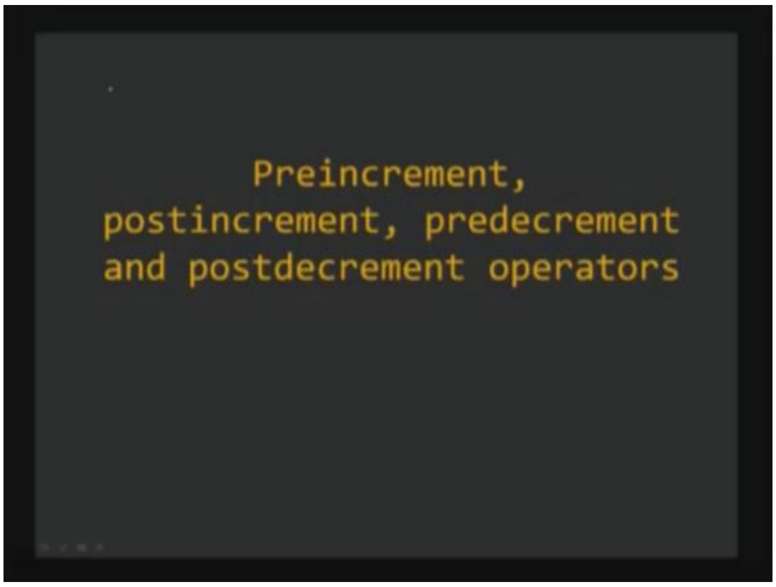
Introduction to Programming in C

Prof. Satyadev Nandakumar
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 55

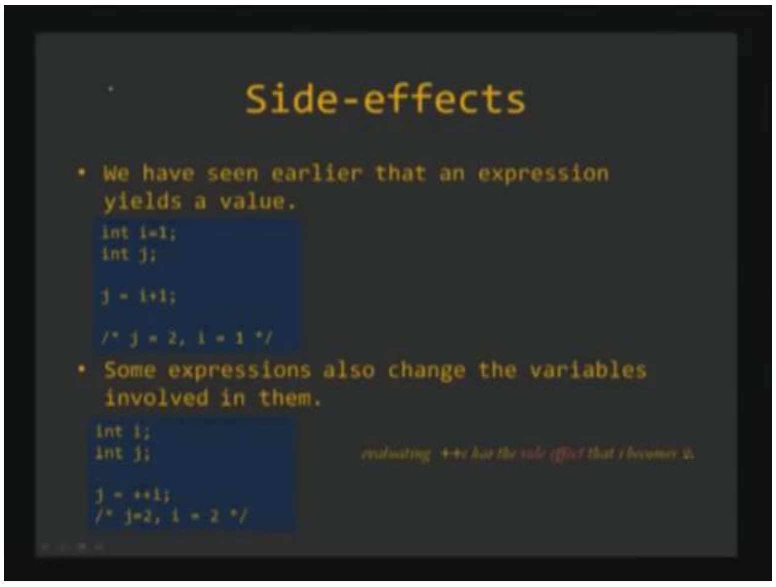
In this video will talk about how pre increment, post increment and operators like that work in c.

(Refer Slide Time: 00:00)



Preincrement, postincrement, predecrement and postdecrement operators

(Refer Slide Time: 00:09)



Side-effects

- We have seen earlier that an expression yields a value.

```
int i=1;
int j;

j = i+1;

/* j = 2, i = 1 */
```

- Some expressions also change the variables involved in them.

```
int i;
int j;

j = ++i;
/* j=2, i = 2 */
```

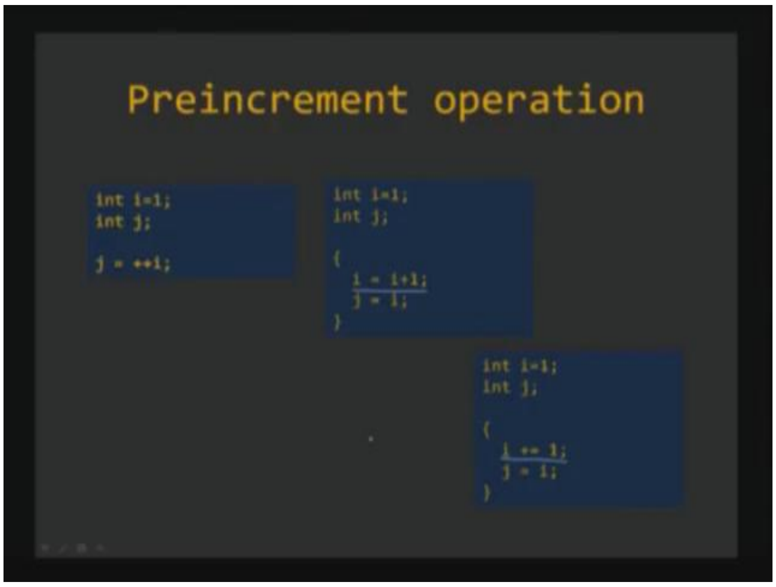
evaluating ++i has the side effect that i becomes 2.

So, will first introduce the notion of side effects. Now, we have earlier seen that any expression in c yields a certain values. So, let us look at a particular example, if you have integer variables i and j, i is assigned to 1 and then, you say that j is assigned to i plus 1. What happens is that, you take the value of i add 1 to it and then result of the expression i plus 1. So, the result of the expression will be 2, which is assigned to j. The value of i itself is un change due to an expression like i plus 1 is just that you read the value of a use it and then, return the value of i plus 1.

Now, some expressions in c also change the variables involved in them. For example, if you have a code like int i. Let us say i is initialized to 1 int j and then, you say j equal to plus plus i. In this case what happens is that, you take the value of i increment it. So, you will get i equal to 2 and then, that incremented value is then assign to j. So, evaluating the expression plus plus i has the side effect, that i becomes 2. So, it not only takes the value of i increments it by 1 and gives it to j, it also has the additional effect that i's value is incremented.

So, contrast the first example and the second example, in the first example when you said i plus 1, the value of i was unchanged and in the second when you said plus plus i, the value of i is changed. So, this is known as a side effect, because in addition to returning the value it also changes the variable involved in plus plus i.

(Refer Slide Time: 02:09)



Preincrement operation

```
int i=1;
int j;

j = ++i;
```

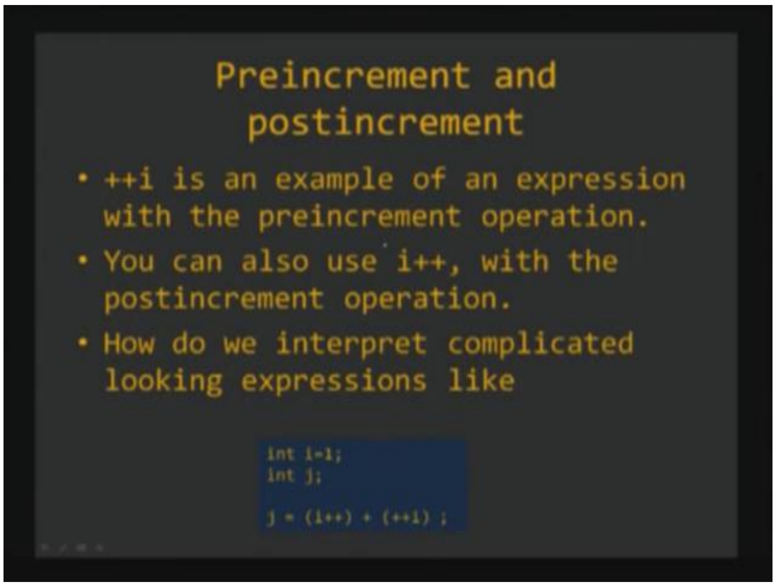
```
int i=1;
int j;

{
  i = i+1;
  j = i;
}
```

```
int i=1;
int j;

{
  i += 1;
  j = i;
}
```

So, now, let us look at the operation in slightly more detail. So, when you say int i equal to 1 in j and then j equal to plus plus i. The effect of this plus plus i can be understood in terms of an equivalent code. So, what you do is, consider a code where you have i equal to i plus 1 and then assign j equal to i. So, in this case i will become 2 after i equal to i plus 1 and then j will be assigned the value 2. So, this is the effect of the pre increment operations.

So, pre increment operation is called. So, because before you use the value of i, you would increment the value of i. So, that is one way of understanding this and these two codes are equivalent in effect. There is a slightly different way of writing this, which is a short form for writing i equal to i plus 1. So, instead of doing this, you can say i plus equal to 1. So, plus equal to 1 says the effect i equal to i plus 1. So, it is a short form of writing it. So, all these codes are have equivalent effect. So, plus plus i is called a pre increment operator, because before you use the value of i it is value is incremented.

(Refer Slide Time: 03:31)



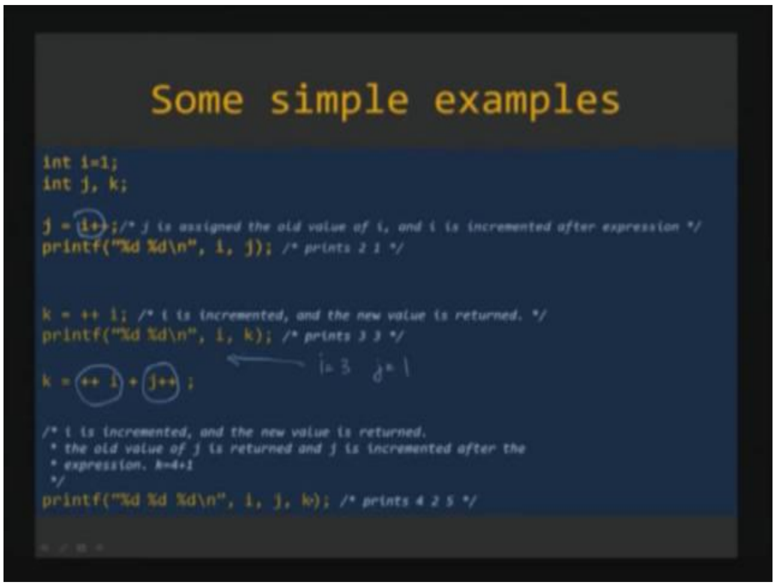
Preincrement and postincrement

- ++i is an example of an expression with the preincrement operation.
- You can also use i++, with the postincrement operation.
- How do we interpret complicated looking expressions like

```
int i=1;
int j;

j = (i++) + (++i) ;
```

Now, there is also the post increment operator. So, plus plus i is an example of an expression with the pre increment operation. And you can also use i plus plus, which is known as the post increment operation and confusing thing is how do we interrupt fairly complicated expressions like the following. So, suppose you have int i equal to 1 in j and then j equal to i plus plus plus plus plus i. So, what should we expect in this case is this allowed behavior what does it mean? What will be the result? Which is stored in j? So, let us look at these things in slightly greater detail.

(Refer Slide Time: 04:16)



Some simple examples

```
int i=1;
int j, k;

j = i++;/* j is assigned the old value of i, and i is incremented after expression */
printf("%d %d\n", i, j); /* prints 2 1 */


k = ++ i; /* i is incremented, and the new value is returned. */
printf("%d %d\n", i, k); /* prints 3 3 */

k = ++ i + j++ ;

/* i is incremented, and the new value is returned.
 * the old value of j is returned and j is incremented after the
 * expression. k=4+1
 */
printf("%d %d %d\n", i, j, k); /* prints 4 2 5 */
```

So, first let us look at some simple examples and try to understand the behavior. So, suppose you have i equal to 1 and then two variables j and k and first you say int j equal to i plus plus. So, this is the post increment operator. So, what happens here is that, you take the value of i assign into j. So, that is i equal to 1, the current value of i will be assign to j and after the expression is over i will be incremented. So, then i will become 2. The old value of i is assigned to j and then the value of i will be incremented. So, it is the post increment operator.

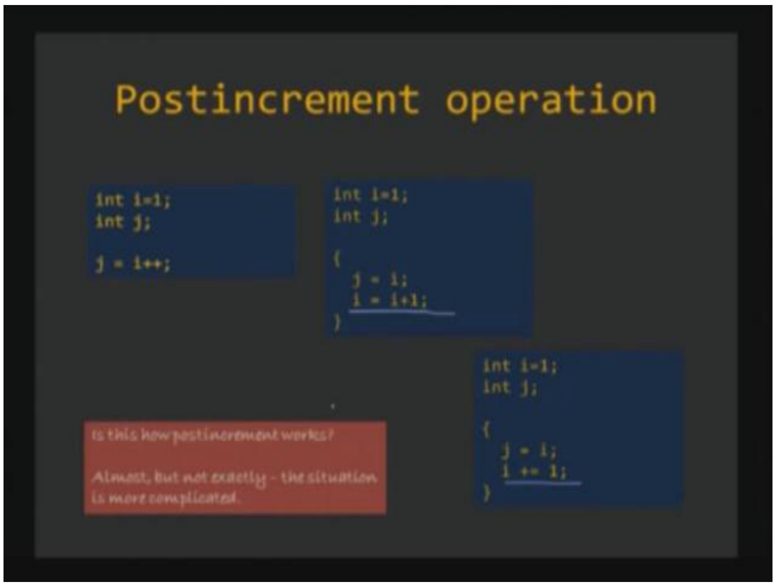
So, when you printf i and j here, i will be 2 and j will be 1. Because, old value of i was what was told in j. Now, let us look at plus plus i. So, if you say k equal to plus plus i, i is now 2 when it is starts and you pre increment i. So, you increment i, i becomes 3 and that value is stored in k. So, it is the pre increment operator,. So, the value will be incremented before the assignment will take place.

So, when you print i and k, i will be 3 and k will also be 3. So, notice the difference between the first case and the second case, the pre increment versus the post increment. Now, let us look at slightly more complicated examples. So, at this point what do we have? We have i equal to 3 and j equal to 1 at this point and then you say k equal to plus plus i plus j plus plus.

So, take a minute and think about what will happen here, you pre increment i. So, the value of this expression, that is used to add will be 4. Because, the value of i will be incremented before it is used in the plus expression, where as this is the post increment expression. So, the old value of j will be used and then j will be incremented.

So, here the value that will be used will be 4 and here the value that will be used will be the old value of j which is 1. So, k will be 4 plus 1 which is 5, i will be incremented. So, i becomes 4 and after this expression is over j will be incremented,. So, j becomes 2. So, when you print this you will say that i is 4, j is 2 and k is 5. So, understand why k is 5? Because, it is 4 plus 1 rather than 4 plus 2. So, this is fairly simple can be understood in terms of the pre increment and the post increment operator.

(Refer Slide Time: 07:25)



Postincrement operation

```
int i=1;
int j;

j = i++;
```

```
int i=1;
int j;

{
  j = i;
  i = i+1;
}
```

```
int i=1;
int j;

{
  j = i;
  i += 1;
}
```

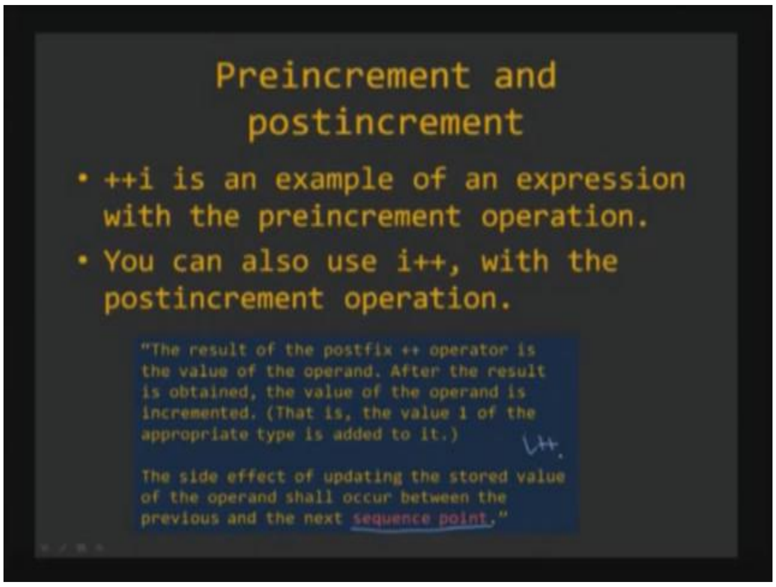
Is this how postincrement works?

Almost, but not exactly – the situation is more complicated.

So, let us look at some code that is equivalent to the post increment operation. So, suppose you have j equal to i plus plus, you can think of it like the following, you can say that j is assign to i. So, the old value of i is assign to j and then, the value of i is increment i equal to i plus 1. If you want to use the compound assignment operation, then what you can do is j equal to i and i equal to i plus 1. So, this is equivalent to i plus equal to 1 is equivalent to i equal to i plus 1.

So, contrast with the pre increment operation, there i equal to i plus 1 will be done before j equal to i, here j equal to i will be done before i equal to i plus 1. So, can we see that this is exactly how post increment works? And the answer is yes, in this particular case this is exactly how it works. But, if you want to understand the general case, we have to understand a slightly more advanced concept in c. And note that, this is not something that strictly false in to an introductory course. But, in case you want to understand exactly how it works, then we will look at the general case.

(Refer Slide Time: 08:42)



Preincrement and postincrement

- ++i is an example of an expression with the preincrement operation.
- You can also use i++, with the postincrement operation.

"The result of the postfix ++ operator is the value of the operand. After the result is obtained, the value of the operand is incremented. (That is, the value 1 of the appropriate type is added to it.)

i++.

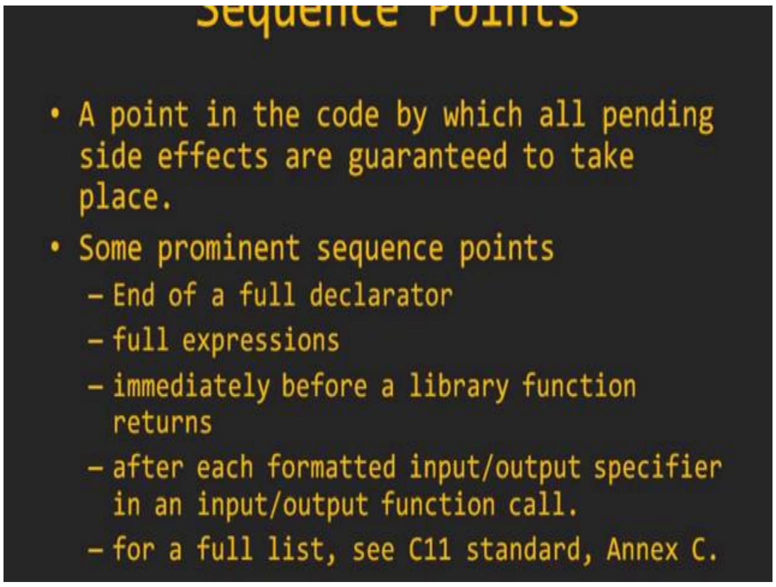
The side effect of updating the stored value of the operand shall occur between the previous and the next sequence point."

So, to understand the general case instead of writing a few examples and compiling it and saying, one way to do it would be to go to the c standard and say what is the standard say and here it is slightly surprising. So, the result of the postfixes operator is the value of the operands. So, this is the old value of the operand will be return, after the result is obtain, the value of the operand is incremented, this is what we saw in the last let.

Now, when is the operand increment, we loosely said last time that after the expression is over then the value of i will be incremented. But, what is the precise point at which the value of i will be incremented, this is slightly surprising. So, the side effect of updating the stored value of the operand shall occur between the previous and the next sequence point. So, when you have an i plus plus operation, it will not be immediately updated, it will be updated only after a place known as the sequence point.

So, let us just understand briefly what is mean by a sequence point. So, before we get into it let me emphasis, we are trying to understand. So, the post increment operation will say that, the old value of i will be used and the value of i will be incremented after the expression, we are time to precisely understand after what point can we say that i's value would have been incremented.

(Refer Slide Time: 10:23)



Sequence Points

- A point in the code by which all pending side effects are guaranteed to take place.
- Some prominent sequence points
  - End of a full declarator
  - full expressions
  - immediately before a library function returns
  - after each formatted input/output specifier in an input/output function call.
  - for a full list, see C11 standard, Annex C.

So, a sequence point has defined in the standard is a point in the code by which all pending side effects are ensured to be over. So, this is very technical definition and it is to be understood by compiler writers. But, we will briefly understand what does it mean? So, some prominent sequence points include end of full decelerators. So, for example, if I have a declaration int i equal to 0 comma j equal to 0, then a full decelerated gets over after i equal to 0. So, after i equal to 0 there is sequence point here.

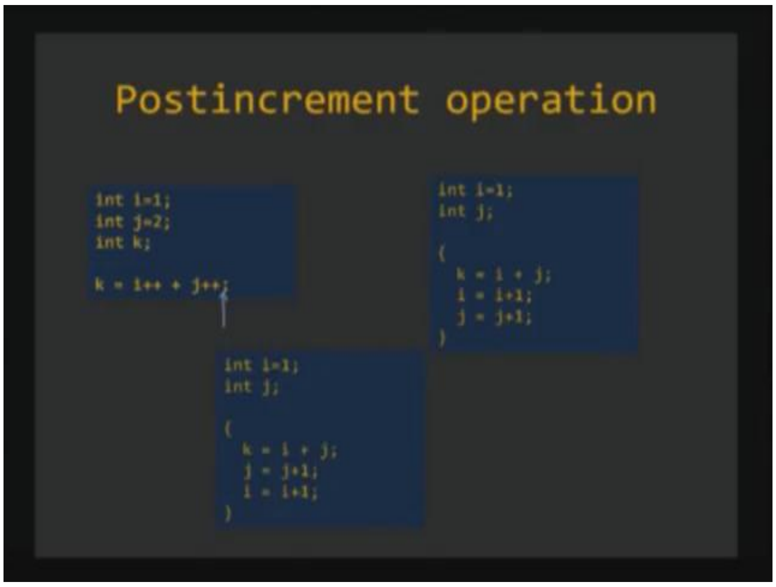
So, if there are any pending side effects, then it will be incremented at this point, this is another full decelerator. So, it will after that again any pending side effects will be ensured to be done. Then, the surprising think is suppose you have full expressions. So, suppose you have like i plus plus plus 3. So, the major think to understand will be when is this i plus plus suppose to happen, will it happen immediately after i plus plus and the think is then the c standard does not say that, that has to happen.

The c standard says that, the next sequence point is the semicolon. So, when you see the semicolon operation, you will know that, this hole think is what is known as a full expression j equal to i plus plus plus 3. So, that is known as a full expression. So, after you encounter a full expression any bending side effects. So, this is the pending side effect, that will be updated. So, only at that point c standard says that, by now you should have updated the i plus plus operation.

Before that the compiler is free to do what it works, it may or may not updated. So, this is actually slightly confusing and contrary to the popular understanding of when should i plus plus have to again the general cases slightly confusing, it is not what would expect, it just says that by the next sequence point in the code, all pending side effects should be take in place.

Now, it does in say that exactly at the end of the sequence point, you will update all side effects, compilers of free to do what it wants, all it says is that by the time you need the next sequence point pending side effect should take place in whatever all. So, this is slightly technical. Now, for a full stand list of course,, you have to refer to the c set standard, which is not really recommended I am in,, but it is just that if you want to understand that, then you can look at the standard.

(Refer Slide Time: 13:29)



Postincrement operation

```
int i=1;
int j=2;
int k;

k = i++ + j++;
```

```
int i=1;
int j;

{
  k = i + j;
  i = i+1;
  j = j+1;
}
```

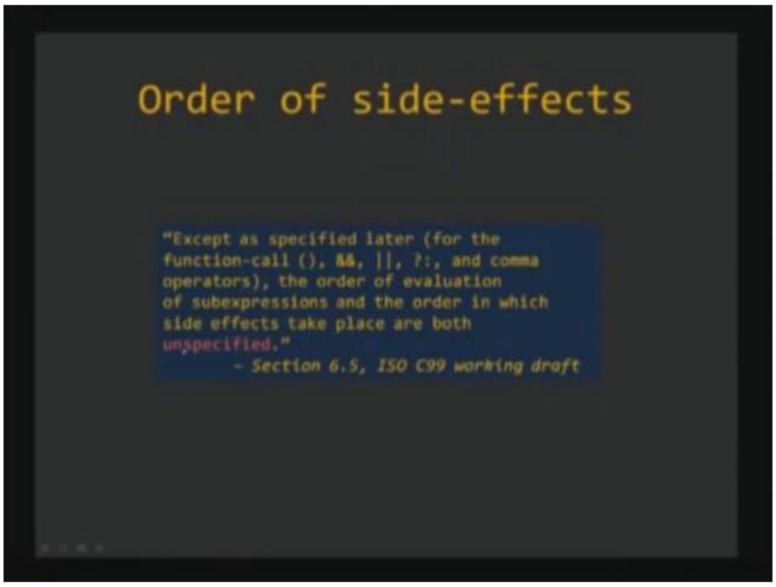
```
int i=1;
int j;

{
  k = i + j;
  j = j+1;
  i = i+1;
}
```

So, let us try to understand the post increment operation. So, int's again slightly greater detail. So, if you say that k equal to i plus plus plus j plus plus. Now, there are to ways to do it. Of course, k equal to i plus j, the old values of i and j are used and then, you calculate k assign in to k. And then, you can do i equal to i plus 1, j equal to j plus 1, because the standard says that by the time you see the full expression pending updates must be happen it.

So, you can say that by the time you see the semicolon operation I will do i equal to i plus 1 and j equal to j plus 1. Now, if you think a minute you could also do update j first and then i. So, I know that by the time is see the semicolon pending update should happen,, but in what order should it happen is it i equal to i plus 1 first and j equal to j plus 1 next or is it the other way round. And the answer is that, the c standard does in say.

So, it leaves it deliberately unspecified,. So, that the compiler can do what it wants.

(Refer Slide Time: 14:41)



Order of side-effects

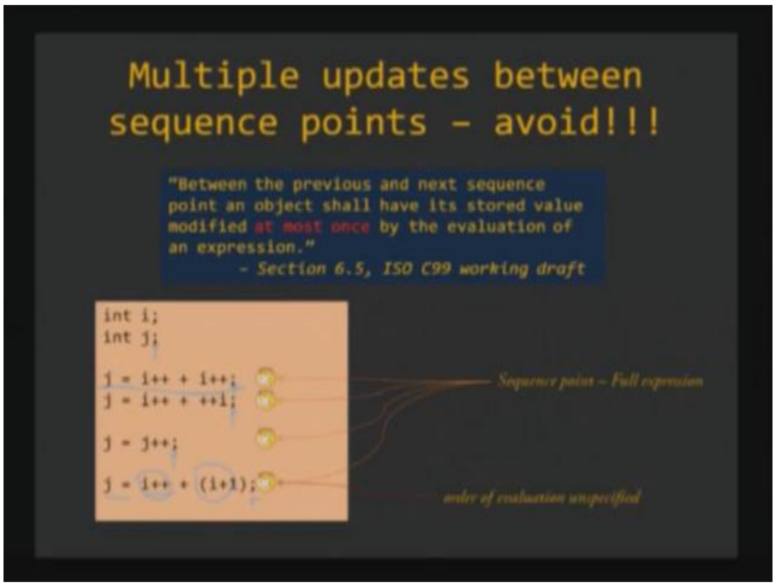
"Except as specified later (for the function-call (), &&, ||, ?:, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified."

– Section 6.5, ISO C99 working draft

So, here is the second certlity in this whole business. So, if you say, what is the order of the side effects? There are certain operations, where the sequence is specified. For example, the function call the logical AND operator, the logical OR operator, the conditional operator and the comma operator. So, for very specific operators the sequence is specified. But, in all other operations the order of evaluation of sub expressions is unspecified.

And similarly, the order of side effects is also unspecified. So, in the previous slide doing i equal to i plus 1 before j equal to j plus 1 is valid, as also j equal to j plus 1 and i equal to i plus 1. So, both these orders are valid and the c standard does not say that, what should really happen? So, what in practice you will notice is that, in one compiler a certain order may happen, in another compilers certain other order may happen. So, it is left to the compiler and you cannot assume anything about, what really happens? Which order it happens?

(Refer Slide Time: 15:50)



Multiple updates between sequence points – avoid!!!

"Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression."
– *Section 6.5, ISO C99 working draft*

```
int i;
int j;

j = i++ + i++;
j = i++ + ++i;

j = j++;

j = i++ + (i+1);
```

Sequence point – Full expression
order of evaluation unspecified

Further and here is the most important think as for as the sequence points are conserved. The c standard says that, an object or a variable can have it is stored value modified at most once by the evaluation of an expression between two sequence points, this is very important. So, between two sequence points, if a variable is to be updated by a side effect, then it should be updated at most once. Beyond that, if it is updated multiple times, the c standard says that the result is actually unspecified.

So, let us look at a few specific examples to see, what is actually happening here? So, let us take the first expression j equal to i plus plus plus i plus plus. So, we know that, here is a sequence point and we know that, here is a sequence point. These are full expressions between these two sequence points, the value of i is updated more than once. Here is i plus plus plus i plus plus and the c standard says that, the behavior is unspecified, this is somewhat surprising.

Because, you may try it out multiple times and you will see that consistently some behavior is happening. But, what the c standard says is that, if you take the code and compile it with a different compiler, the result may be different. So, the result of this expression is actually unspecified. Similarly, let us look at the next example. So, here the sequence point is a full expression, let us look at the next expression.

So, i plus plus plus plus plus i. So, post increment and then pre increment. Again even in this case, the result is unspecified, because these two are the sequence points here between this full expression and between these full expressions. So, you have two sequence points and between these two sequence points, the value of i is updated more than once. So, the result is unspecified according to the c standard.

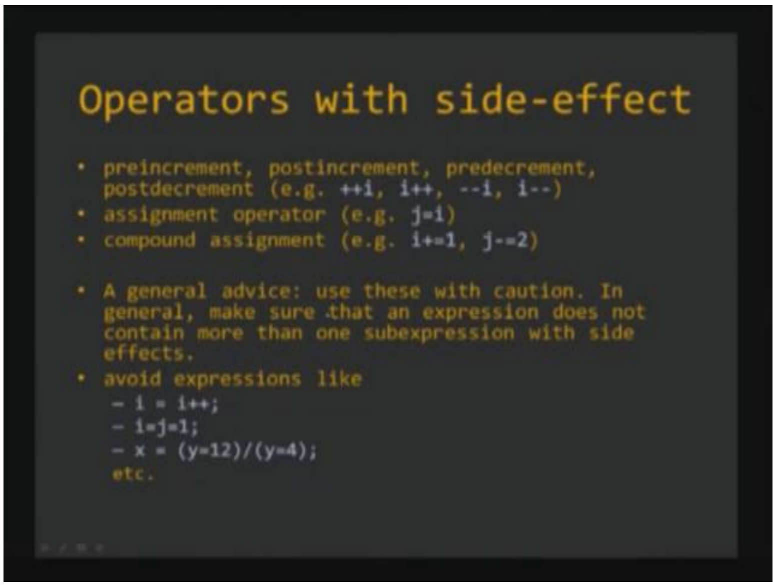
Let us look at this interesting example, j equal to j plus plus. Again result is unspecified, because you can have these two sequence points and between these, the value of j is updated twice. First, by the post increment operator and then by the assignment operator. So, the value of j is updated more than once, the result is unspecified. The last expression, it is interesting.

So, if you look at the two sequence points here, you have one full expression here, another full expression here. Between these, the value of i is updated only once, here and the value of j is updated only once, here. So, it is not that the value of i is updated more than once,. So, the value of j is updated more than once. But here, it say which of these sub expressions happened first? Is it i plus plus that happens first or i plus 1 that happens first.

So, according to the c standard that is actually unspecified. So, the order of evaluation of the sub expressions is unspecified, according to the c standard. So, let us just go back to that and this is, what it says, the order of evaluation of sub expressions is also unspecified. So, if you look at this expression j equal to i plus plus plus i plus 1, it is not clear which happens first, i plus plus or it is i plus 1, that is also unspecified.

So, here are the sequence points which end at full expression and the specific case of the last example, it is not that values of variables are updated more than once. It is just that the sub expressions may be evaluated in whatever order, it may occur.

(Refer Slide Time: 20:03)



Operators with side-effect

- preincrement, postincrement, predecrement, postdecrement (e.g. ++i, i++, --i, i--)
- assignment operator (e.g. j=i)
- compound assignment (e.g. i+=1, j-=2)
- A general advice: use these with caution. In general, make sure that an expression does not contain more than one subexpression with side effects.
- avoid expressions like
  - i = i++;
  - i=j=1;
  - x = (y=12)/(y=4);

  etc.

So, all this is slightly confusing. So, let us just summarize something that you can take away for, as per as programming is concerned. So, let us list out a few operators with side effects. So, let us say pre increment, post increment, pre decrement, post decrement all of these have side effects. In addition to returning the value, it also updates the variable. The assignment operator clearly has side effects.

So, if you say j equal to i;; obviously, the value of i will be assigned to j and we have earlier seen that, as an operation it returns the value that was assigned. So, that has the side effect, because it updates j and also writtens the value, which is the value of j. We have also seen this compound assignments. So, you can say i plus equal to 1, which is the same as i equal to i plus 1 and j minus equal to 2, which is as same as j equal to j minus 2. So, all these operators have side effects.

And the general advice is that, use operators with side effects with extreme caution. In general, if you use them make sure that a single full expression does not contain more than one sub expression with side effects. So, make sure that even if you want to use these expressions with side effects, make sure that one full expression contains at most one side effect.

So, avoid expressions like i equal to i plus plus, as we have seen before, this has two updates on i. So, the result is unspecified i equal to j equal to 1. Well, here is a full expression that has two side effects, technically the result is you can predict what the result is,, but as a programming practice, please avoid these kind of expressions.

Because, this is an expression that involves multiple updates and it is not really that the result is unspecified. Because, the updates are on different variables.

But still, as a good coding practice avoid such expressions. So, let us look at the third example, you have x equal to y equal to 12 divided by y equal to 4. Again, it is not clear, which of the sub expressions y equal to 12 or y equal to 4, which will happen first? So, the result of this expression is very difficult to interpret. So, in general do not use full expressions that have more than one side effects, even if they are on the single variable.

If it is multiple updates on a single variable, then the c standard clearly says that, the result is unspecified. But, even if they are on multiple variables, try to avoid writing such expressions. You can always write slightly longer code, where the meaning of the code will be perfectly clear and the result will be completely specified.

Thank you.