Introduction to Programming

Week – *4*, Lecture – *3*

Conditionals in C – Part 2

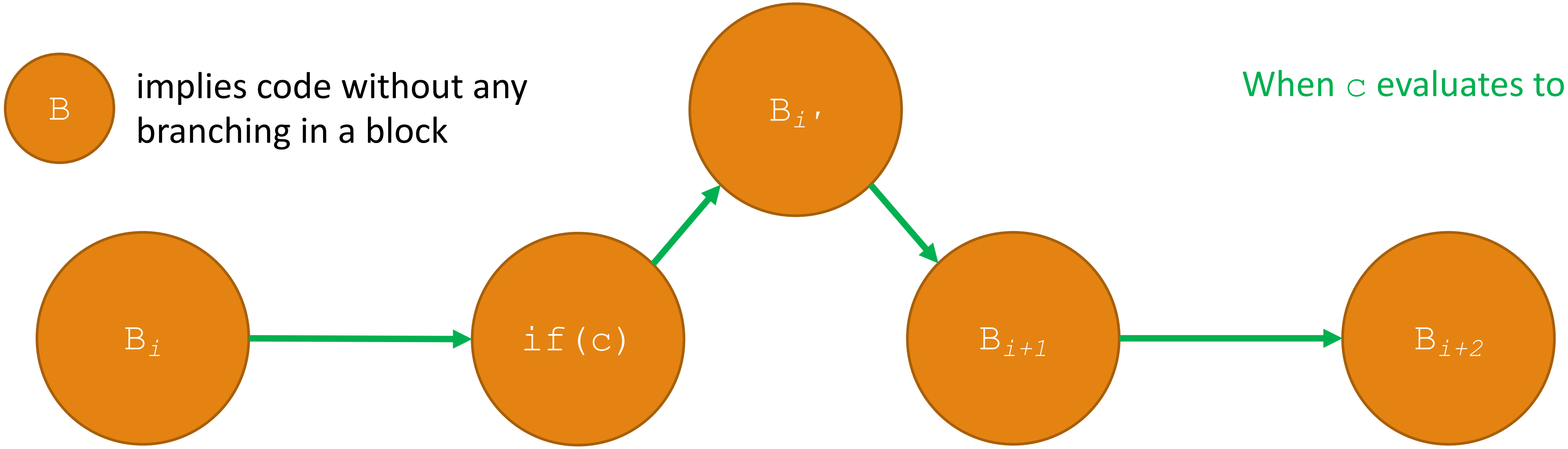
SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Recap...

A simple `if` statement allows a detour in execution of the code when a condition is *true*

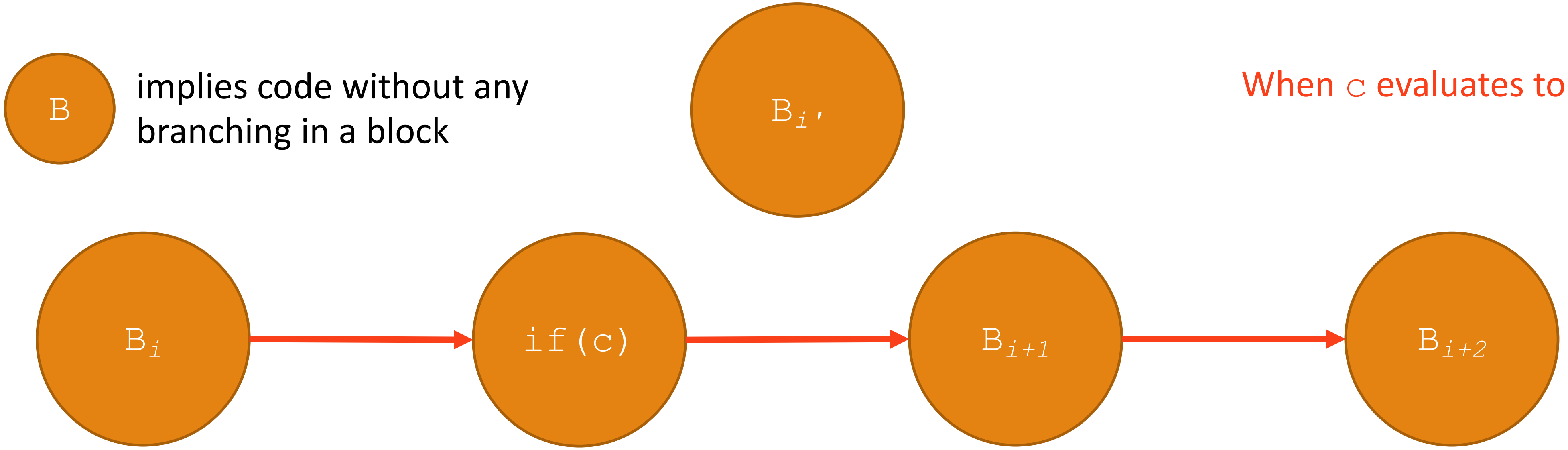


B implies code without any branching in a block

When `c` evaluates to true

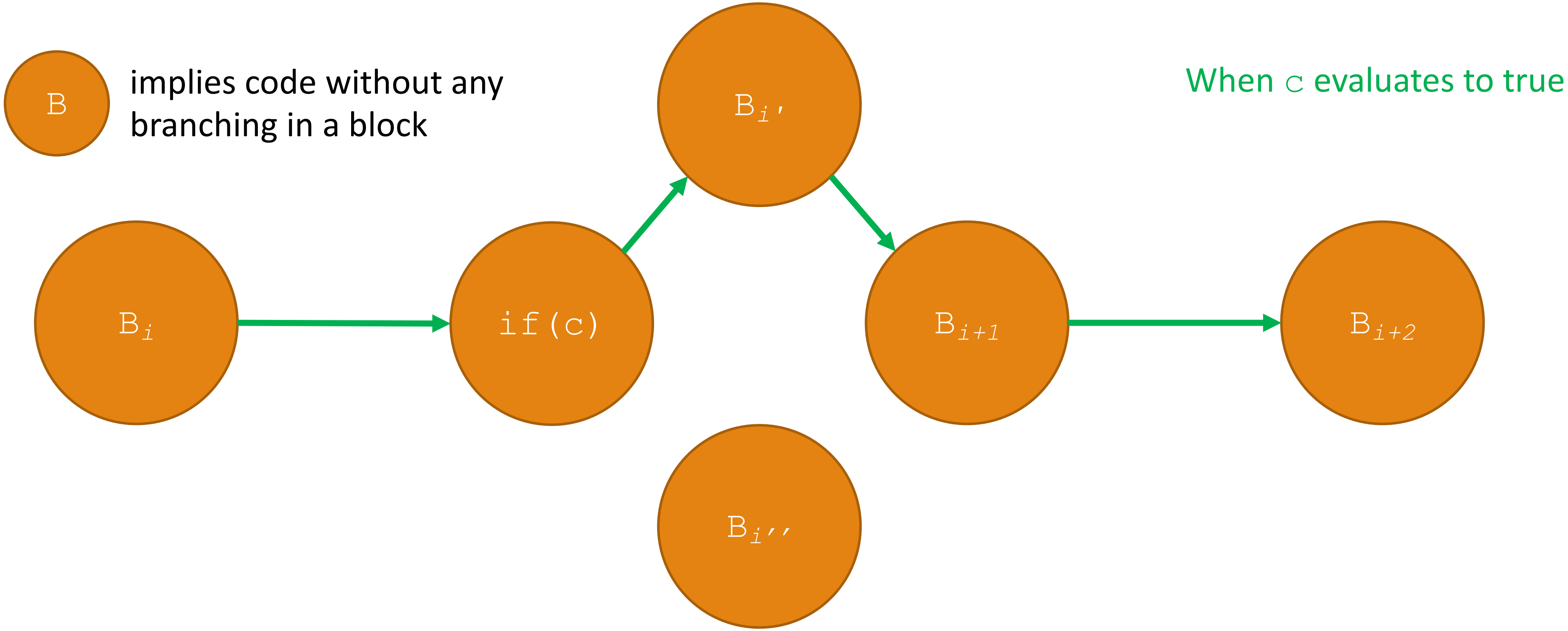
Recap…

A simple `if` statement allows a detour in execution of the code when a condition is *true*


B
implies code without any branching in a block
$B_{i'}$
When c evaluates to false
B_i
if(c)
B_{i+1}
B_{i+2}

Recap...

An `if` statement can be accompanied by an `else` statement to allow another detour



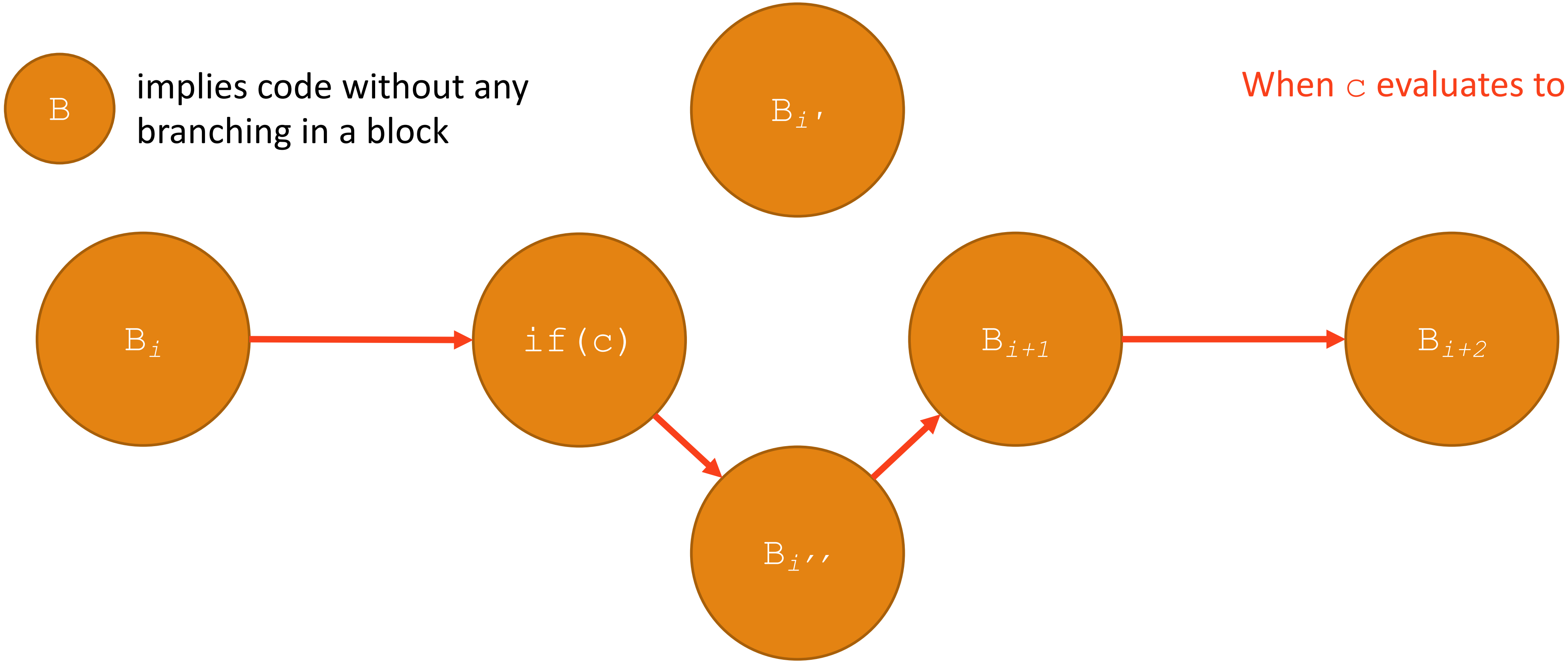
implies code without any branching in a block

When `c` evaluates to true

Recap...

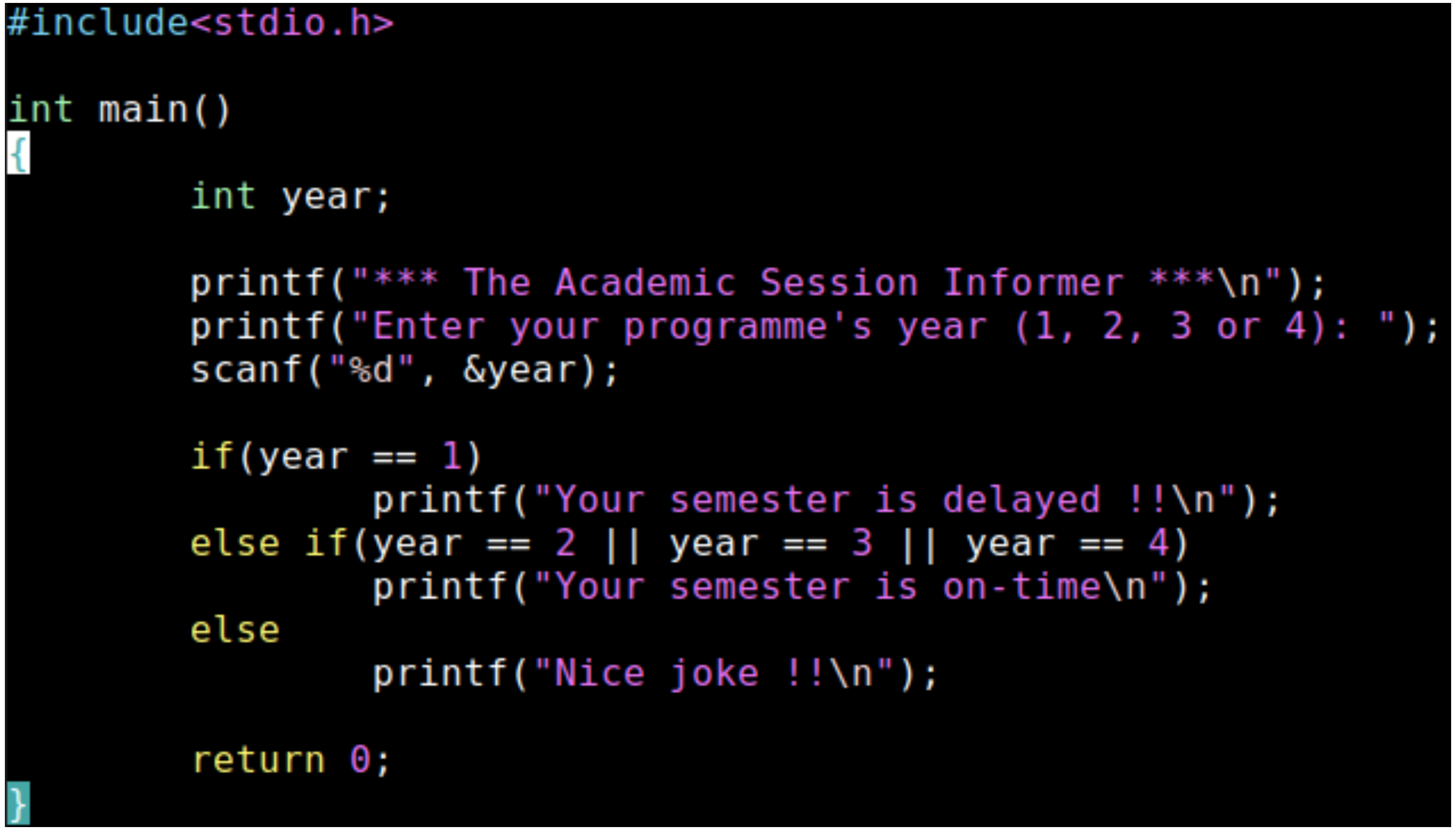
An `if` statement can be accompanied by an `else` statement to allow another detour

B implies code without any branching in a block



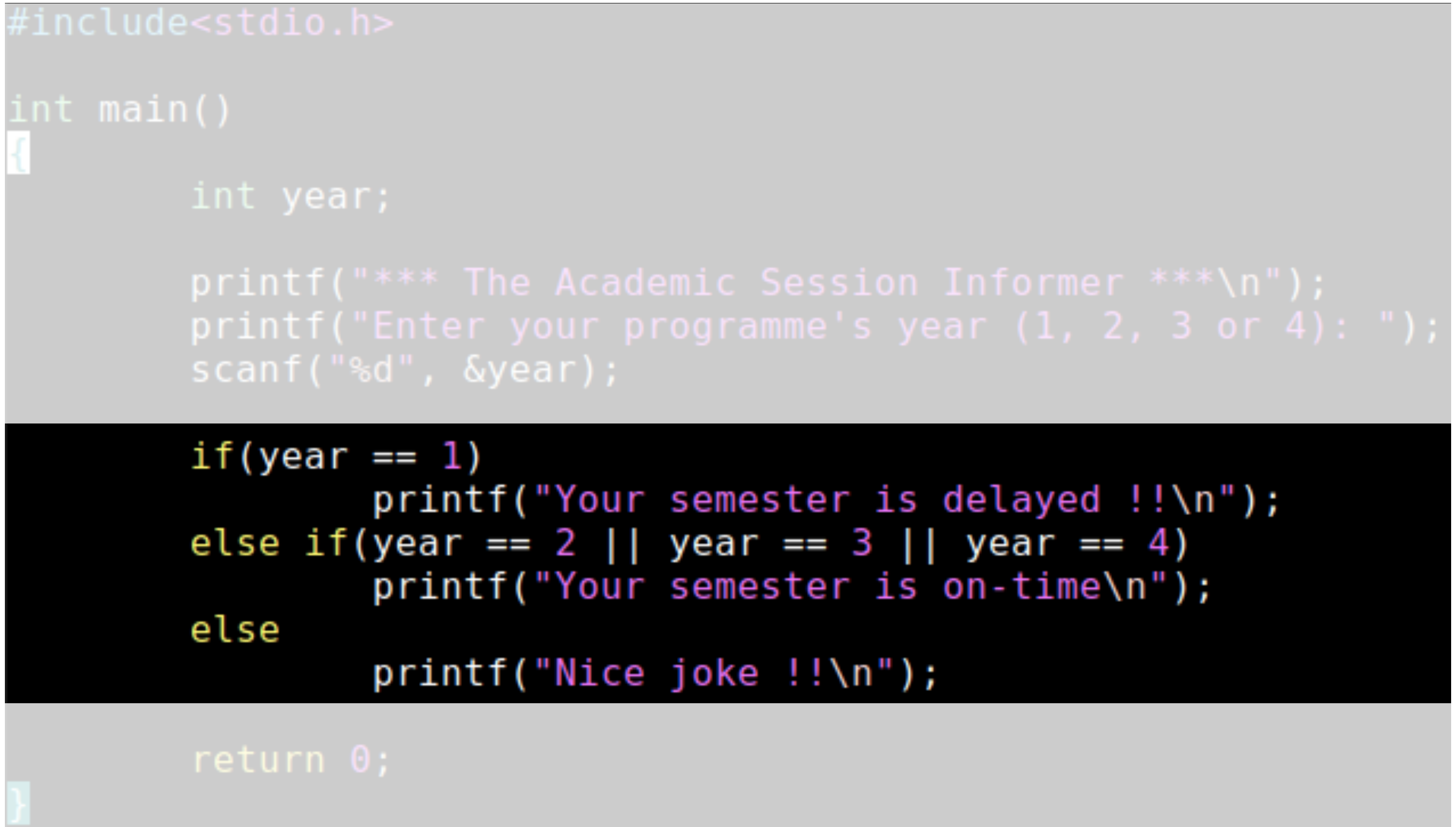
When `c` evaluates to false

A typical `if-else` ladder



```c
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        if(year == 1)
                printf("Your semester is delayed !!\n");
        else if(year == 2 || year == 3 || year == 4)
                printf("Your semester is on-time\n");
        else
                printf("Nice joke !!\n");

        return 0;
}
```

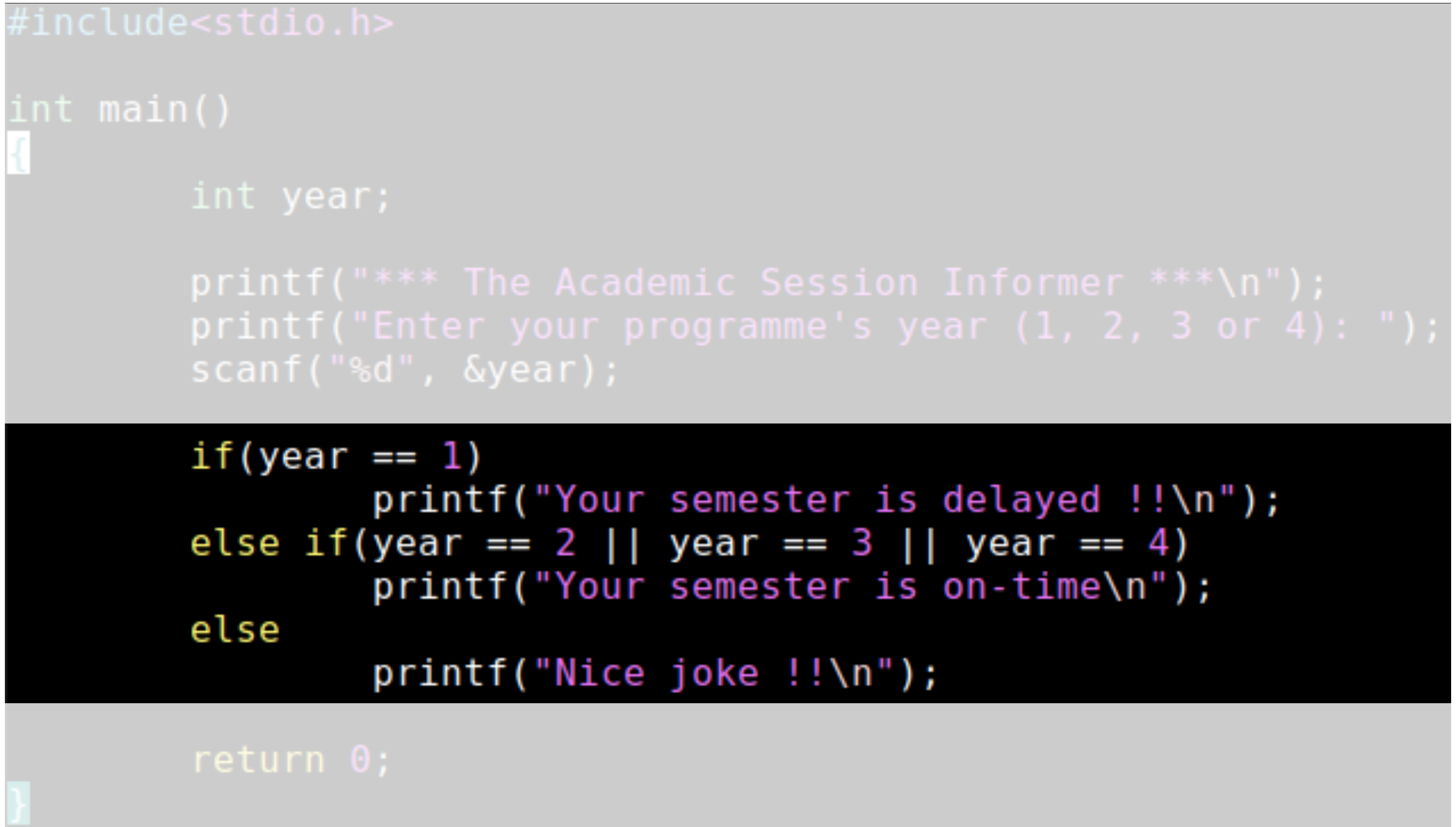
A typical `if-else` ladder



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        if(year == 1)
                printf("Your semester is delayed !!\n");
        else if(year == 2 || year == 3 || year == 4)
                printf("Your semester is on-time\n");
        else
                printf("Nice joke !!\n");

        return 0;
}
```

This is a special type of `if-else` ladder though…

A typical `if-else` ladder



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        if(year == 1)
                printf("Your semester is delayed !!\n");
        else if(year == 2 || year == 3 || year == 4)
                printf("Your semester is on-time\n");
        else
                printf("Nice joke !!\n");

        return 0;
}
```

This is a special type of `if-else` ladder though…

The conditions here are based on the value of a single *integer* variable – `year` !!

The `switch-case` construct

For cases where the `if-else` ladders are based on different values of a single variable…

- … you can also use another construct – the `switch-case` construct or just a switch in short

The switch-case construct

The syntax is:

```
switch(v)
{
     case a:
          statements to execute, if v == a is true
     case b:
          statements to execute, if v == b is true
     case c:
          statements to execute, if v == c is true
     case d:
          statements to execute, if v == d is true
     …
     default:
          statements to execute, when no other case matches
}
```

The `switch-case` construct

The syntax is:

We begin by “switching” a variable as shown

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

The `switch-case` construct

The syntax is:

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

We begin by “switching” a variable as shown

In the Academic Informer example, we can switch the `year` variable

The `switch-case` construct

The syntax is:

We then define ”cases” for each value of the variable, *that is relevant* for our us

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

The switch-case construct

The syntax is:

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

We then define ”cases” for each value of the variable, *that is relevant* for our us

In the Academic Informer example, we have four values that are relevant – `1`, `2`, `3` and `4`

The switch-case construct

The syntax is:

For each case, we can provide one or statements, that should be executed – it is equivalent to a code block, but you don’t need braces around it

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

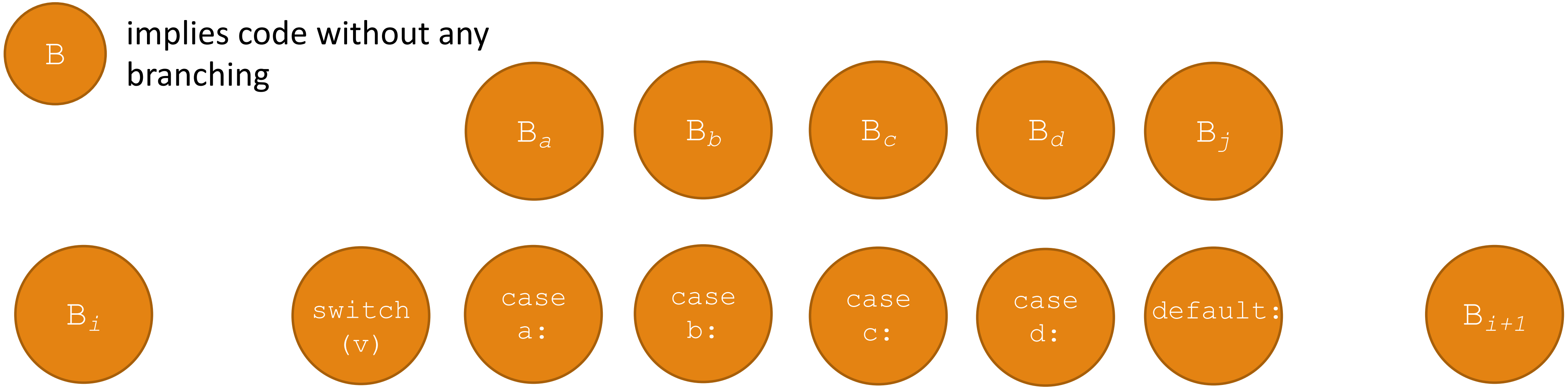
The `switch-case` construct

The syntax is:

```
switch(v)
{
        case a:
                statements to execute, if v == a is true
        case b:
                statements to execute, if v == b is true
        case c:
                statements to execute, if v == c is true
        case d:
                statements to execute, if v == d is true
        …
        default:
                statements to execute, when no other case matches
}
```

We can also define a default section, containing statements, which will be executed if none of the cases match the variable’s current value

The `switch-case` construct



B implies code without any branching

B_a B_b B_c B_d B_j

B_i

switch (v)

case a:

case b:

case c:

case d:

default:

B_{i+1}

The `switch-case` construct

For cases where the `if-else` ladders are based on different values of a single variable…

- … you can also use another construct – the `switch-case` construct or just a switch in short

Not only in C, the `switch-case` construct can be found in many programming languages

- The types of variables that can be used with `switch-case`, may be different for different languages
- In C, integers (or any data type that can be implicitly converted to integers) can be used with `switch-case`

The `switch-case` construct

For cases where the `if-else` ladders are based on different values of a single variable…

- … you can also use another construct – the `switch-case` construct or just a switch in short

Not only in C, the `switch-case` construct can be found in many programming languages

- The types of variables that can be used with `switch-case`, may be different for different languages
- In C, integers (or any data type that can be implicitly converted to integers) can be used with `switch-case`

There is a major issue with the `switch-case` construct though !!

- While the "cases" may seem similar to the different "else if" clauses, they are actually not…
- The order in which you define the cases, matter

The `switch-case` construct

For cases where the `if-else` ladders are based on different values of a single variable…

- … you can also use another construct – the `switch-case` construct or just a switch in short

Not only in C, the `switch-case` construct can be found in many programming languages

- The types of variables that can be used with `switch-case`, may be different for different languages
- In C, integers (or any data type that can be implicitly converted to integers) can be used with `switch-case`

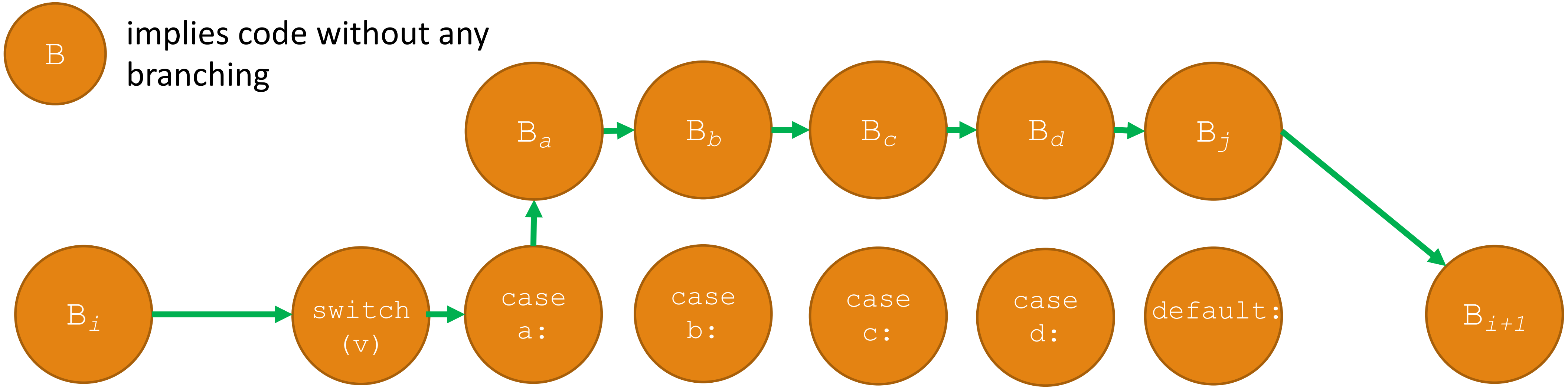
There is a major issue with the `switch-case` construct though !!

- While the "cases" may seem similar to the different "else if" clauses, they are actually not…
- The order in which you define the cases, matter

Once a particular case matches… the statements defined for that case are executed…

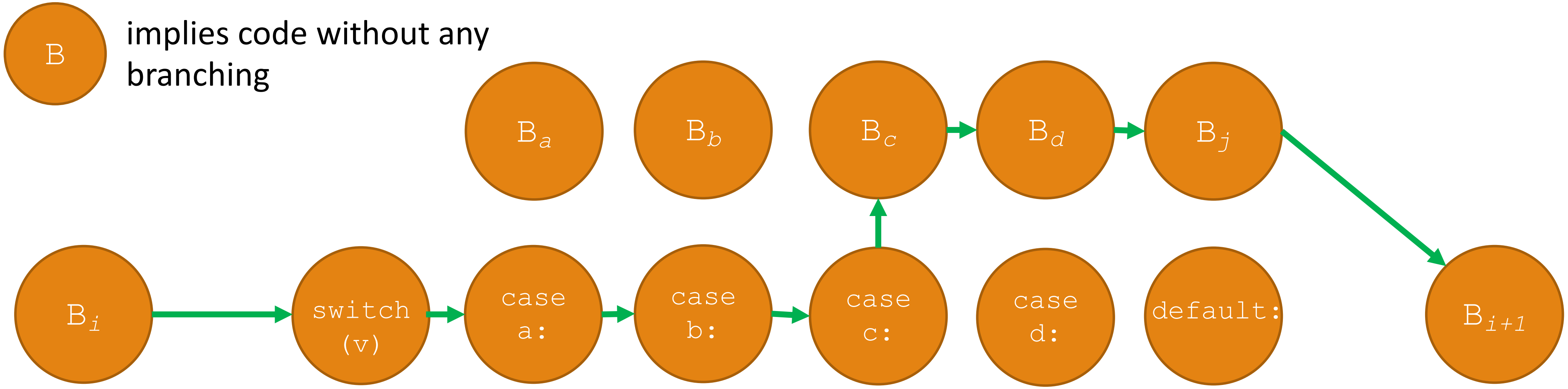
- … as well as, statements defined for all the cases below this case are also executed

The `switch-case` construct



When `v` = a

The `switch-case` construct



When `v` = c

The `switch-case` construct

For cases where the `if-else` ladders are based on different values of a single variable…

- … you can also use another construct – the `switch-case` construct or just a switch in short

Not only in C, the `switch-case` construct can be found in many programming languages

- The types of variables that can be used with `switch-case`, may be different for different languages
- In C, integers (or any data type that can be implicitly converted to integers) can be used with `switch-case`

There is a major issue with the `switch-case` construct though !!

- While the "cases" may seem similar to the different "else if" clauses, they are actually not…
- The order in which you define the cases, matter

Once a particular case matches… the statements defined for that case are executed…

- … as well as, statements defined for all the cases below this case are also executed

This is why, usually, each block of statements end with a special statement – the `break` statement

The switch-case construct

The syntax is:

```
switch(v)
{
        case a:
                statements to execute, if v == a is true; break;
        case b:
                statements to execute, if v == b is true; break;
        case c:
                statements to execute, if v == c is true; break;
        case d:
                statements to execute, if v == d is true; break;
        …
        default:
                statements to execute, when no other case matches
}
```

The switch-case construct

The syntax is:

Something like this…

```
switch(v)
{
        case a:
                statements to execute, if v == a is true; break;
        case b:
                statements to execute, if v == b is true; break;
        case c:
                statements to execute, if v == c is true; break;
        case d:
                statements to execute, if v == d is true; break;
        …
        default:
                statements to execute, when no other case matches
}
```

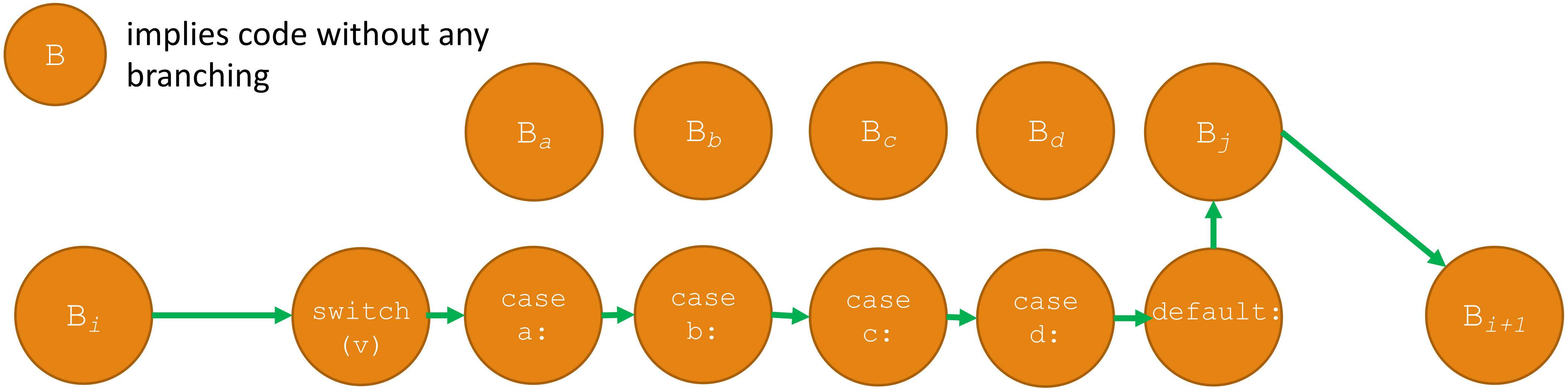
The `switch-case` construct

The syntax is:

You don’t need a `break` statement for the last code block in the construct, because the control will anyhow exit the switch

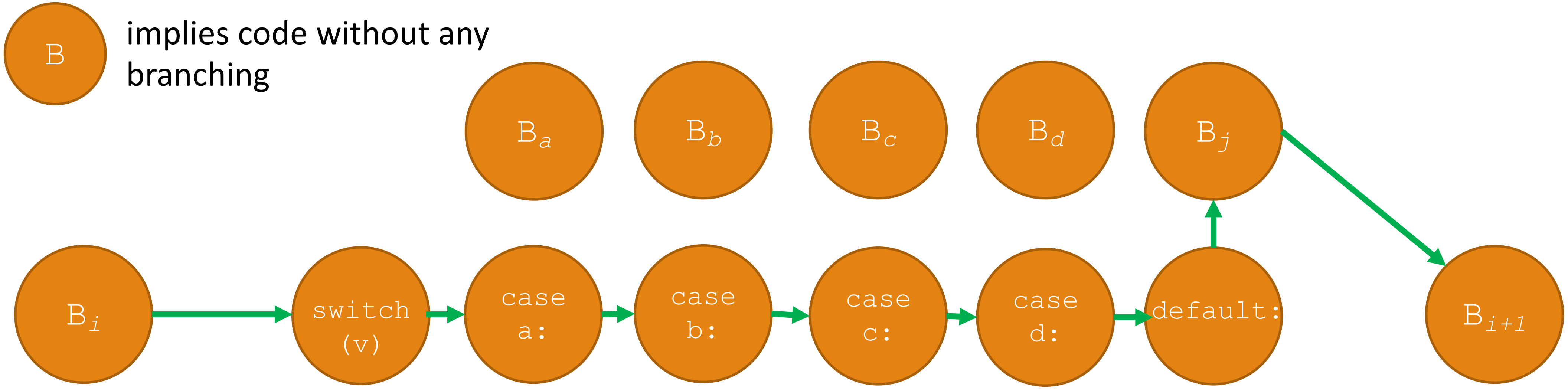
```
switch(v)
{
        case a:
                statements to execute, if v == a is true; break;
        case b:
                statements to execute, if v == b is true; break;
        case c:
                statements to execute, if v == c is true; break;
        case d:
                statements to execute, if v == d is true; break;
        …
        default:
                statements to execute, when no other case matches
}
```

The `switch-case` construct



When the value of `v` doesn't match any case

The `switch-case` construct



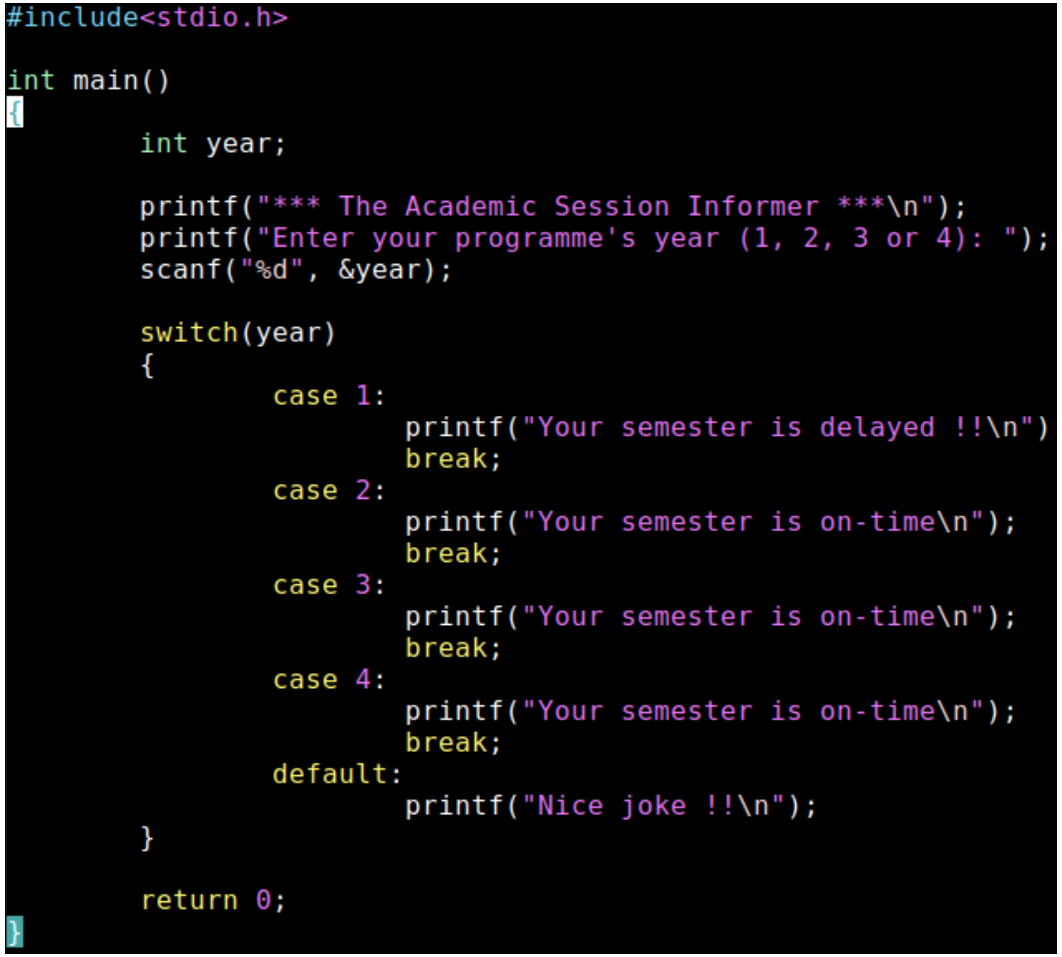
We usually define default at the end, but it is not necessary (if you define it anywhere else, you'll need a `break` statement there too !!)

When the value of `v` doesn't match any case

The `break` statement

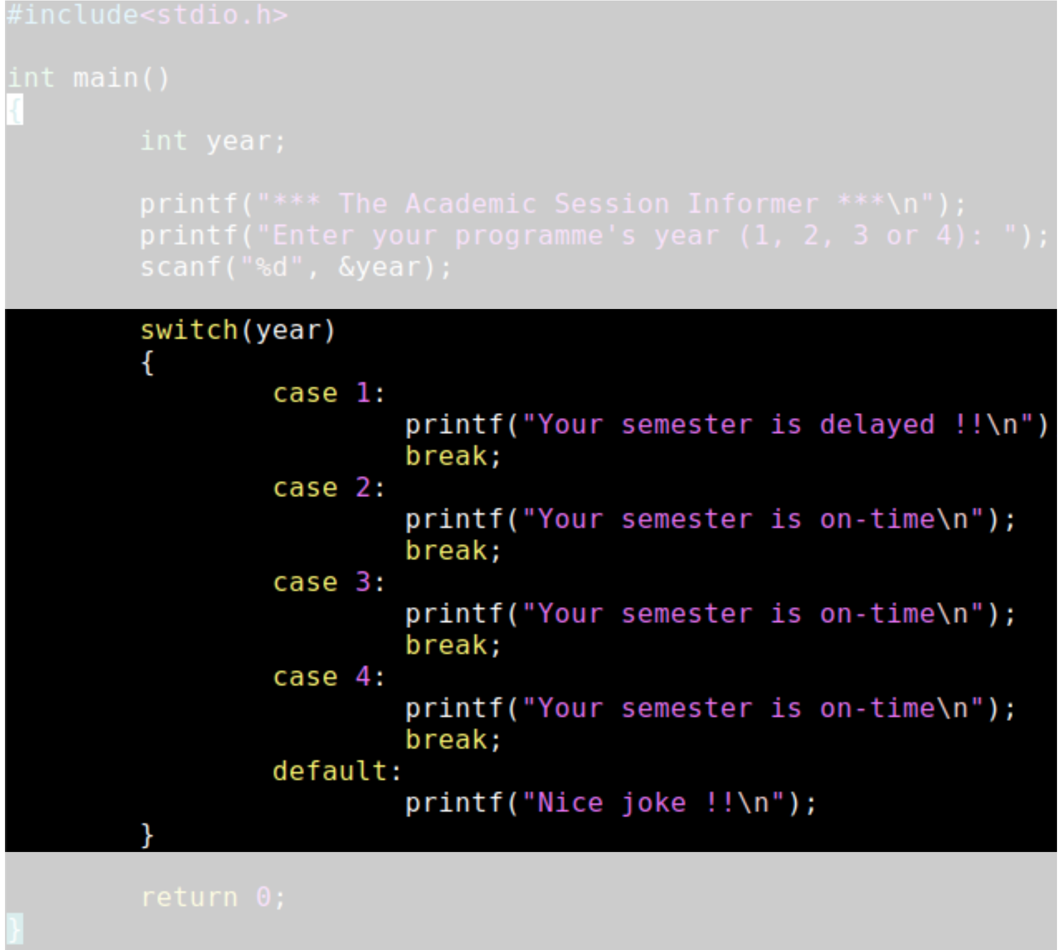
The `break` statement forces the control to go out of the *innermost* switch or loop

The equivalent `switch-case`…



```c
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                        printf("Your semester is on-time\n");
                        break;
                case 3:
                        printf("Your semester is on-time\n");
                        break;
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

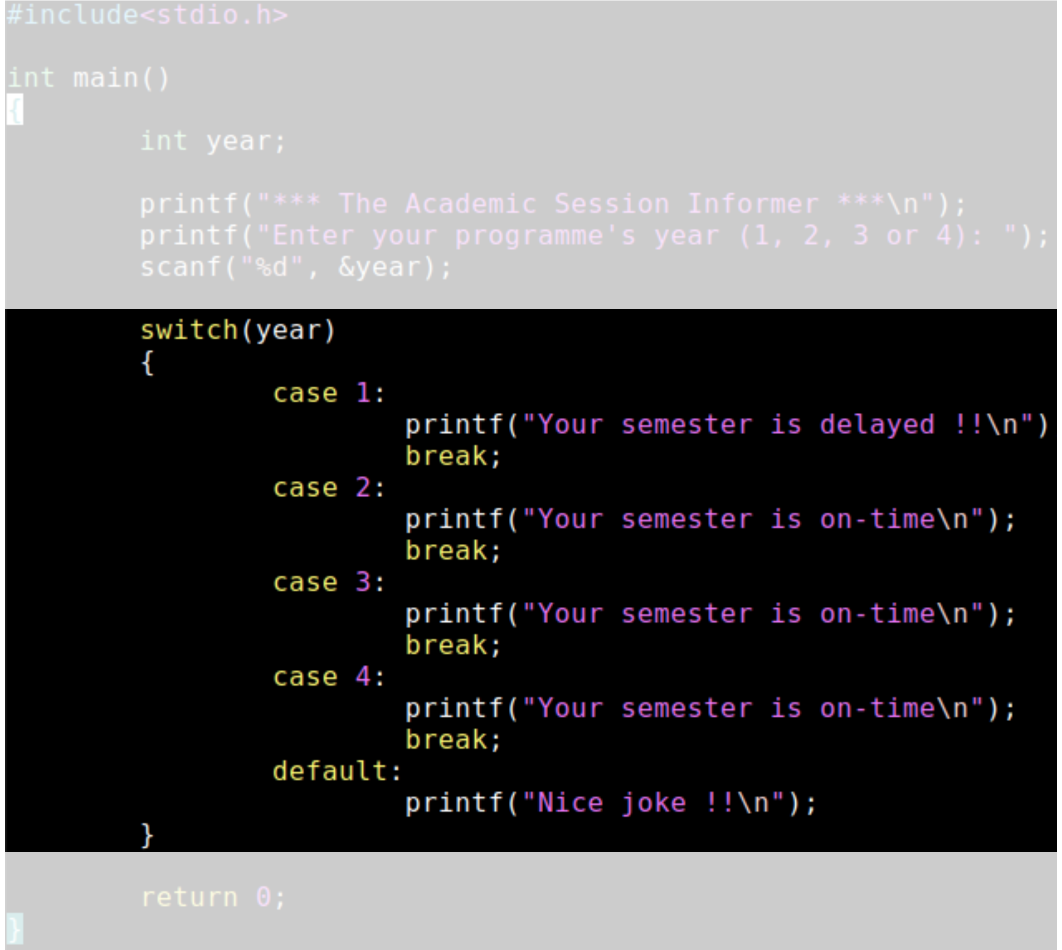
The equivalent `switch-case`…



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                        printf("Your semester is on-time\n");
                        break;
                case 3:
                        printf("Your semester is on-time\n");
                        break;
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

This the same Academic Informer program, but with a `switch-case` construct

The equivalent `switch-case`…

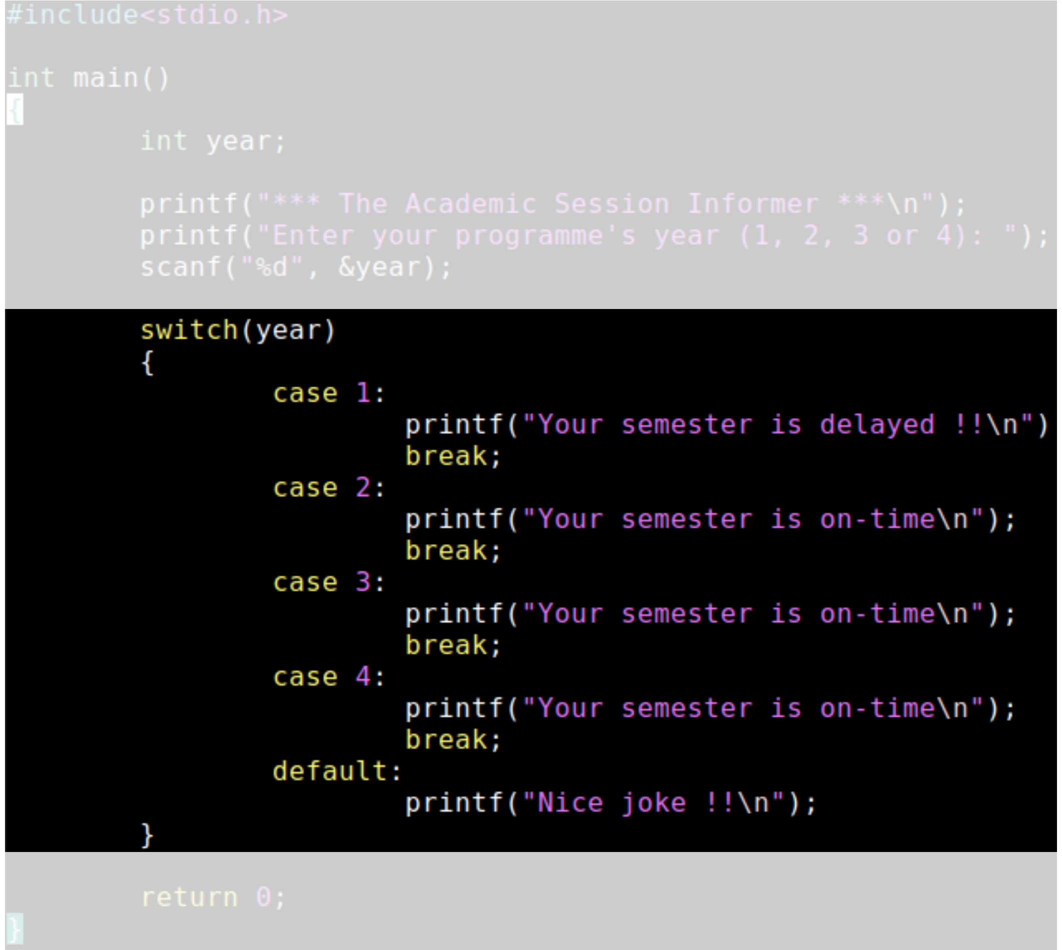


```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                        printf("Your semester is on-time\n");
                        break;
                case 3:
                        printf("Your semester is on-time\n");
                        break;
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

This the same Academic Informer program, but with a `switch-case` construct

By the way, you can see a lot of repetition here, because the code for case `2`, `3` and `4`, is basically the same

The equivalent `switch-case`...



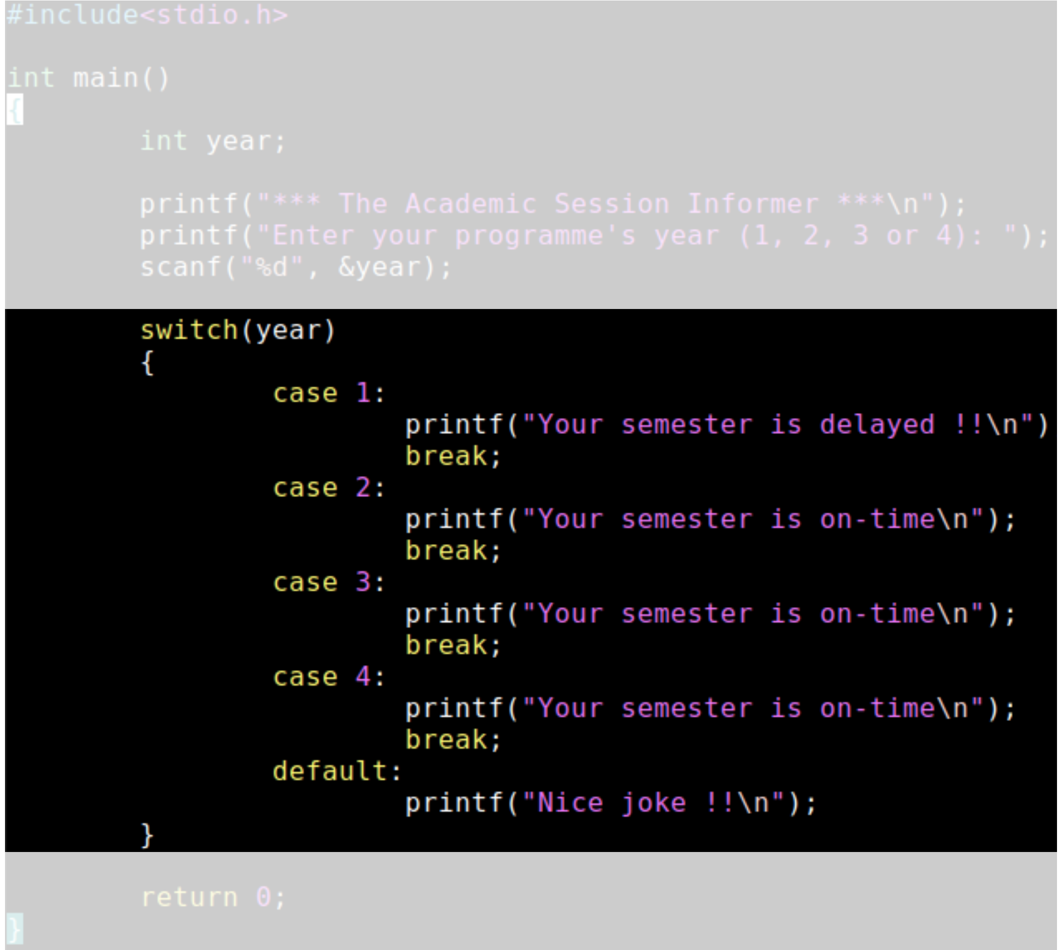
```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                        printf("Your semester is on-time\n");
                        break;
                case 3:
                        printf("Your semester is on-time\n");
                        break;
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

This the same Academic Informer program, but with a `switch-case` construct

By the way, you can see a lot of repetition here, because the code for case `2`, `3` and `4`, is basically the same

This is where, the drawback of switch-case, can become a smart hack !!

The equivalent `switch-case`…



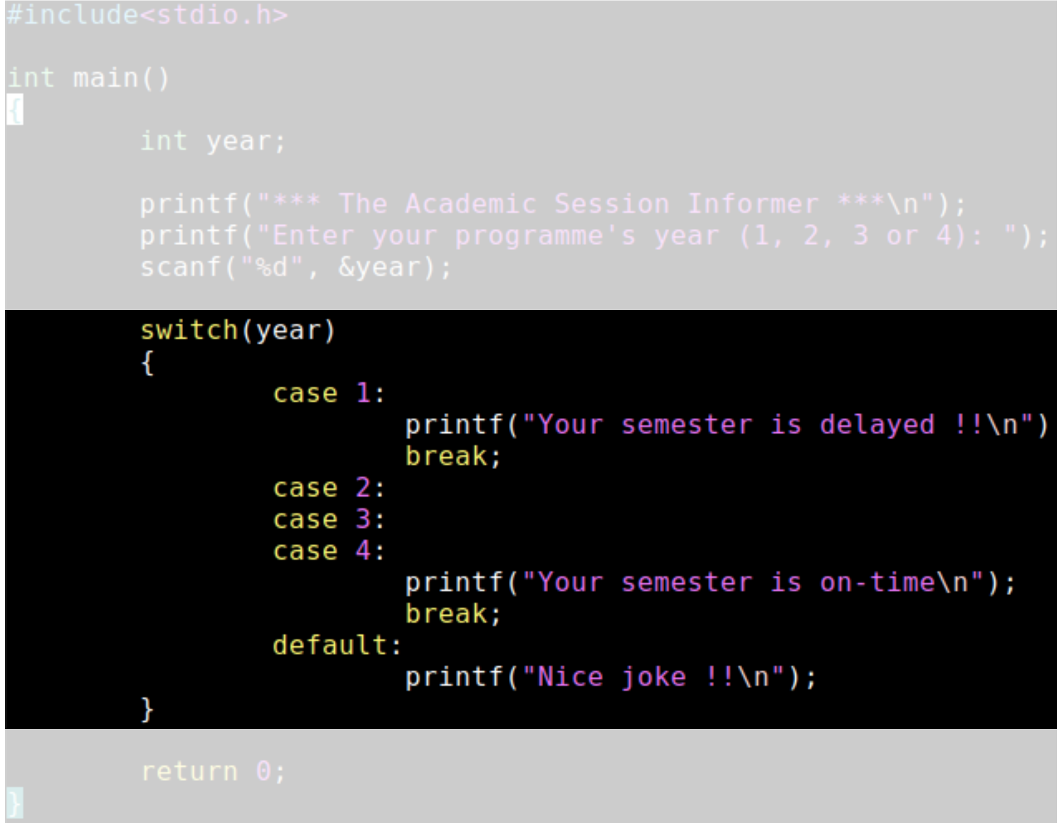
```c
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                        printf("Your semester is on-time\n");
                        break;
                case 3:
                        printf("Your semester is on-time\n");
                        break;
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

This the same Academic Informer program, but with a `switch-case` construct

By the way, you can see a lot of repetition here, because the code for case `2`, `3` and `4`, is basically the same

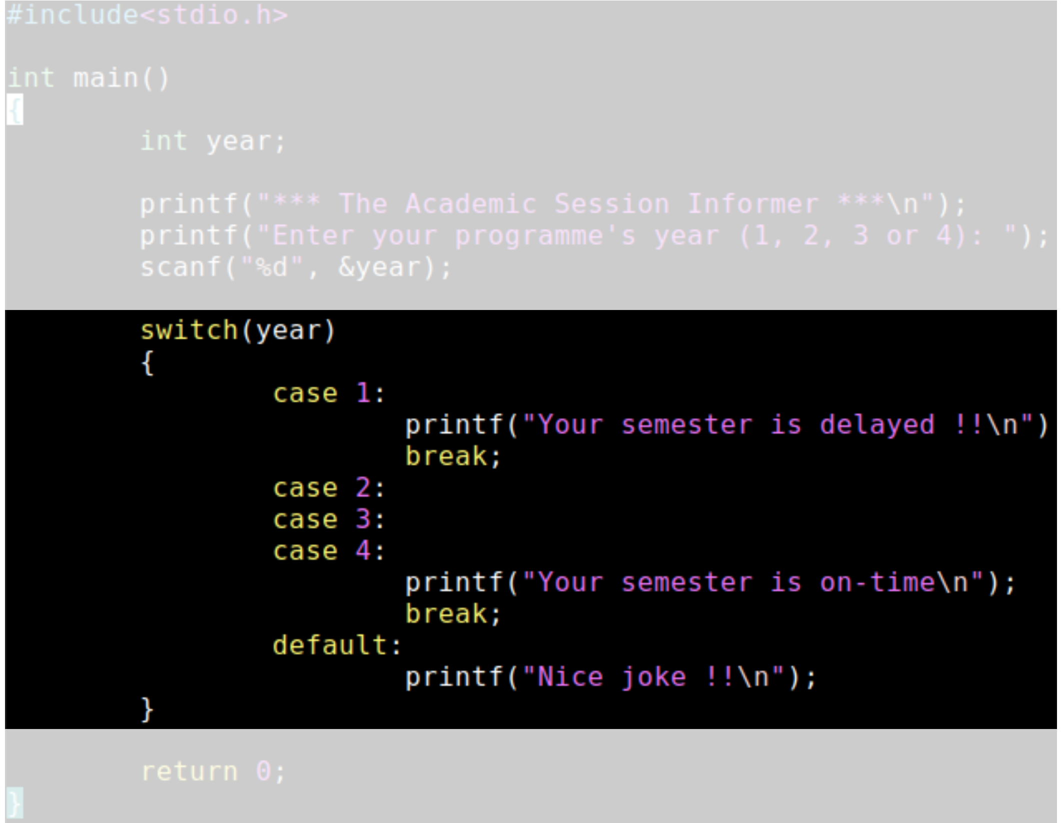
This is where, the drawback of switch-case, can become a smart hack !!

The equivalent `switch-case`…



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                case 3:
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

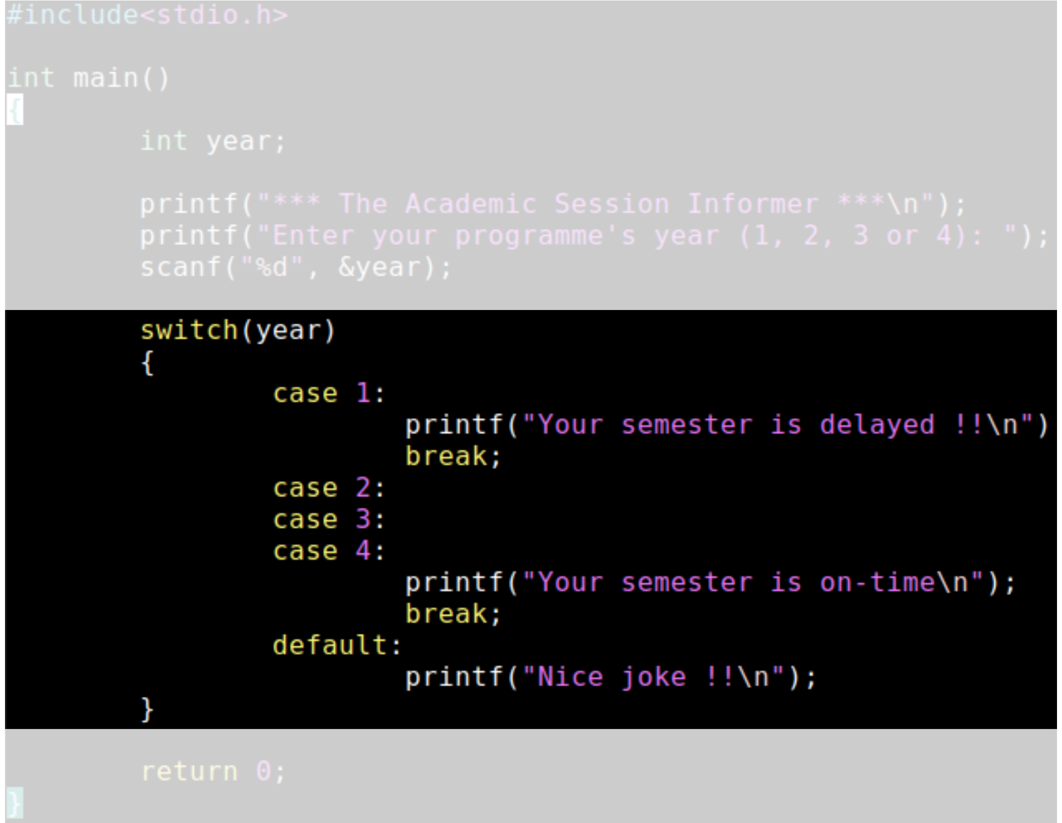
The equivalent `switch-case`…



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                case 3:
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

Something like this !!

The equivalent `switch-case`…

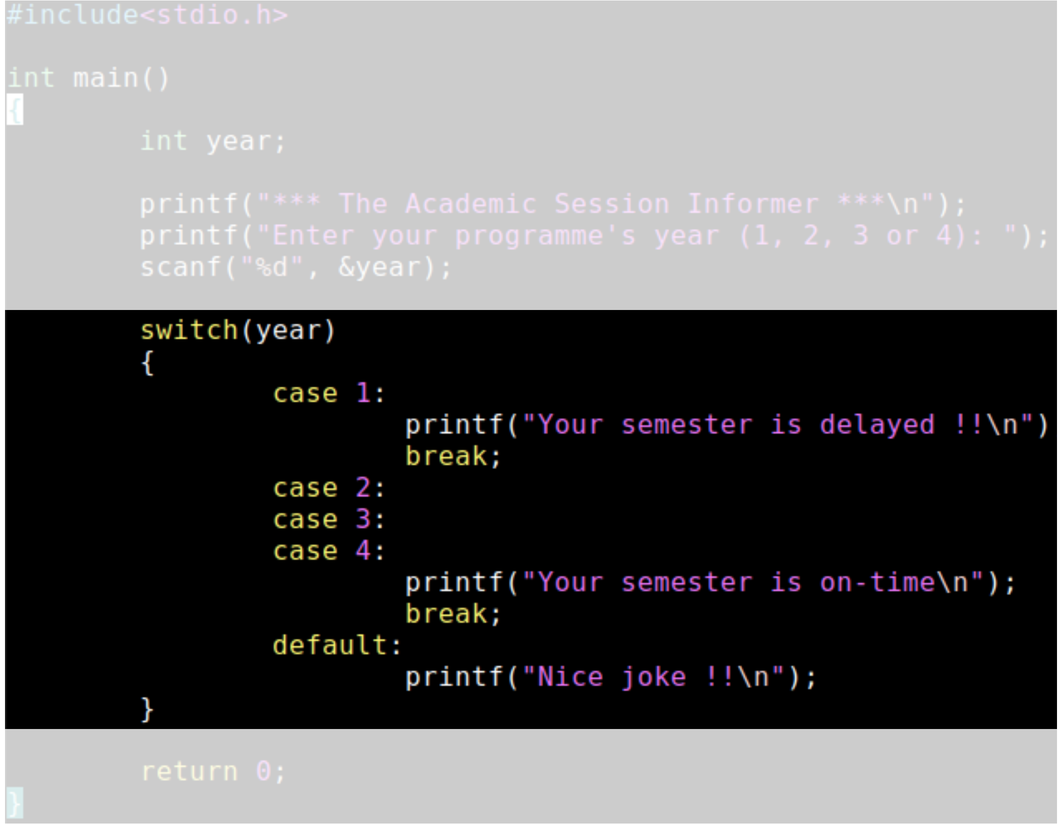


```c
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                case 3:
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

Something like this !!

Actually, we can define cases, without any associated statements, and then chain them like this

The equivalent `switch-case`…



```
#include<stdio.h>

int main()
{
        int year;

        printf("*** The Academic Session Informer ***\n");
        printf("Enter your programme's year (1, 2, 3 or 4): ");
        scanf("%d", &year);

        switch(year)
        {
                case 1:
                        printf("Your semester is delayed !!\n")
                        break;
                case 2:
                case 3:
                case 4:
                        printf("Your semester is on-time\n");
                        break;
                default:
                        printf("Nice joke !!\n");
        }

        return 0;
}
```

Something like this !!

Actually, we can define cases, without any associated statements, and then chain them like this

This way, we can provide the same logic for multiple values of the variable, without having to rewrite it !!

The `break` statement

The `break` statement forces the control to go out of the *innermost* switch or loop

The term “innermost” is important here, because if you have a

The `break` statement

The `break` statement forces the control to go out of the *innermost* switch or loop

The term "innermost" is important here, because if you have a

- … switch inside a loop,

The `break` statement

The `break` statement forces the control to go out of the *innermost* switch or loop

The term “innermost” is important here, because if you have a

- … switch inside a loop,
- … or a loop inside a switch,

The `break` statement

The `break` statement forces the control to go out of the *innermost* switch or loop

The term “innermost” is important here, because if you have a

- … switch inside a loop,
- … or a loop inside a switch,
- … or a *nested* loop or *nested* switch,

The `break` statement

The `break` statement forces the control to go out of the *innermost* switch or loop

The term “innermost” is important here, because if you have a

- … switch inside a loop,
- … or a loop inside a switch,
- … or a *nested* loop or *nested* switch,
- … then `break` will take the control to the immediately next statement after the inner switch or loop

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

```
if(condition1)
{
    ....
    if(condition2)
    {
        // nested if
        ....
    }
    ....
}
```

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

```
if(condition$_1$)
{
    ....
    if(condition$_2$)
    {
        // nested if
        ....
    }
    ....
}
```

```
switch(v)
{
    ....
    case a:
        ....
        switch(v')
        {
            // nested switch
            ....
        }
        ....
}
```

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

```
if(condition1)
{
    ....
    if(condition2)
    {
        // nested if
        ....
    }
    ....
}
```

```
switch(v)
{
    ....
    case a:
        ....
        switch(v')
        {
            // nested switch
            ....
        }
        ....
}
```

```
for(...)
{
    ....
    for(...)
    {
        // nested loop
        ....
    }
    ....
}
```

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

```
if(condition1)
{
      ....
      if(condition2)
      {
            // nested if
            ....
      }
      ....
}
```

```
switch(v)
{
      ....
      case a:
            ....
            switch(v')
            {
                  // nested switch
                  ....
            }
            ....
}
```

By the way, the loops need not be of same type…

```
for(...)
{
      ....
      for(...)
      {
            // nested loop
            ....
      }
      ....
}
```

Detour – What is nesting?

Nesting is a common term used in programming in a rather wide sense

It means the presence of a construct, “inside” an outer construct of the same type

- … e.g. a loop inside another loop is called a “nested” loop
- … or, an `if` statement inside another `if` statement is called a “nested” if statement

Some examples of nesting are

```
if(condition₁)
{
    ....
    if(condition₂)
    {
        // nested if
        ....
    }
    ....
}
```

```
switch(v)
{
    ....
    case a:
        ....
        switch(v')
        {
            // nested switch
            ....
        }
        ....
}
```

By the way, the loops need not be of same type…

```
for(...)
{
    ....
    for(...)
    {
        // nested loop
        ....
    }
    ....
}
```

… e.g. either one could also be `while` or `do-while`

The `break` statement

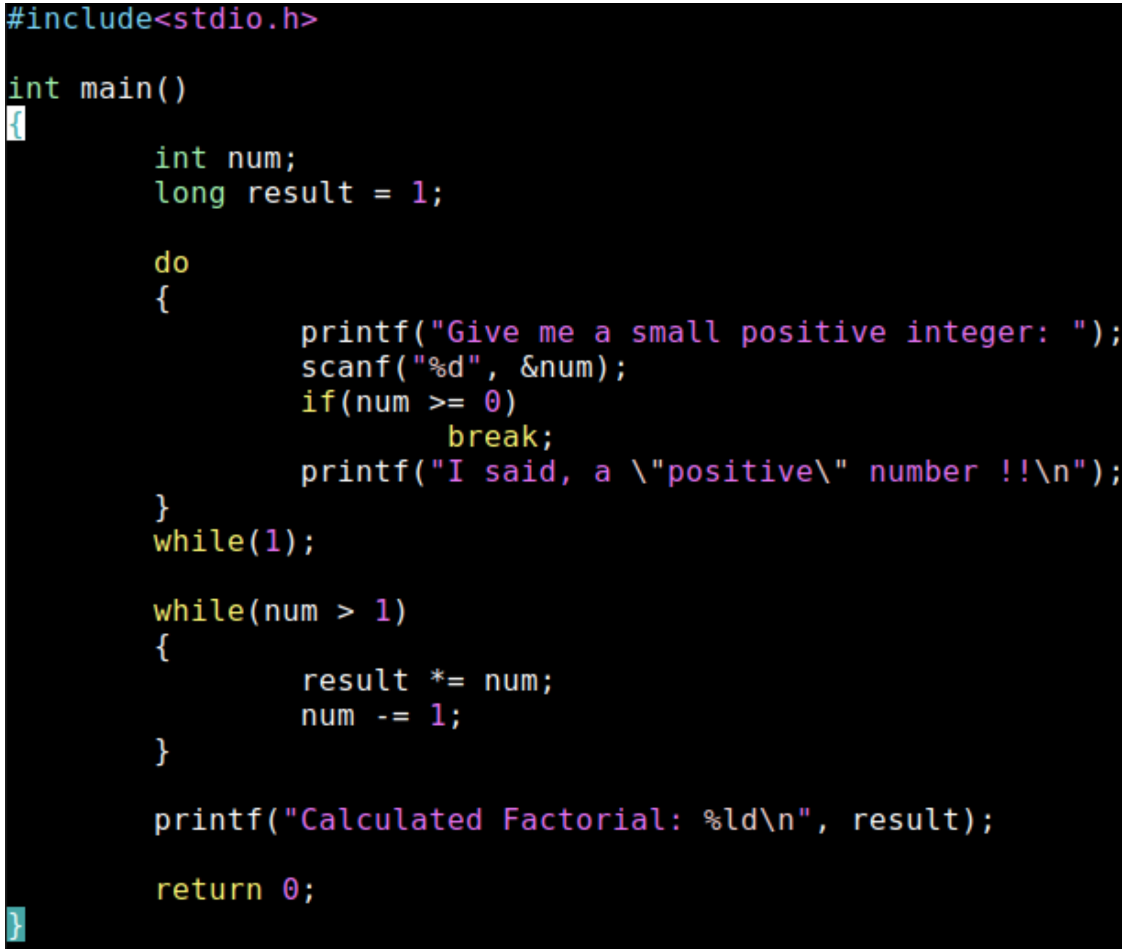
The `break` statement forces the control to go out of the *innermost* switch or loop

The term “innermost” is important here, because if you have a

- … switch inside a loop,
- … or a loop inside a switch,
- … or a *nested* loop or *nested* switch,
- … then `break` will take the control to the immediately next statement after the inner switch or loop

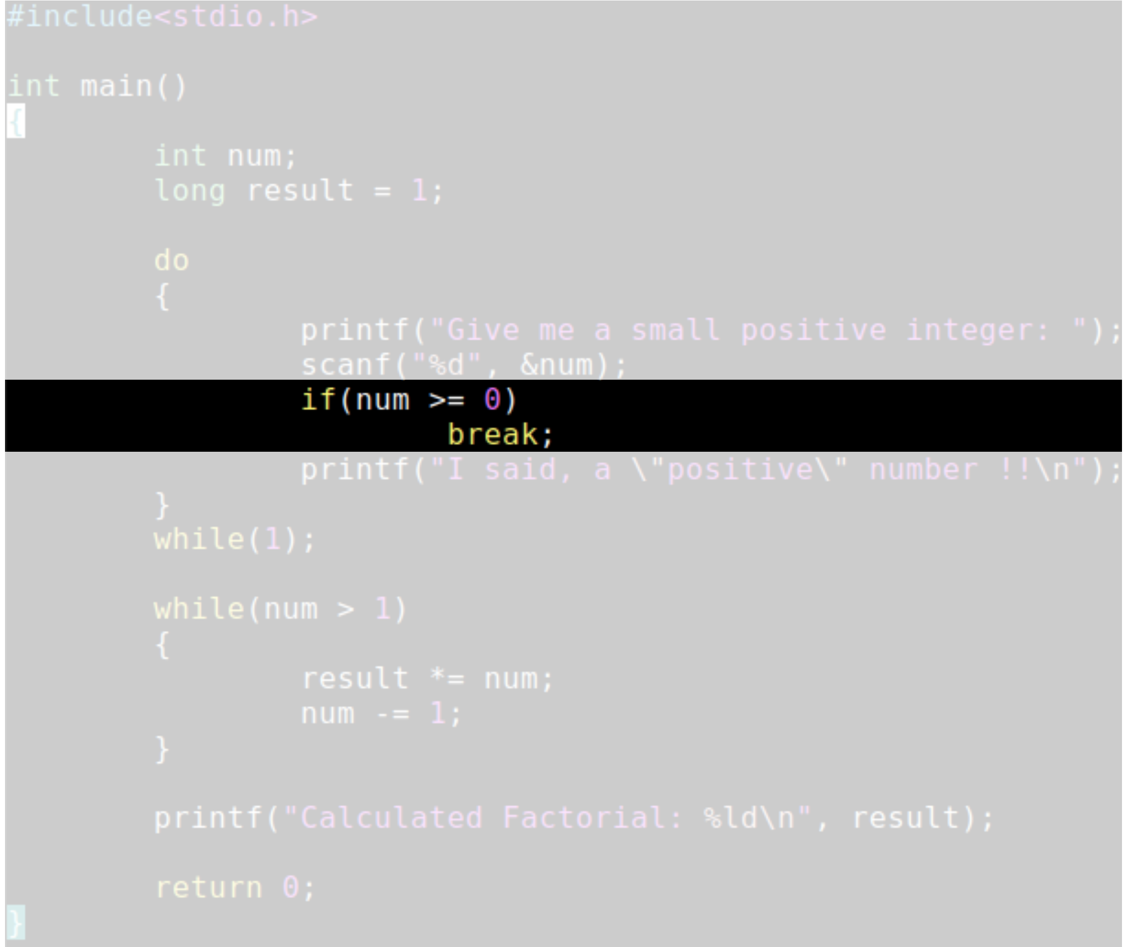
Thus, `break` can be used to put “loop terminating conditions” in the body of the loop as well

One more Factorial Program !!



```c
#include<stdio.h>

int main()
{
        int num;
        long result = 1;

        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
                if(num >= 0)
                        break;
                printf("I said, a \"positive\" number !!\n");
        }
        while(1);

        while(num > 1)
        {
                result *= num;
                num -= 1;
        }

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

        return 0;
}
```

One more Factorial Program !!



```
#include<stdio.h>

int main()
{
        int num;
        long result = 1;

        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
                if(num >= 0)
                        break;
                printf("I said, a \"positive\" number !!\n");
        }
        while(1);

        while(num > 1)
        {
                result *= num;
                num -= 1;
        }

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

        return 0;
}
```

This is a common way to break out of a loop, when a particular condition is true

The `continue` statement

The `break` statement can take the control out of the innermost loop or switch

The `continue` statement

The `break` statement can take the control out of the innermost loop or switch

The `continue` statement can be used to begin the “next iteration” of a loop immediately

- A `continue` statement takes the control to the end of the loop’s body
- … as if, it was the last statement of the loop’s body

The `continue` statement

The `break` statement can take the control out of the innermost loop or switch

The `continue` statement can be used to begin the “next iteration” of a loop immediately

- A `continue` statement takes the control to the end of the loop’s body
- … as if, it was the last statement of the loop’s body

It is not used as often as the `break` statement though !!

The `continue` statement

The `break` statement can take the control out of the innermost loop or switch

The `continue` statement can be used to begin the "next iteration" of a loop immediately

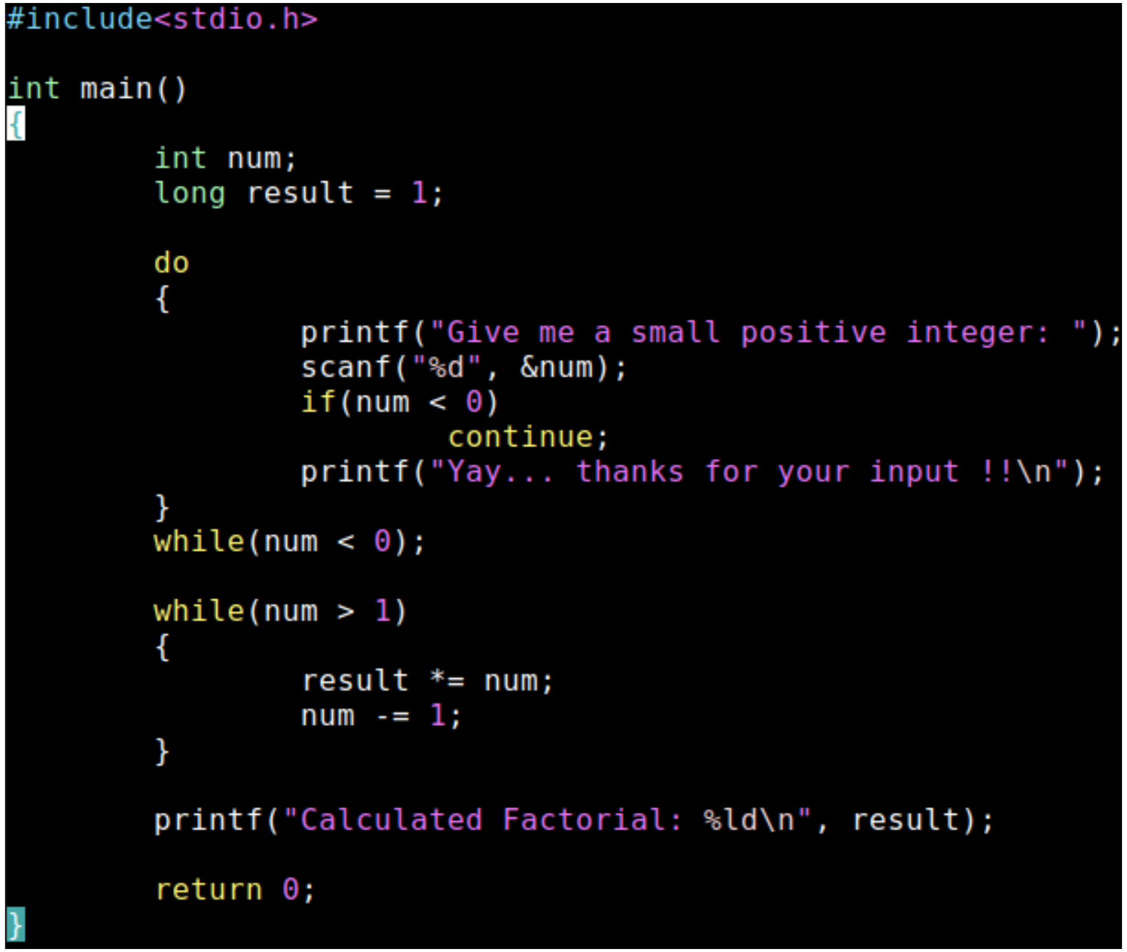
- A `continue` statement takes the control to the end of the loop's body
- … as if, it was the last statement of the loop's body

It is not used as often as the `break` statement though !!

It is not applicable to a switch – only applicable to a loop

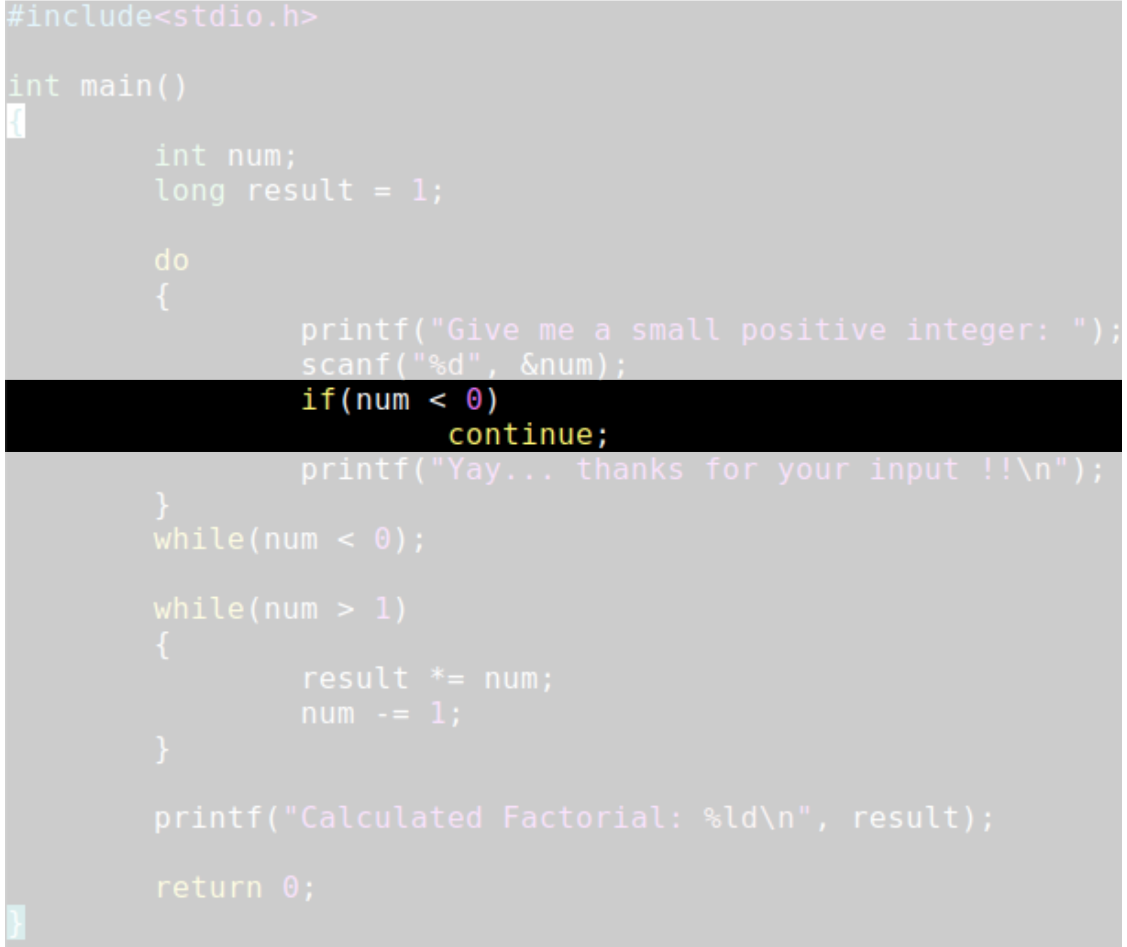
- Thus, if you have a switch within a loop, a `continue` statement inside the switch, will apply to the loop

… and one more !!



```c
#include<stdio.h>

int main()
{
        int num;
        long result = 1;

        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
                if(num < 0)
                        continue;
                printf("Yay... thanks for your input !!\n");
        }
        while(num < 0);

        while(num > 1)
        {
                result *= num;
                num -= 1;
        }

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

        return 0;
}
```

… and one more !!



```
#include<stdio.h>

int main()
{
        int num;
        long result = 1;

        do
        {
                printf("Give me a small positive integer: ");
                scanf("%d", &num);
                if(num < 0)
                        continue;
                printf("Yay... thanks for your input !!\n");
        }
        while(num < 0);

        while(num > 1)
        {
                result *= num;
                num -= 1;
        }

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

        return 0;
}
```

Just a “rather forced” use of `continue` to give you an example !!

Homework !!

Try out `switch-case` examples with a `char` type variable

- Also try them one with `float` or `double`, but with the case values being integers

There is another drawback of `switch-case`, that we have not discussed here; find it out !!

- Hint: try using variables as case values, instead of constants

Convert the following code, to an equivalent code, without any nesting:

```
if(i == 0)
{
    if(j == 0)
        printf("false and false");
    else
        printf("false and true");
}
else
{
    if(j == 0)
        printf("true and false");
    else
        printf("true and true");
}
```