Introduction to Programming

Week – *4*, Lecture – *2*

Loops in C – `for`

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

What we know so far…

In the `do-while` loop, the condition is put at the end, after the body of the loop

- This means that the loop will run at least once, even if the condition is false at the start of the loop

What we know so far…

In the `do-while` loop, the condition is put at the end, after the body of the loop

- This means that the loop will run at least once, even if the condition is false at the start of the loop

In the `while` loop, the condition is put at the beginning, before the body of the loop

- This means that the loop may not even run once, if the condition is false at the start itself

What we know so far…

In the `do-while` loop, the condition is put at the end, after the body of the loop

- This means that the loop will run at least once, even if the condition is false at the start of the loop

In the `while` loop, the condition is put at the beginning, before the body of the loop

- This means that the loop may not even run once, if the condition is false at the start itself

In addition, there are two types of statements that are relevant for a loop's execution

- Statements that set initial values for variables that are important for the loop
- Statements within the loop, that update the values of these variables

What we know so far…

In the `do-while` loop, the condition is put at the end, after the body of the loop

- This means that the loop will run at least once, even if the condition is false at the start of the loop

In the `while` loop, the condition is put at the beginning, before the body of the loop

- This means that the loop may not even run once, if the condition is false at the start itself

In addition, there are two types of statements that are relevant for a loop's execution

- Statements that set initial values for variables that are important for the loop
- Statements within the loop, that update the values of these variables

For `while` and `do-while` loops, there is no designated slot for these statements

What we know so far…

In the `do-while` loop, the condition is put at the end, after the body of the loop

- This means that the loop will run at least once, even if the condition is false at the start of the loop

In the `while` loop, the condition is put at the beginning, before the body of the loop

- This means that the loop may not even run once, if the condition is false at the start itself

In addition, there are two types of statements that are relevant for a loop's execution

- Statements that set initial values for variables that are important for the loop
- Statements within the loop, that update the values of these variables

For `while` and `do-while` loops, there is no designated slot for these statements

The `for` loop in C (as well as C++ and Java), provide slots to provide these statements as well

- … in addition to the condition to stay in the loop

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

Initialisation statements are typical assignment statements of the form `x = y` (`y` could be a constant, a variable or an expression)

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

This is the condition to stay in the loop – similar to the condition in a `do-while` or `while` loop

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

Typically, we increment or decrement the variable(s) that we initialised in the initialisation slot (usually by a fixed amount, e.g. `1`)

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

After the statements before the loop get executed…

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

… the initialisation statements are executed

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

Then, the condition is checked; if it is false, the flow doesn't enter the loop… otherwise…

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

… the statements within the loop are executed

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

Then, the update statements are executed

The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

Post-update, the condition is rechecked; if it is true, the control enters the loop again… otherwise…

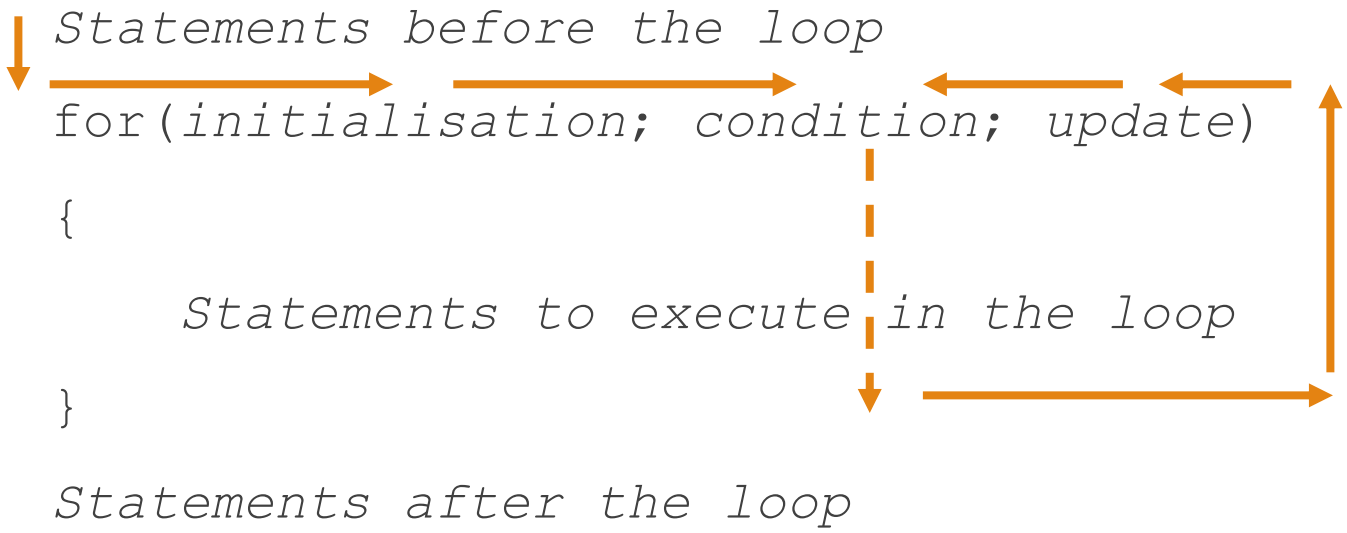
The `for` loop in C

```
Statements before the loop
for(initialisation; condition; update)
{
    Statements to execute in the loop
}
Statements after the loop
```

The flow moves through a `for` loop in a specific fashion

… the loop is terminated

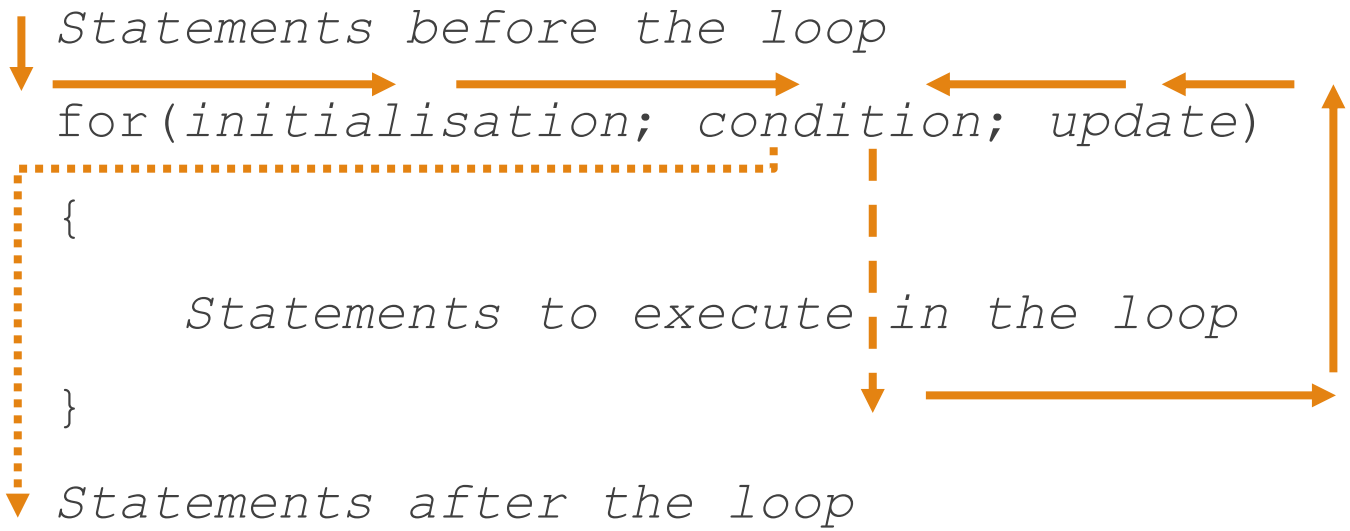
The `for` loop in C



The flow moves through a `for` loop in a specific fashion

When `condition` is true

The `for` loop in C



The flow moves through a `for` loop in a specific fashion

When `condition` is true

When `condition` is false

When to use `for` loop?

Similar to the `while` and `do-while` loop, the `for` loop too can be used for any scenario

When to use `for` loop?

Similar to the `while` and `do-while` loop, the `for` loop too can be used for any scenario

We usually employ the `for` loop when we know precisely, the number of times the loop should run

When to use `for` loop?

Similar to the `while` and `do-while` loop, the `for` loop too can be used for any scenario

We usually employ the `for` loop when we know precisely, the number of times the loop should run

A common example, is to browse through the elements of an array

- … i.e. access each element of the array in order (so the loop should run ”size of the array” times)

When to use `for` loop?

Similar to the `while` and `do-while` loop, the `for` loop too can be used for any scenario

We usually employ the `for` loop when we know precisely, the number of times the loop should run

A common example, is to browse through the elements of an array

- … i.e. access each element of the array in order (so the loop should run ”size of the array” times)

Remember that Marks Calculator task? The `for` loop is a good choice there

Revisiting the Marks Summer

```c
#include<stdio.h>

int main()
{
        int total_marks, total_maximum_marks;
        int marks[5];
        int max[5];

        printf("Please provide marks for five subjects\n");
        printf("Enter the marks in the format obtained/maximum\n");
        printf("Example:\n");
        printf("90/100\n");

        scanf("%d/%d", &marks[0], &max[0]);
        scanf("%d/%d", &marks[1], &max[1]);
        scanf("%d/%d", &marks[2], &max[2]);
        scanf("%d/%d", &marks[3], &max[3]);
        scanf("%d/%d", &marks[4], &max[4]);

        total_marks = marks[0] + marks[1] + marks[2] + marks[3] + marks[4];
        total_maximum_marks = max[0] + max[1] + max[2] + max[3] + max[4];

        printf("Total obtained marks: %d\n", total_marks);

        printf("Total maximum marks: %d\n", total_maximum_marks);

        return 0;
}
```

Revisiting the Marks Summer

```c
#include<stdio.h>

int main()
{
        int total_marks, total_maximum_marks;
        int marks[5];
        int max[5];

        printf("Please provide marks for five subjects\n");
        printf("Enter the marks in the format obtained/maximum\n");
        printf("Example:\n");
        printf("90/100\n");

        scanf("%d/%d", &marks[0], &max[0]);
        scanf("%d/%d", &marks[1], &max[1]);
        scanf("%d/%d", &marks[2], &max[2]);
        scanf("%d/%d", &marks[3], &max[3]);
        scanf("%d/%d", &marks[4], &max[4]);

        total_marks = marks[0] + marks[1] + marks[2] + marks[3] + marks[4];
        total_maximum_marks = max[0] + max[1] + max[2] + max[3] + max[4];

        printf("Total obtained marks: %d\n", total_marks);

        printf("Total maximum marks: %d\n", total_maximum_marks);

        return 0;
}
```

Let us get back to this part of the code again…

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

```
for(int i = 0; i < 5; i++)
        scanf("%d/%d", &marks[i], &max[i]);
```

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

```
for(int i = 0; i < 5; i++)
        scanf("%d/%d", &marks[i], &max[i]);
```

Modern C compilers allow “creating” a variable, who’s “scope” is limited to the `for` loop

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

```
for(int i = 0; i < 5; i++)
        scanf("%d/%d", &marks[i], &max[i]);
```

Modern C compilers allow “creating” a variable, who’s “scope” is limited to the `for` loop

The scope of a variable, is the code block in which it is “visible”

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

```
for(int i = 0; i < 5; i++)
        scanf("%d/%d", &marks[i], &max[i]);
```

Modern C compilers allow “creating” a variable, who’s “scope” is limited to the `for` loop

The scope of a variable, is the code block in which it is “visible”

Here, the variable `i` is only visible within the `for` loop, and not outside it

These are all equivalent…

```
scanf("%d/%d", &marks[0], &max[0]);
scanf("%d/%d", &marks[1], &max[1]);
scanf("%d/%d", &marks[2], &max[2]);
scanf("%d/%d", &marks[3], &max[3]);
scanf("%d/%d", &marks[4], &max[4]);
```

```
int i = 0;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]); i = i+1;
scanf("%d/%d", &marks[i], &max[i]);
```

```
for(int i = 0; i < 5; i++)
        scanf("%d/%d", &marks[i], &max[i]);
```

Modern C compilers allow “creating” a variable, who’s “scope” is limited to the `for` loop

The scope of a variable, is the code block in which it is “visible”

Here, the variable `i` is only visible within the `for` loop, and not outside it

Also, if the `for` loop has a single statement in the loop’s body, the braces, i.e. the `{` and `}`, are optional

Some points about the `for` loop

The initialisation part of a `for` loop is executed “exactly once”

- This is right after the statements before the loop were executed

Some points about the `for` loop

The initialisation part of a `for` loop is executed "exactly once"

- This is right after the statements before the loop were executed

All three parts of a `for` loop can be an *empty statement*

- An empty statement is "just a semicolon" – i.e. `;`
- This means, that it is perfectly legal to write this:

```
for(;;)
{
    // body of the loop
}
```

Some points about the `for` loop

The initialisation part of a `for` loop is executed "exactly once"

- This is right after the statements before the loop were executed

All three parts of a `for` loop can be an *empty statement*

- An empty statement is "just a semicolon" – i.e. `;`
- This means, that it is perfectly legal to write this:

```
for(;;)
{
    // body of the loop
}
```

If the condition is not provided, by default it is *evaluated to true*

- So the above loop is basically an infinite loop, unless you find a mechanism to "break" out of it
- This mechanism is often used in programming to provide "loop exit conditions" in the "middle of the loop"
- We will see this mechanism in the next lecture

Another example – the same old Factorial

```c
#include<stdio.h>

int main()
{
        int num = -1, i;
        long result = 1;

        for(;num < 0;)
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
        }

        for(i = 2; i <= num; i++)
                result *= i;

        printf("Calculated Factorial: %ld\n", result);

        return 0;
}
```

Another example – the same old Factorial

```c
#include<stdio.h>

int main()
{
        int num = -1, i;
        long result = 1;

        for(;num < 0;)
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
        }

        for(i = 2; i <= num; i++)
                result *= i;

        printf("Calculated Factorial: %ld\n", result);

        return 0;
}
```

We are providing only the condition here, and not using the two other slots in the `for` loop

Another example – the same old Factorial

```c
#include<stdio.h>

int main()
{
        int num = -1, i;
        long result = 1;

        for(;num < 0;)
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
        }

        for(i = 2; i <= num; i++)
                result *= i;

        printf("Calculated Factorial: %ld\n", result);

        return 0;
}
```

Here, we start the loop with an initial value of `2` for `i`, taking care of the factorial of `0` problem !!

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

But there are “exceptions” – we will see one now

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

But there are “exceptions” – we will see one now

The `++` operator is a very common *unary* operator used with the `for` loop

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

But there are "exceptions" – we will see one now

The `++` operator is a very common *unary* operator used with the `for` loop

It has two avatars – the "*prefix* `++`" (e.g. `++i`) and the "*postfix* `++`" (e.g. `i++`)

- The *postfix* `++` can be interpreted as "use the value of the variable, then increment its value by 1"
- The *prefix* `++` can be interpreted as "increment the value of the variable by 1, then use it"

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

But there are "exceptions" – we will see one now

The `++` operator is a very common *unary* operator used with the `for` loop

It has two avatars – the "*prefix* `++`" (e.g. `++i`) and the "*postfix* `++`" (e.g. `i++`)

- The *postfix* `++` can be interpreted as "use the value of the variable, then increment its value by 1"
- The *prefix* `++` can be interpreted as "increment the value of the variable by 1, then use it"

They both have different precedence and *associativity*

- *postfix* `++` has a higher precedence than *prefix* `++`
- The associativity of *postfix* `++` is left to right (what we know so far), and that of *prefix* `++` is right to left

Detour – The ++ operator

Remember that for most cases, we assumed that the expressions are evaluated from left to right

But there are "exceptions" – we will see one now

The `++` operator is a very common *unary* operator used with the `for` loop

It has two avatars – the "*prefix* `++`" (e.g. `++i`) and the "*postfix* `++`" (e.g. `i++`)

- The *postfix* `++` can be interpreted as "use the value of the variable, then increment its value by 1"
- The *prefix* `++` can be interpreted as "increment the value of the variable by 1, then use it"

They both have different precedence and *associativity*

- *postfix* `++` has a higher precedence than *prefix* `++`
- The associativity of *postfix* `++` is left to right (what we know so far), and that of *prefix* `++` is right to left

However, associativity is an issue, only when you have two operators with same precedence

- Let us leave the associativity doubts for later weeks; for now, just concentrate on their usage

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

What we mean is, as long as you use it in the following two forms, both avatars are identical in effect:

```
i++;
++i;
```

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

What we mean is, as long as you use it in the following two forms, both avatars are identical in effect:

```
i++;
++i;
```

- Basically, as long as you do not use them in a complex expression, they both have the same effect

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

What we mean is, as long as you use it in the following two forms, both avatars are identical in effect:

```
i++;
++i;
```

- Basically, as long as you do not use them in a complex expression, they both have the same effect

If you do use it in an expression, this what it means:

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

What we mean is, as long as you use it in the following two forms, both avatars are identical in effect:

```
i++;
++i;
```

- Basically, as long as you do not use them in a complex expression, they both have the same effect

If you do use it in an expression, this what it means:

- The *prefix* ++:

```
j = k * ++i - l;
```
is equivalent to
```
i = i + 1;
j = k * i - l;
```

Detour – The ++ operator

For now, start using the ++ operator ”in isolation”

What we mean is, as long as you use it in the following two forms, both avatars are identical in effect:

```
i++;
++i;
```

- Basically, as long as you do not use them in a complex expression, they both have the same effect

If you do use it in an expression, this what it means:

- The *prefix* ++:
  `j = k * ++i - l;` is equivalent to `i = i + 1; j = k * i - l;`
- The *postfix* ++:
  `j = k * i++ - l;` is equivalent to `j = k * i - l; i = i + 1;`

Homework !!

Find out which other loop(s) provide you the liberty of not using braces…

- … when the loop's body consists of just one statement

Are there any other places where this – single statement code block without braces – is applicable?

- Find out !!

Go through the Operator Precedence Table again

- Find out where the *postfix* and *prefix* ++ operators lie on this table
- Using this table, convince yourself that we did not need to include them in parentheses in the example on the previous slide !!