Introduction to Programming

Week – *3*, Lecture – *1*

Conditionals in C – Part 1

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

The use of braces

Procedure QuadraticEquationSolver

```
    Inputs: a, b, c

    D = b * b - 4 * a * c;
    if (D = 0)
    {
        x1 = x2 = -b / (2 * a)
    }
    else if (D > 0)
    {
        x1 = (-b + √D) / (2 * a)
        x2 = (-b - √D) / (2 * a)
        return as Output : x1, x2
    }
    else
    {
        return as Output : "No real roots"
    }
```

The use of braces

Procedure QuadraticEquationSolver

```
Inputs: a, b, c

D = b * b - 4 * a * c;
if (D = 0)
{
    x1 = x2 = -b / (2 * a)
}
else if (D > 0)
{
    x1 = (-b + √D) / (2 * a)
    x2 = (-b - √D) / (2 * a)
    return as Output : x1, x2
}
else
{
    return as Output : "No real roots"
}
```

Remember the full pseudocode for solving Quadratic Equations?

The use of braces

```
Procedure QuadraticEquationSolver
    Inputs: a, b, c

    D = b * b - 4 * a * c;
    if (D = 0)
    {
        x1 = x2 = -b / (2 * a)
    }
    else if (D > 0)
    {
        x1 = (-b + √D) / (2 * a)
        x2 = (-b - √D) / (2 * a)
        return as Output : x1, x2
    }
    else
    {
        return as Output : "No real roots"
    }
```

Remember the full pseudocode for solving Quadratic Equations?

What purpose did these braces serve?

The use of braces

```
Procedure QuadraticEquationSolver
    Inputs: a, b, c

    D = b * b - 4 * a * c;
    if (D = 0)
    {
        x1 = x2 = -b / (2 * a)
    }
    else if (D > 0)
    {
        x1 = (-b + √D) / (2 * a)
        x2 = (-b - √D) / (2 * a)
        return as Output : x1, x2
    }
    else
    {
        return as Output : "No real roots"
    }
```

Remember the full pseudocode for solving Quadratic Equations?

What purpose did these braces serve?

They simply marked start and end of a set of steps for a particular scenario

Code “blocks”

In C, we use paired braces to create *code blocks*

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

- … unless, a *branching* construct is encountered

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

- … unless, a *branching* construct is encountered
- We will look one class of branching construct in this lecture… there are others

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

- … unless, a *branching* construct is encountered
- We will look one class of branching construct in this lecture… there are others

There is a code block for the `main()` method, inside which, we wrote the code in previous examples

Code “blocks”

```c
#include <stdio.h>
#include <math.h>

int main()
{
        int a, b, c, D;

        printf("Please formulate your equation in the form ax^2 + bx + c = 0\n");
        printf("Then provide the values for the parameters in the order a, b and c\n");
        printf("Example: a=1, b=2, c=-15\n");
        scanf("a=%d, b=%d, c=%d", &a, &b, &c);

        D = b * b - 4 * a * c;

        double rootD = sqrt(D);

        double x1 = (-b + rootD) / (2 * a);
        double x2 = (-b - rootD) / (2 * a);

        printf("The roots of the equation (%d)x^2 + (%d)x + (%d) = 0 are %lf and %lf\n", a, b, c, x1, x2);
        return 0;
}
```

Code “blocks”

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main()
5  {
6          int a, b, c, D;
7
8          printf("Please formulate your equation in the form ax^2 + bx + c = 0\n");
9          printf("Then provide the values for the parameters in the order a, b and c\n");
10         printf("Example: a=1, b=2, c=-15\n");
11         scanf("a=%d, b=%d, c=%d", &a, &b, &c);
12
13         D = b * b - 4 * a * c;
14
15         double rootD = sqrt(D);
16
17         double x1 = (-b + rootD) / (2 * a);
18         double x2 = (-b - rootD) / (2 * a);
19
20         printf("The roots of the equation (%d)x^2 + (%d)x + (%d) = 0 are %lf and %lf\n", a, b, c, x1, x2);
21         return 0;
22 }
```

These are the braces I am talking about !!

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

- … unless, a *branching* construct is encountered
- We will look one class of branching construct in this lecture… there are others

There is a code block for the `main()` method, inside which, we wrote the code in previous examples

- Since we have not yet used any branching, it means the code inside the block is executed *sequentially*
- This means, “all the instructions” of the program will be executed

Code “blocks”

In C, we use paired braces to create *code blocks*

A code block is a collection of statements, which are executed in the order they appear

- … unless, a *branching* construct is encountered
- We will look one class of branching construct in this lecture… there are others

There is a code block for the `main()` method, inside which, we wrote the code in previous examples

- Since we have not yet used any branching, it means the code inside the block is executed *sequentially*
- This means, “all the instructions” of the program will be executed

But sometimes, not all the instructions in a program should be executed in every case

- For example, if `D < 0`, you would not like the following statement to be executed:
  `double rootD = sqrt(D);`
- For this you need some “branching” mechanism

What is branching?

Remember the Program Counter or PC register?

- It gets incremented by one after each instruction is executed
- … this is the default processing mode of a computer, unless it is not explicitly changed
- … something like the first law of motion !! ;)

What is branching?

Remember the Program Counter or PC register?

- It gets incremented by one after each instruction is executed
- … this is the default processing mode of a computer, unless it is not explicitly changed
- … something like the first law of motion !! ;)

Branching is a mechanism, which allows you to break sequential code execution

- Basically, it is the equivalent of an “external force”, that can change the acceleration or direction :D

What is branching?

Remember the Program Counter or PC register?

- It gets incremented by one after each instruction is executed
- … this is the default processing mode of a computer, unless it is not explicitly changed
- … something like the first law of motion !! ;)

Branching is a mechanism, which allows you to break sequential code execution

- Basically, it is the equivalent of an “external force”, that can change the acceleration or direction :D

It may be required if your program contains different codes for different scenarios

- … like in our example, you would like different instructions to be executed for different values of `D`

What is branching?

Remember the Program Counter or PC register?

- It gets incremented by one after each instruction is executed
- … this is the default processing mode of a computer, unless it is not explicitly changed
- … something like the first law of motion !! ;)

Branching is a mechanism, which allows you to break sequential code execution

- Basically, it is the equivalent of an “external force”, that can change the acceleration or direction :D

It may be required if your program contains different codes for different scenarios

- … like in our example, you would like different instructions to be executed for different values of `D`

All programming languages provide constructs for branching

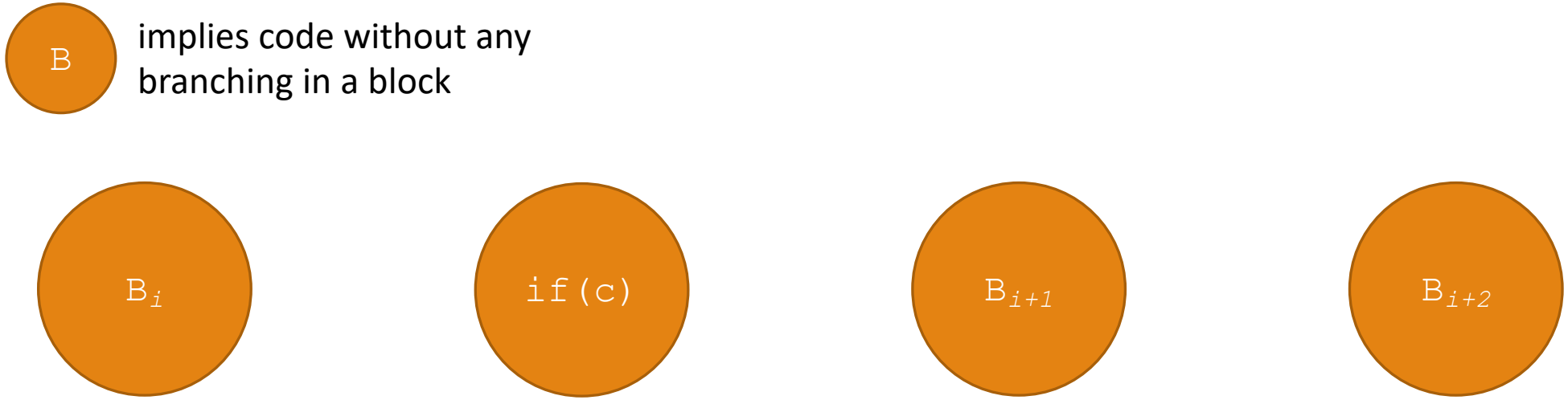
- At Assembly level, we have instructions to explicitly change the value of PC to something else

The `if` construct

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

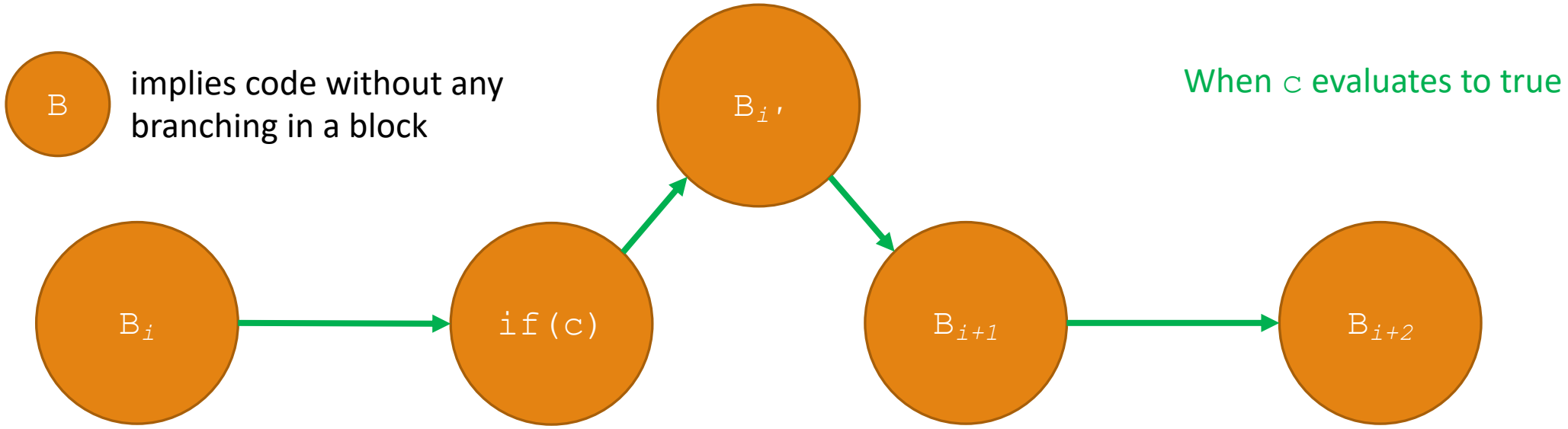
The `if` construct

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



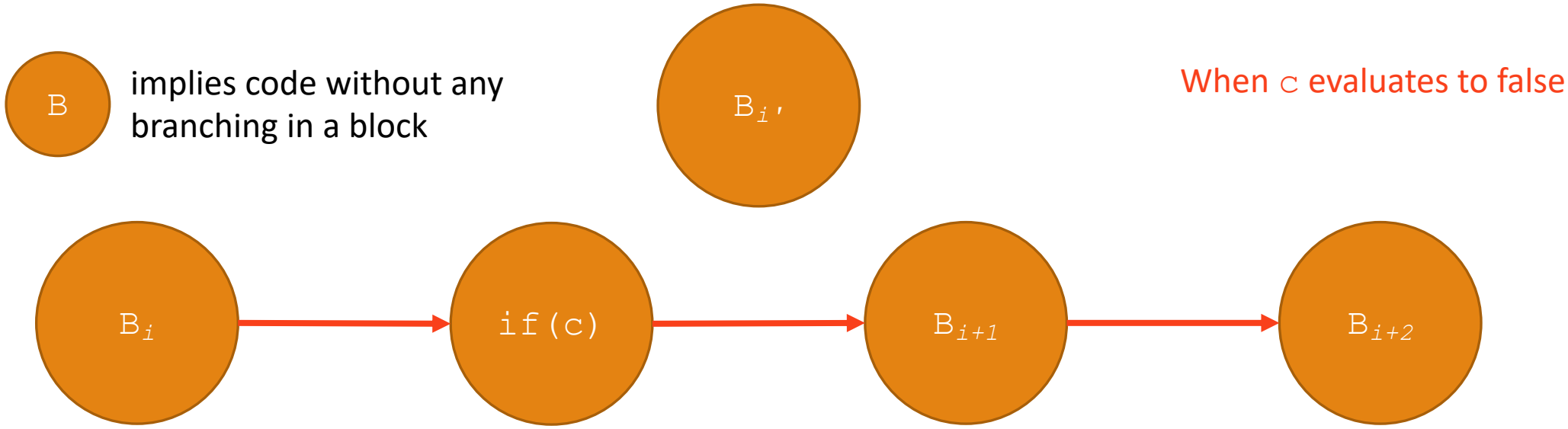
The `if` construct

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



The `if` construct

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



The `if` construct

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        if(percentage >= 75) {
                printf("Looks like you got a Distinction !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

The `if` construct

```
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        if(percentage >= 75) {
                printf("Looks like you got a Distinction !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

B_i

$B_{i'}$

B_{i+1}

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCalculator DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCalculator DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

Since the condition is true, the evaluation goes on like:

$B_i \rightarrow B_{i'} \rightarrow B_{i+1}$

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCalculator DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 305
... and the maximum  marks: 500
You got 61.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

The `if` construct

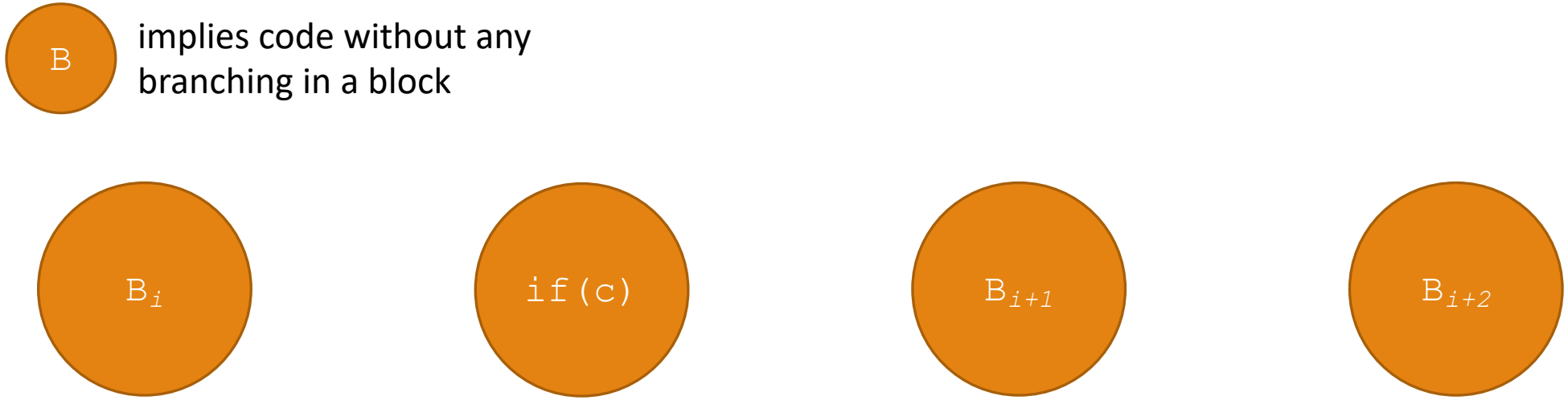
```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCalculator DistinctionPrinter.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCalculator
Tell me your total marks: 305
... and the maximum  marks: 500
You got 61.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

Since the condition is false, the evaluation goes on like:

$B_i \rightarrow B_{i+1}$

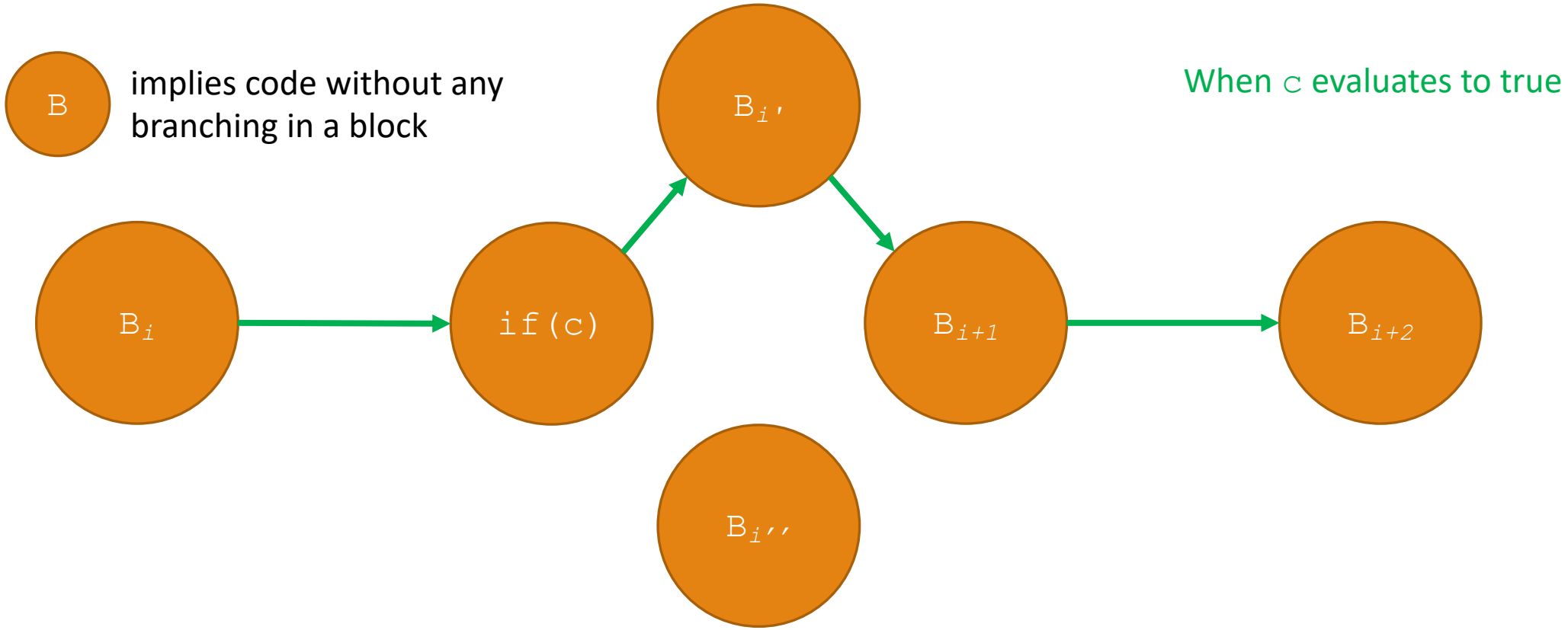
The `if-else` ladder

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



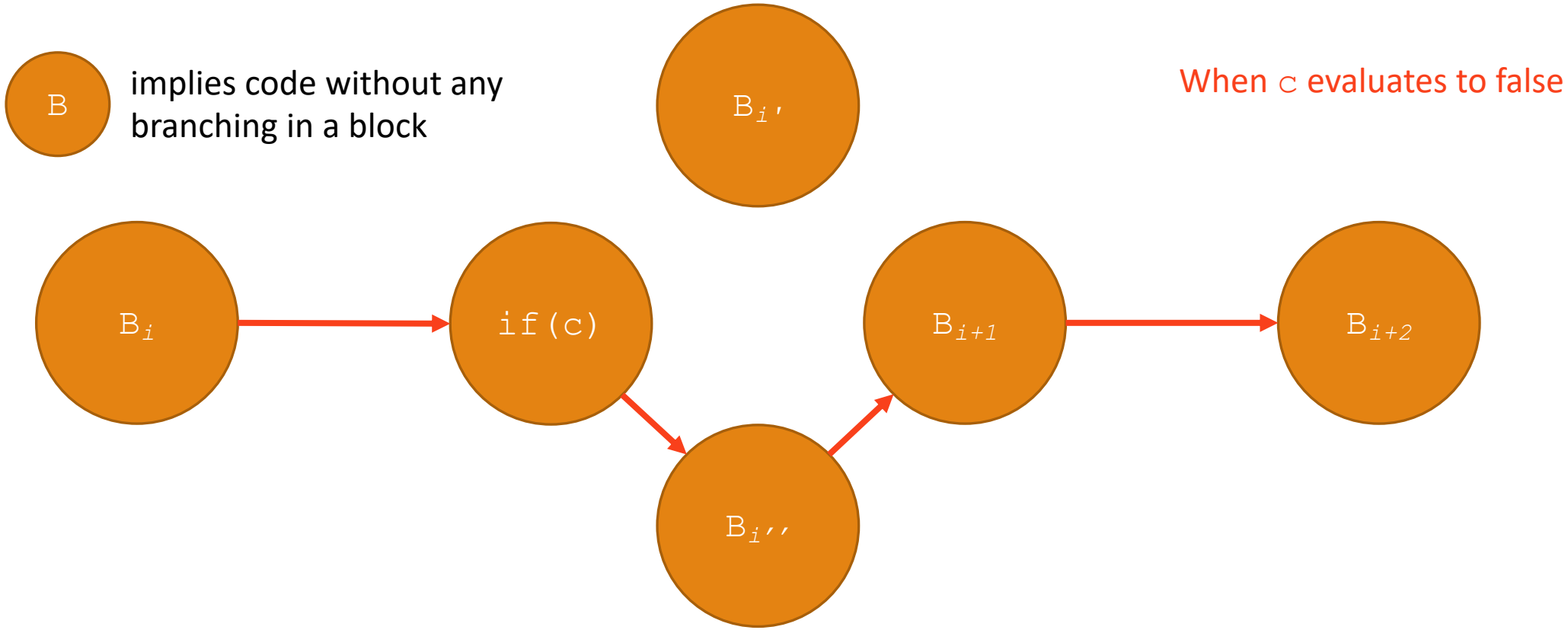
The `if-else` ladder

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



The `if-else` ladder

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



The `if-else` ladder

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        if(percentage >= 75) {
                printf("Looks like you got a Distinction !!\n");
        } else {
                printf("There is a lot of scope for improvement !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

The `if-else` ladder

```
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        if(percentage >= 75) {
                printf("Looks like you got a Distinction !!\n");
        } else {
                printf("There is a lot of scope for improvement !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

B_i

$B_{i'}$

$B_{i''}$

B_{i+1}

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCommentator PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCommentator PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

Since the condition is true, the evaluation goes on like:

$B_i \rightarrow B_{i'} \rightarrow B_{i+1}$

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCommentator PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 305
... and the maximum  marks: 500
There is a lot of scope for improvement !!
You got 61.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

The `if` construct

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o PercentageCommentator PercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 395
... and the maximum  marks: 500
Looks like you got a Distinction !!
You got 79.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./PercentageCommentator
Tell me your total marks: 305
... and the maximum  marks: 500
There is a lot of scope for improvement !!
You got 61.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

Since the condition is false, the evaluation goes on like:

$B_i \rightarrow B_{i''} \rightarrow B_{i+1}$

The `if-else` ladder

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

An if statement can also be added right after an else, to create an `if-else-if` ladder

The `if-else` ladder

```c
#include <stdio.h>
#include <math.h>

int main()
{
        int a, b, c, D;

        printf("Please formulate your equation in the form ax^2 + bx + c = 0\n");
        printf("Then provide the values for the parameters in the order a, b and c\n");
        printf("Example: a=1, b=2, c=-15\n");
        scanf("a=%d, b=%d, c=%d", &a, &b, &c);

        D = b * b - 4 * a * c;

        if(D == 0) {
                double x = -b / 2 * a;
                printf("The root of the equation (%d)x^2 + (%d)x + (%d) = 0 is %lf\n", a, b, c, x);
        }
        else if(D > 0) {
                double rootD = sqrt(D);
                double x1 = (-b + rootD) / (2 * a);
                double x2 = (-b - rootD) / (2 * a);
                printf("The roots of the equation (%d)x^2 + (%d)x + (%d) = 0 are %lf and %lf\n", a, b, c, x1, x2);
        }
        else {
                printf("The equation (%d)x^2 + (%d)x + (%d) = 0 does not have real roots\n", a, b, c);
        }

        return 0;
}
```

The `if-else` ladder

```c
#include <stdio.h>
#include <math.h>

int main()
{
        int a, b, c, D;

        printf("Please formulate your equation in the form ax^2 + bx + c = 0\n");
        printf("Then provide the values for the parameters in the order a, b and c\n");
        printf("Example: a=1, b=2, c=-15\n");
        scanf("a=%d, b=%d, c=%d", &a, &b, &c);

        D = b * b - 4 * a * c;

        if(D == 0) {
                double x = -b / 2 * a;
                printf("The root of the equation (%d)x^2 + (%d)x + (%d) = 0 is %lf\n", a, b, c, x);
        }
        else if(D > 0) {
                double rootD = sqrt(D);
                double x1 = (-b + rootD) / (2 * a);
                double x2 = (-b - rootD) / (2 * a);
                printf("The roots of the equation (%d)x^2 + (%d)x + (%d) = 0 are %lf and %lf\n", a, b, c, x1, x2);
        }
        else {
                printf("The equation (%d)x^2 + (%d)x + (%d) = 0 does not have real roots\n", a, b, c);
        }

        return 0;
}
```

Something like this !!

Finer points about conditions in C – (1/3)

A condition always results an `int` value after evaluation

- This value is 0 if the condition is false, and non-zero (usually 1) if the condition is true

Finer points about conditions in C – (1/3)

A condition always results an `int` value after evaluation

- This value is 0 if the condition is false, and non-zero (usually 1) if the condition is true

There are a set of operators in C which are used for making comparisons

- They are <, >, <=, >=, != (the not equal to operator) and ==

Finer points about conditions in C – (1/3)

A condition always results an `int` value after evaluation

- This value is 0 if the condition is false, and non-zero (usually 1) if the condition is true

There are a set of operators in C which are used for making comparisons

- They are <, >, <=, >=, != (the not equal to operator) and ==

One of the most common confusions for beginners in C, is the difference between = and ==

- = is assignment, it "assigns" the "value" on the right to a "variable" on the left
- You **cannot** have a constant on the LHS of =; on RHS, you can have either a variable or a constant
- == is a condition, it "checks" if the "value" on the right is equal to the "value" on the left
- You can have a variable or constant on either side of ==

Finer points about conditions in C – (1/3)

A condition always results an `int` value after evaluation

- This value is 0 if the condition is false, and non-zero (usually 1) if the condition is true

There are a set of operators in C which are used for making comparisons

- They are <, >, <=, >=, != (the not equal to operator) and ==

One of the most common confusions for beginners in C, is the difference between = and ==

- = is assignment, it "assigns" the "value" on the right to a "variable" on the left
- You **cannot** have a constant on the LHS of =; on RHS, you can have either a variable or a constant
- == is a condition, it "checks" if the "value" on the right is equal to the "value" on the left
- You can have a variable or constant on either side of ==

Assignment operator also "returns" a value – the value on RHS

- So, if you write something like this by mistake: `if(i = j)` instead of `if(i == j)`
- there won't be an error, but will be evaluated as true, if `j` is anything but `0`

Finer points about conditions in C – (2/3)

The statements to execute when a particular condition is true (or false) are put inside braces

- Basically, you create a code block for each case

Finer points about conditions in C – (2/3)

The statements to execute when a particular condition is true (or false) are put inside braces

- Basically, you create a code block for each case

There is an exception – if your code block has a single statement, you can skip the braces

- Basically, these two versions are equivalent

```
if(x==y)
{
    printf("true");
}
else
{
    printf("false");
}
```

```
if(x==y)
    printf("true");
else
    printf("false");
```

Finer points about conditions in C – (2/3)

The statements to execute when a particular condition is true (or false) are put inside braces

- Basically, you create a code block for each case

There is an exception – if your code block has a single statement, you can skip the braces

- Basically, these two versions are equivalent

```
if(x==y)
{
    printf("true");
}
else
{
    printf("false");
}
```

```
if(x==y)
    printf("true");
else
    printf("false");
```

All comparison operators have lower precedence than arithmetic operators

- Within comparison operators, ! = and == have lower precedence than others

Finer points about conditions in C – (2/3)

The statements to execute when a particular condition is true (or false) are put inside braces

- Basically, you create a code block for each case

There is an exception – if your code block has a single statement, you can skip the braces

- Basically, these two versions are equivalent

```
if(x==y)
{
    printf("true");
}
else
{
    printf("false");
}
```

```
if(x==y)
    printf("true");
else
    printf("false");
```

All comparison operators have lower precedence than arithmetic operators

- Within comparison operators, ! = and == have lower precedence than others

However, they have higher precedence than assignment and logical operators

Finer points about conditions in C – (3/3)

What if you need to create a more complex condition?

Finer points about conditions in C – (3/3)

What if you need to create a more complex condition?

You can use the logical operators

Finer points about conditions in C – (3/3)

What if you need to create a more complex condition?

You can use the logical operators

C has three logical operators to represent the three operations in Boolean Algebra

- && is the operator to represent AND operation
- || is the operator to represent OR operation
- ! Is the operator to represent NOT operation

Using logical operators in conditions

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        int passed = percentage >= 33;

        if(!passed)
        {
                printf("Sorry.. you have failed :(\n");
        }
        else {
                if(percentage >= 90)
                        printf("You are doing great !!\n");
                else if(percentage < 90 && percentage >= 75)
                        printf("You are doing well, kep going !!\n");
                else if(percentage < 75 && percentage >= 50)
                        printf("Keep working hard, you can do it !!\n");
                else
                        printf("There's still time... push hard !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

Using logical operators in conditions

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        int passed = percentage >= 33;

        if(!passed)
        {
                printf("Sorry.. you have failed :(\n");
        }
        else {
                if(percentage >= 90)
                        printf("You are doing great !!\n");
                else if(percentage < 90 && percentage >= 75)
                        printf("You are doing well, kep going !!\n");
                else if(percentage < 75 && percentage >= 50)
                        printf("Keep working hard, you can do it !!\n");
                else
                        printf("There's still time... push hard !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

The use of ! reverses the condition (true becomes false and vice-versa)

Using logical operators in conditions

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        int passed = percentage >= 33;

        if(!passed)
        {
                printf("Sorry.. you have failed :(\n");
        }
        else {
                if(percentage >= 90)
                        printf("You are doing great !!\n");
                else if(percentage < 90 && percentage >= 75)
                        printf("You are doing well, kep going !!\n");
                else if(percentage < 75 && percentage >= 50)
                        printf("Keep working hard, you can do it !!\n");
                else
                        printf("There's still time... push hard !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

The use of ! reverses the condition (true becomes false and vice-versa)

&& returns true if both conditions are true

Using logical operators in conditions

```c
#include<stdio.h>

int main()
{
        int total, maximum;

        printf("Tell me your total marks: ");
        scanf("%d", &total);

        printf("... and the maximum  marks: ");
        scanf("%d", &maximum);

        float percentage = 100.0 * total / maximum;

        int passed = percentage >= 33;

        if(!passed)
        {
                printf("Sorry.. you have failed :(\n");
        }
        else {
                if(percentage >= 90)
                        printf("You are doing great !!\n");
                else if(percentage < 90 && percentage >= 75)
                        printf("You are doing well, kep going !!\n");
                else if(percentage < 75 && percentage >= 50)
                        printf("Keep working hard, you can do it !!\n");
                else
                        printf("There's still time... push hard !!\n");
        }

        printf("You got %f%% marks\n", percentage);
}
```

The use of ! reverses the condition (true becomes false and vice-versa)

&& returns true if both conditions are true

Similarly, || (not used in this example) would have returned true, if any of the condition was true

Using logical operators in conditions

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DetailedPercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o DPC DetailedPercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 50
... and the maximum  marks: 500
Sorry.. you have failed :(
You got 10.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 300
... and the maximum  marks: 500
Keep working hard, you can do it !!
You got 60.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 390
... and the maximum  marks: 500
You are doing well, kep going !!
You got 78.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

Using logical operators in conditions

```
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ vim DetailedPercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ gcc -o DPC DetailedPercentageCommentator.c
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 50
... and the maximum  marks: 500
Sorry.. you have failed :(
You got 10.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 300
... and the maximum  marks: 500
Keep working hard, you can do it !!
You got 60.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$ ./DPC
Tell me your total marks: 390
... and the maximum  marks: 500
You are doing well, kep going !!
You got 78.000000% marks
saurabh@saurabh-VirtualBox:~/C/examples/Week 3$
```

You can see how different code blocks get executed in different cases

Homework !! (They'll get intense now…)

Find out what `-o` switch does with gcc

- While you are at it, also find out about `-g` and `-c` switches

In the examples related to calculation of percentages

- … replace "`100.0`" with "`100`", and try out some inputs
- Do you find anything interesting? Find out the reasons behind that !!
- May be this link could help you understand: https://stackoverflow.com/questions/3602827/what-is-the-behavior-of-integer-division

Read more about implicit and explicit type conversions in C

- See this link for getting an overview: https://www.guru99.com/c-type-casting.html