 plus 2 comma sum equal to sum minus 1`. So, how does it follows? First, you evaluate the expression 1. So, first, in this case, you evaluate `i plus 2`; then you evaluate `sum equal to sum minus 1`; and return the value of the lost expression. So, the whole – the comma operation is involved in an expression called the comma expression. Every expression has a value and the value of the comma expression will be expression 2.



(Refer Slide Time: 13:22)

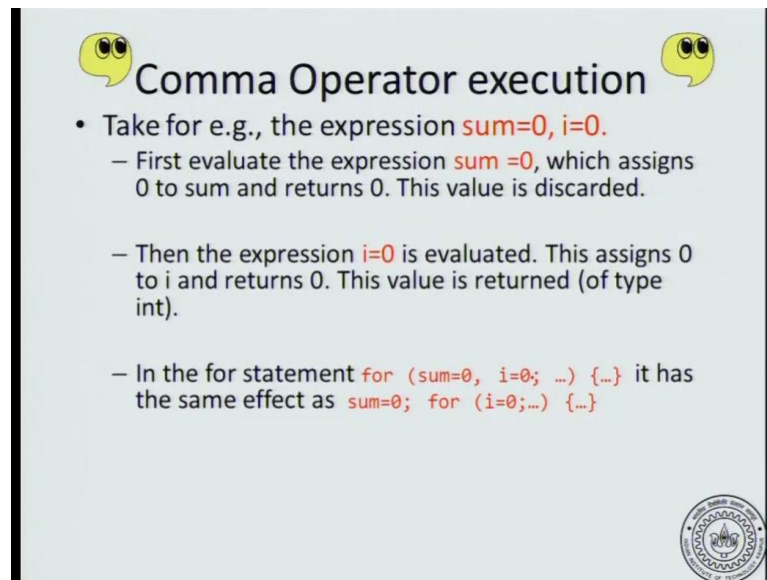
Comma operator: used in for statement

- Comma as an operator is a binary operator that takes two expressions as operands.  
`expr1 , expr2`
- Think of `,` as just like `+` or `-` or `*` or `/` or `=` or `==` etc.. Some examples,
  - `i+2, sum=sum-1`
  - `scanf("%d",&m), sum=0, i=0`  
`0, i=0`
- Execution of `expr1, expr2` proceeds as follows.
- Evaluate `expr1`, discard its result and then evaluate `expr2` and return its value (and type).



So, what if you have multiple expressions? You figure out what is the associativity of the comma expression. The comma expression associates left to right. So, this expression will become scan f and so on; sum equal to 0; i equal to 0. So, this... For the first comma, this is expression 1 and this is expression 2. So, this expression evaluates to the result of sum equal to 0; which is 0 as we know. Now, the second level is you have 0 comma i equal to 0. So, the first comma expression is evaluated and its result is expression 2 of that expression, which is a value of sum equal to 0, which is 0. So, the outer expression becomes 0 comma i equal to 0. The value of that expression is the value of expression 2 in that bigger expression, which is the value of i equal to 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)

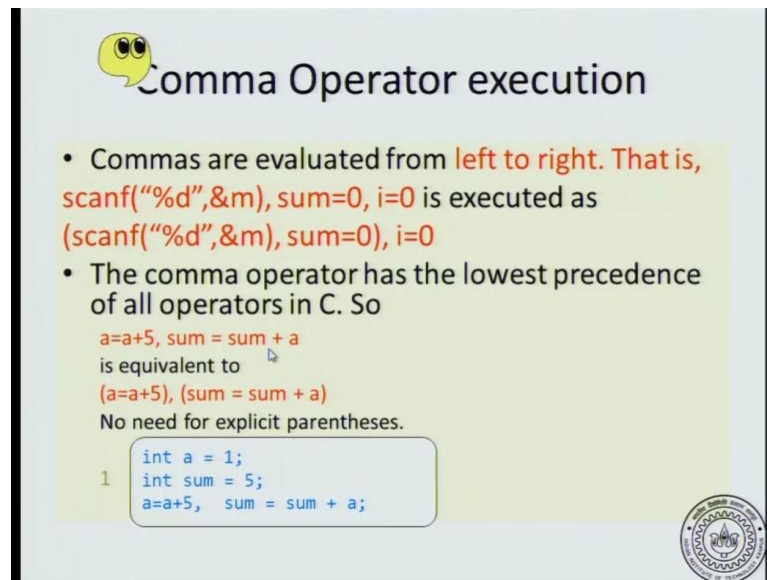


## Comma Operator execution

- Take for e.g., the expression `sum=0, i=0`.
  - First evaluate the expression `sum =0`, which assigns 0 to sum and returns 0. This value is discarded.
  - Then the expression `i=0` is evaluated. This assigns 0 to i and returns 0. This value is returned (of type int).
  - In the for statement `for (sum=0, i=0; ...) {...}` it has the same effect as `sum=0; for (i=0;...) {...}`

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 comma 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 comma expressions, where each comma expression has exactly two terms. This is what we did in the previous example. Now, comma 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 equal to 0, comma equal to 0. It will initialize both values at the same time; both variables at the same time.

(Refer Slide Time: 15:49)



### Comma Operator execution

- Commas are evaluated from **left to right**. That is, `scanf("%d",&m), sum=0, i=0` is executed as `(scanf("%d",&m), sum=0), i=0`
- The comma operator has the lowest precedence of all operators in C. So  
`a=a+5, sum = sum + a`  
is equivalent to  
`(a=a+5), (sum = sum + a)`  
No need for explicit parentheses.

```
1 int a = 1;  
  int sum = 5;  
  a=a+5, sum = sum + a;
```

So, commas 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 comma expression; if we have multiple commas, what you do is you associate them just like you did with plus and star; you have multiple comma expressions. And then group them two at a time. So, it becomes two comma expressions. And then evaluate them. Now, the comma expression has the lowest precedence of any operator in C. So, if you have an operation like a equal to a plus 5 comma sum equal to sum plus a, what will happen is you do this expression a equal to a plus 5; then do this expression sum equal to sum plus a. And then evaluate the comma expression. And therefore, when you have a comma 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
high	Parenthesis	()	Left to right
	Boolean Not, unary -	! -	Right to left
P R E C E D E N C E	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
low	Comma	,	Left to right

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


(Refer Slide Time: 17:10)

### Comma Operator vs Separator

- The symbol , is used both as an operator and as a separator.
- We met the separator version earlier in multiple declarations and/or initializations.  

```
int sum =0, i=0, j=0;
```
- Also used in functions e.g., scanf and printf for separating operands  

```
scanf( "%d%d", a, b ); printf( "%d %d", a, b );
```
- The use of comma here is as a separator—not as an operator. In function calls (e.g., scanf, printf), variable definitions and initializations.
- Always clear from context whether comma is being used as a separator or as an operator.



This is also a slightly different meaning of the comma 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 equal to 0, comma equal to zero, comma j equal to 0; this is not the comma expression; it is just a separator as in English. So, similarly, when you call a function, you have comma to separate out

the arguments. That does not mean that, the arguments are inside a comma expression. Here comma is just a separator as in English. And it is always clear from the context whether a comma 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.