Conditionals and Loops

Decision Making

The **if** statement is used to execute some code if a condition is true.

Syntax:

```
if (condition) {
  statements
}
```

The **condition** specifies which expression is to be evaluated. If the condition is true, the statements in the curly brackets are executed.

> If the condition is **false**, the statements are simply ignored, and the program continues to run after the if statements body.

The if Statement

Use **relational operators** to evaluate conditions.

For example:

```
if (7 > 4) {
  cout << "Yes";
}

// Outputs "Yes"
```

Try It Yourself

The **if** statement evaluates the condition (7>4), finds it to be **true**, and then executes the cout statement.
If we change the greater operator to a less than operator (**7<4**), the statement will not be executed and nothing will be printed out.

> A condition specified in an if statement does not require a semicolon.

Relational Operators

Additional relational operators:

Operator	Description	Example	
>=	Greater than or equal to	7 >= 4	True
<=	Less than or equal to	7 <= 4	False
==	Equal to	7 == 4	False
!=	Not equal to	7 != 4	True

Example:

```
if (10 == 10) {
  cout <<  "Yes";
}

// Outputs "Yes"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Relational Operators

The **not equal to** operator evaluates the operands, determines whether or not they are equal. If the operands are not equal, the condition is evaluated to **true**.
For example:

```
if (10 != 10) {
  cout <<  "Yes";
}
```

Try It Yourself

The above condition evaluates to **false** and the block of code is not executed.

Relational Operators

You can use relational operators to compare variables in the **if** statement.
For example:

```
int a = 55;
int b = 33;
if (a > b) {
  cout << "a is greater than b";
}

// Outputs "a is greater than b"
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

The else Statement

An **if** statement can be followed by an optional **else** statement, which executes when the condition is **false**.
Syntax:

```
if (condition) {
  //statements
}
else {
 //statements
}
```

The code above will test the condition:
- If it evaluates to **true**, then the code inside the **if** statement will be executed.
- If it evaluates to **false**, then the code inside the **else** statement will be executed.

> When only **one** statement is used inside the if/else, then the curly braces can be omitted.

The else Statement

For example:

```
int mark = 90;

if (mark < 50) {
  cout << "You failed." << endl;
}
else {
  cout << "You passed." << endl;
}

// Outputs "You passed."
```

Try It Yourself

> Tap **Try It Yourself** to play around with the code!

The else Statement

In all previous examples only one statement was used inside the if/else statement, but you may include as many statements as you want.

For example:

```
int mark = 90;

if (mark < 50) {
  cout << "You failed." << endl;
  cout << "Sorry" << endl;
}
```

```
else {
  cout << "Congratulations!" << endl;
  cout << "You passed." << endl;
  cout << "You are awesome!" << endl;
}

/* Outputs
Congratulations!
You passed.
You are awesome!
*/
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

Nested if Statements

You can also include, or **nest**, if statements within another if statement.
For example:

```
int mark = 100;

if (mark >= 50) {
  cout << "You passed." << endl;
  if (mark == 100) {
    cout <<"Perfect!" << endl;
  }
}
else {
  cout << "You failed." << endl;
}

/*Outputs
You passed.
Perfect!
*/
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

The if else Statement

In if/else statements, **a single statement** can be included without enclosing it into curly braces.

```
int a = 10;
if (a > 4)
  cout << "Yes";
else
  cout << "No";
```

Try It Yourself

> Including the curly braces anyway is a good practice, as they clarify the code and make it easier to read.

Loops

A **loop** repeatedly executes a set of statements until a particular condition is satisfied.

A **while** loop statement repeatedly executes a target statement as long as a given condition remains **true**.
Syntax:

```
while (condition) {
  statement(s);
}
```

The loop iterates while the condition is **true**.

> At the point when the condition becomes **false**, program control is shifted to the line that immediately follows the loop.

The while Loop

The loop's **body** is the block of statements within curly braces.
For example:

```
int num = 1;
while (num < 6) {
  cout << "Number: " << num << endl;
  num = num + 1;
}

/* Outputs
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
*/
```

Try It Yourself

The example above declares a variable equal to 1 (int num = 1).
The **while** loop checks the condition (num < 6), and executes the statements in its body, which increment the value of **num** by one each time the loop runs.

> After the 5th iteration, **num** becomes 6, and the condition is evaluated to **false**, and the loop stops running.

The while Loop

The increment value can be changed. If changed, the number of times the loop is run will change, as well.

```
int num = 1;
while (num < 6) {
  cout << "Number: " << num << endl;
  num = num + 3;
}

/* Outputs
Number: 1
Number: 4
*/
```

Try It Yourself

Without a statement that eventually evaluates the loop condition to **false**, the loop will continue indefinitely.

The for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that executes a specific number of times.
Syntax:

```
for ( init; condition; increment ) {
  statement(s);
}
```

The **init** step is executed first, and does not repeat.
Next, the **condition** is evaluated, and the body of the loop is executed if the condition is true.
In the next step, the **increment** statement updates the loop control variable.
Then, the loop's body repeats itself, only stopping when the condition becomes **false**.

For example:

```
for (int x = 1; x < 10; x++) {
 // some code
 }
```

The **init** and **increment** statements may be left out, if not needed, but remember that the **semicolons** are mandatory.

The for Loop

The example below uses a **for** loop to print numbers from 0 to 9.

```
for (int a = 0; a < 10; a++) {
  cout << a << endl;
}

/* Outputs
0
1
2
3
4
5
6
7
8
9
*/
```

Try It Yourself

In the **init** step, we declared a variable **a** and set it to equal 0.
a < 10 is the **condition**.
After each iteration, the a++ **increment** statement is executed.

> When **a** increments to 10, the condition evaluates to **false**, and the loop stops.

The for Loop

It's possible to change the increment statement.

```
for (int a = 0; a < 50; a+=10) {
  cout << a << endl;
}
/* Outputs
0
10
20
30
40
*/
```

Try It Yourself

You can also use decrement in the statement.

```
for (int a = 10; a >= 0; a -= 3) {
  cout << a << endl;
}

/* Outputs
10
7
4
1
*/
```

Try It Yourself

When using the for loop, don't forget the **semicolon** after the **init** and **condition** statements.

The do...while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.
A **do...while** loop is similar to a **while** loop. The one difference is that the **do...while** loop is guaranteed to execute **at least one time**.
Syntax:

```
do {
  statement(s);
} while (condition);
```

For example, you can take input from the user, then check it. If the input is wrong, you can take it again.

The do...while Loop

Here is an example:

```
int a = 0;
do {
  cout << a << endl;
  a++;
} while(a < 5);

/* Outputs
0
1
2
3
4
*/
```

Try It Yourself

Don't forget the **semicolon** after the while statement.

---

while vs. do...while

If the condition evaluated to **false**, the statements in the **do** would still run once:

```
int a = 42;
do {
  cout << a << endl;
  a++;
} while(a < 5);

// Outputs 42
```

Try It Yourself

The **do...while** loop executes the statements at least once, and then tests the condition.
The **while** loop executes the statement after testing condition.

The do...while Loop

As with other loops, if the condition in the loop never evaluates to **false**, the loop will run forever.
For example:

```
int a = 42;
do {
  cout << a << endl;
} while (a > 0);
```

Try It Yourself

This will print 42 to the screen **forever**.

Always test your loops, so you know that they operate in the manner you expect.

Multiple Conditions

Sometimes there is a need to test a variable for equality against multiple values. That can be achieved using multiple if statements.
For example:

```
int age = 42;
if (age == 16) {
  cout <<"Too young";
}
if (age == 42) {
  cout << "Adult";
}
if (age == 70) {
  cout << "Senior";
}
```

Try It Yourself

The **switch** statement is a more elegant solution in this scenario.

The switch Statement

The **switch** statement tests a variable against a list of values, which are called **cases**, to determine whether it is equal to any of them.

```
switch (expression) {
  case value1:
    statement(s);
    break;
  case value2:
    statement(s);
    break;
    ...
  case valueN:
    statement(s);
    break;
}
```

Switch evaluates the expression to determine whether it's equal to the value in the case statement. If a match is found, it executes the statements in that case.

A switch can contain any number of **case** statements, which are followed by the **value** in question and a **colon**.

The switch Statement

Here is the previous example written using a single **switch** statement:

```
int age = 42;
switch (age) {
  case 16:
    cout << "Too young";
    break;
  case 42:
    cout << "Adult";
    break;
  case 70:
    cout << "Senior";
    break;
}
```

Try It Yourself

The code above is equivalent to three **if** statements.

Notice the keyword **break;** that follows each case. That will be covered shortly.

The default Case

In a switch statement, the optional **default** case can be used to perform a task when none of the cases is determined to be true.

Example:

```
int age = 25;
switch (age) {
  case 16:
    cout << "Too young";
    break;
  case 42:
    cout << "Adult";
    break;
```

```
  case 70:
    cout << "Senior";
    break;
  default:
    cout << "This is the default case";
}

// Outputs "This is the default case"
```

Try It Yourself

The **default** statement's code executes when none of the cases matches the switch expression.

> The **default** case must appear at the end of the switch.

The break Statement

The **break** statement's role is to terminate the switch statement.
In instances in which the variable is equal to a case, the statements that come after the case continue to execute until they encounter a **break** statement. In other words, leaving out a **break** statement results in the execution of all of the statements in the following cases, even those that don't match the expression.
For example:

```
int age = 42;
switch (age) {
  case 16:
    cout << "Too young" << endl;
  case 42:
    cout << "Adult" << endl;
  case 70:
    cout << "Senior" << endl;
  default:
    cout <<"This is the default case" << endl;
}
/* Outputs
Adult
Senior
This is the default case
*/
```

Try It Yourself

As you can see, the program executed the matching case statement, printing "Adult" to the screen. With no specified **break** statement, the statements continued to run after the matching case. Thus, all the other case statements printed. This type of behavior is called **fall-through**.

> As the switch statement's final case, the **default** case requires no **break** statement.
> The **break** statement can also be used to break out of a loop.

Logical Operators

Use **logical operators** to combine conditional statements and return **true** or **false**.

Operator	Name of Operator	Form
&&	**AND** Operator	y && y
||	**OR** Operator	x || y
!	**NOT** Operator	! x

The **AND** operator works the following way:

Left Operand	Right Operand	Result
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

> In the AND operator, both operands must be **true** for the entire expression to be **true**.

The AND Operator

For example:

```
int age = 20;
if (age > 16 && age < 60) {
  cout << "Accepted!" << endl;
}

// Outputs "Accepted"
```

Try It Yourself

In the example above, the logical AND operator was used to combine both expressions.

> The expression in the if statement evaluates to **true** only if both expressions are **true**.

The AND Operator

Within a single if statement, logical operators can be used to combine **multiple** conditions.

```
int age = 20;
int grade = 80;

if (age > 16 && age < 60 && grade > 50) {
  cout << "Accepted!" << endl;
}
```

Try It Yourself

> The entire expression evaluates to **true** only if all of the conditions are **true**.

The OR Operator

The **OR** (||) operator returns true if any one of its operands is **true**.

Left Operand	Right Operand	Result
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

Example:

```
int age = 16;
int score = 90;
if (age > 20 || score > 50) {
   cout << "Accepted!" << endl;
}

// Outputs "Accepted!"
```

Try It Yourself

You can combine any number of logical **OR** statements you want.
In addition, multiple **OR** and **AND** statements may be chained together.

Logical NOT

The logical **NOT** (!) operator works with just a single operand, reversing its logical state. Thus, if a condition is **true**, the NOT operator makes it **false**, and vice versa.

Right Operand	Result
true	**false**
false	**true**

```
int age = 10;
if ( !(age > 16) ) {
  cout << "Your age is less than 16" << endl;
}
```

```
// Outputs "Your age is less than 16"
```

Try It Yourself

Be careful using this, because !false means true.

End.