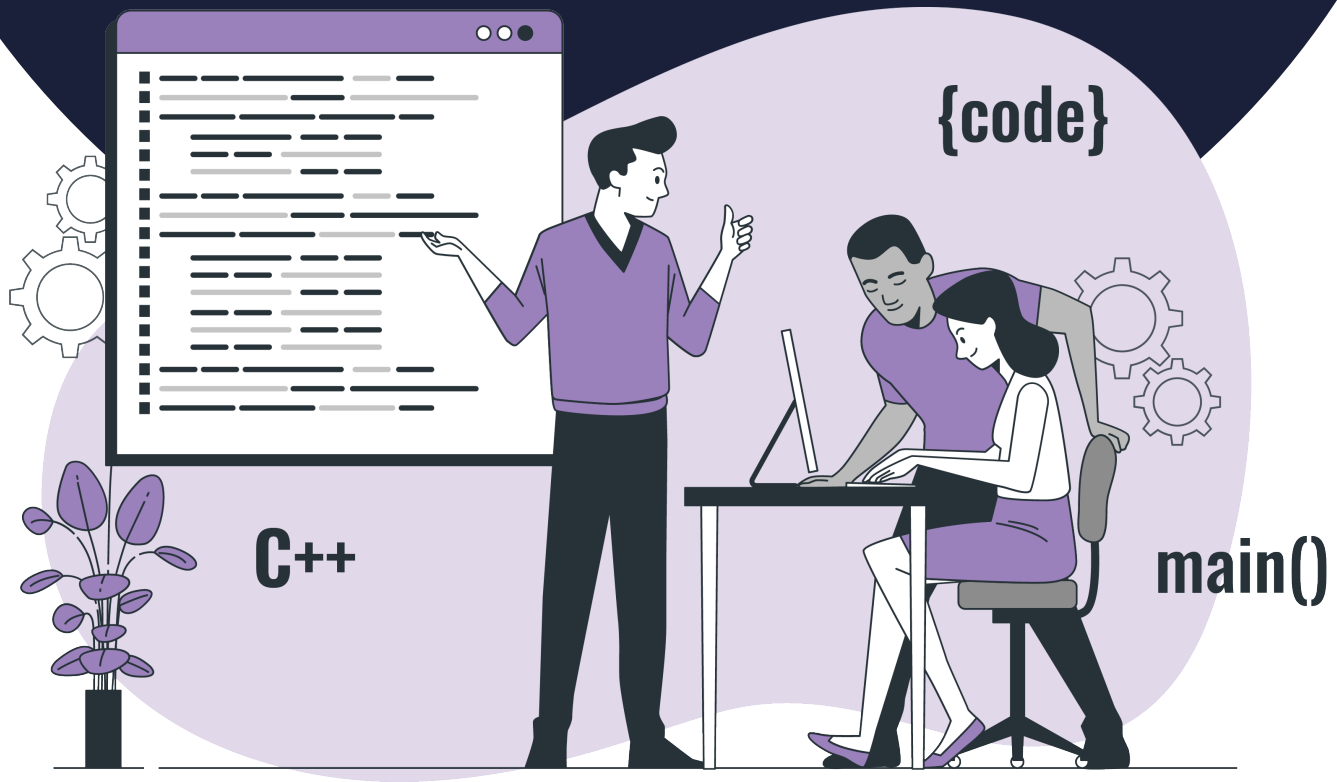
Lesson:

Conditionals



Pre-Requisites

- Basic C++ syntax
- Data types, variables, keywords
- Operators

List of Concepts Involved

- If-statement
- If-else statment
- If-else if statement
- Nested if-else statement
- Conditional operators
- Switch statement

In real life we often encounter situations where our actions are governed by some sort of conditions. For instance, if the weather is rainy, we carry an umbrella. Here carrying an umbrella is an action which is performed only when the condition of the weather being rainy is fulfilled.

In programming, this kind of scenario is handled with the help of conditional statements.
Let us have a look at the different conditional statements supported by C++.

Topic 1: If statement

An *if statement* is the most common conditional statement. It executes when the condition being checked evaluates to true.

Syntax

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

Let us look at this simple example

```
if(marks > 80)
{
      cout << “Pass\n”;
}
```

Case 1: marks = 85
Output: Pass

Explanation

Since the marks is greater than 80 i.e. the condition inside 'if' parentheses is true, we get "Pass" printed in the output.

Case 2: marks = 70
Output : No output

Explanation

Since the marks is not greater than 80 i.e. the condition inside if parentheses is false, the statement inside the if block is skipped/not executed i.e. we get nothing printed in the output.

Let us now look at the variation of if statement i.e. if-else.

Topic 2: If-else statement

Sometimes, we encounter situations where we have to choose between two options. For example, if it's Tuesday I'll eat a veg burger otherwise I'll eat a non-veg burger.
An if-else statement is designed to give this functionality in our code. It executes statements if some condition is true or false. If the condition is true, the if part is executed, otherwise the else part of the statement is executed.

Syntax

```
if (condition)

{

     statement - 1

} else

{

     statement - 2

}
```

Let us understand it with the help of the following example

To illustrate the working of an if-else statement, we can create a grading system. We'll assume that the score ranges between 0 to 100(both inclusive). A score above 33 gets a "Pass" verdict, otherwise it's "Fail".

To solve this problem, we can use an if-else statement. Lets see how !

```
if (score > 33)
    {
      cout << "Pass\n";
    }
else
   {
      cout << "Fail\n";
   }
```

Case 1: Score = 60
Output: Pass

Explanation

Since the score is more than 33 i.e. the condition inside 'if' parentheses is true, we get "Pass" printed.

Case 2: Score = 20
Output: Fail

Explanation

Since the score is not more than 33 i.e. the condition inside 'if' parentheses is false, we get "Fail" printed.

Try these:

1. Find if the input value is odd or even. If it's odd print "Odd", otherwise print "Even".

Note: Input value will be between 1 and 10^6.

2. Find if the input character is 'a' or not.

Note: Input characters Would be lowercase alphabets.

There is another variation of if statement. Let us explore.

Topic 3: If-else if statement

It works on the same principle as if-else statement,however,here we can have multiple conditions. If the condition inside the if block is true, then the code/ statement inside if block will be executed, if it is false, the control will move to the else-if block and will check if it is true and execute the statement in the else-if block. When the condition in else-if block is false, it will execute the statements in the final else block.

We can have multiple else-if blocks too.

Syntax

```
if (condition - 1)
{
    statement - 1
}
else if (condition - 2)
{
    statement - 2
}
else
{
    statement - 3
}
```

Let us try to solve the example below

To implement if else-if, we can further expand our grading system problem. Apart from pass and fail now it will give grades based on the score.

Grade	Score
A	80-100
B	60-80
C	40-60
D	<40

To solve this problem, we can use an if - else if - else statement to execute different actions depending upon the score:

```
if(score > 80)
{
	cout << “A\n”;
}
else if (score > 60)
{
	cout << “B\n”;
}
 else if (score > 40)
{
	cout << “C\n”;
}
else
{
	cout << “D\n”;
}
```

Case 1: Score = 81
Output : A
Explanation

Since the score is more than 80 i.e. the condition inside if parentheses is true “A” gets printed.

Case 2: Score = 61
Output : B
Explanation

Since the score is less than 80, the first block is skipped and since it is more than 60 i.e. the condition inside else-if parentheses is true, “B” gets printed.

Case 3: Score = 41
Output : C
Explanation

Since the score is 41, the first 2 blocks are skipped and then the condition for the third block is checked. It turns out that it is true, so “C” gets printed and the rest is skipped.

Case 4: Score = 21
Output : D

Explanation

Since all the conditions are false, the else-block will execute and we get “D” as output.

Try these

1. Write a program to identify people as “Child” (age < 12), “Teenager” (12 <= age < 18) or “Adult” (age >= 18).
2. Print the maximum of 3 numbers a, b, c taken as input.

Let us now increase the dimension and abilities of if and explore the concept of nested if-else statements.

Topic 4: Nested if-else

It is simply an if-else statement inside another if-else statement.

Syntax

```
if (condition - 1)
{
     if (condition - 2)
     {
           statement - 1
     }
     else
     {
          statement - 2
      }
}
else
{
          statement - 3
}
```

Consider the following scenario :

```
if (score > 33)
{
        if(marks > 80)
        {
                cout << “Gracefully ”;
        }
        cout << “Pass\n”;
}
else
{
       cout << “Fail\n”;
}
```

Note: Watch the curly braces { }

Input: Score=30

Output: Fail

Input : Score=90

Output : Gracefully pass

Input : Score=60

Output : Pass

Operators that assist in the functioning of conditional statements by proper and accurate condition checks are known as conditional operators. We have already seen a few in the previous lecture but it deserves a mention here too. So, let us learn more about them.

Topic 5: Conditional operators

These operators are used when a condition comprises of more than one boolean expression/ condition check. For instance, if we want to print a number only if it is greater than 2 and less than 5, then we will use conditional operators to combine the 2 expressions. We have 3 types of conditional operators - logical-and, logical-or and ternary operator.

a. Logical-and operator (&&)

It is used when we want the condition to be true if both the expressions are true.

Syntax

```
if(condition - 1 && condition - 2)
{
      statement;
}
```

Let us look at this example

Print the number if the input value is greater than 5 and less than 10.

```
if (val > 5 && val < 10)
{
      cout << val << '\n';
}
```

Case 1: val = 3
Output: No output
Explanation

The input value is less than 10 but it is not greater than 5.

Case 2: val = 7
Output: 7
Explanation

The input value is both less than 10 and greater than 5.

Case 3: val = 13
Output: No output
Explanation

The input value is greater than 5 but it is not less than 10.

Common misconception: '&' v/s '&&'

& -> compares bitwise
&& -> logical and

The first one (&) parses through all the conditions, despite their evaluation as true or false. However, the latter traverses through the second condition only if the first condition is evaluated as true, otherwise it negates the entire condition. For instance,

```
cout << (false & 5/0==2);
```

It gives us a runtime error since 5 is divided by 0, which isn't a valid operation. However, if we write-

```
cout << (false && 5/0==2);
```

we get "false" as the output. This is because our first condition is false, which is enough to make the entire condition false here.

Tip: Always use parentheses around every condition.

Try this:

Write a program to print the value of input if it is even and divisible by 3.

b. Logical-or operator (||)

This operator is used when any one of the boolean expressions is evaluated as true.

Syntax

```
if(condition - 1 || condition - 2)
{
      statement;
}
```

Let us look at this example:

Print the number if the input value is greater than 10 or less than 5.

Code

```
if (val < 5 || val > 10)
{
      cout << val;
}
```

Case 1: val = 3
Output: 3
Explanation

The input value is less than 5. This is enough to satisfy the condition so the second condition won't be tested and val will be printed.

Case 2 : val = 7
Output: No output
Explanation

Both the conditions are evaluated as false.

Case 3: val = 13
Output: 13
Explanation

The input value is not smaller than 5 but it is greater than 10. So, all in all, it will be evaluated as true.

Try this

Write a program to print the value of input if it is divisible by 3 or 5.

Common misconception: '|' v/s '||'

| -> compares bitwise
|| -> logical or

The first one parses through all the conditions, despite their evaluation as true or false. However, the latter traverses through the second condition only if the first condition is evaluated as false, otherwise it validates the entire condition. For instance,

```
cout << (true | 5/0==2);
```

It gives a runtime error since 5 is being divided by 0, which isn't a valid operation. However, if we write-

```
cout << (true || 5/0==2);
```

We get "true" as the output. This is because our first condition is true, which is enough to make the entire condition true in case of logical or.

Ternary operator (?:)

It is a smaller version for the if-else statement. If the condition is true then statement - 1 is executed else the statement - 2 is executed.

Syntax

```
condition ? statement - 1 : statement - 2;
```

Let us have a look at this example:

Without ternary operator

```
if (val % 2 == 1)
{
      cout << “Value entered is odd\n”;
}
else
{
      cout << “Value entered is even\n”;
}
```

With ternary operator

```
val % 2 == 1 ? cout << “Value entered is odd\n”; : cout << “Value entered is even\n”;
```

Case 1: val = 1
Output (without ternary operator) - Value entered is odd
Output (with ternary operator) - Value entered is odd

Case 2: val = 2
Output (without ternary operator) - Value entered is odd
Output (with ternary operator) - Value entered is odd

Try these

1. Write a short program that gives the following as output -
For each multiple of 3, print "Fizz" instead of the number.
For each multiple of 5, print "Buzz" instead of the number.
For numbers which are multiples of both 3 and 5, print "FizzBuzz" instead of the number.
Otherwise print the number itself.

2. Write a short program that prints each number from 1 to 100 on a new line, except if the number is a multiple of 5 or 7.

We have now reached to the final conditional statement viz switch case. This renders the program to look extremely neat and organized. Let us learn more about it.

Topic 6: Switch statement

Assume we have a variable and we want to do multiple operations on it based upon the value it stores. In such cases the switch statements are the most convenient and useful concept.

It is like an if-else ladder with multiple conditions, where we check for equality of a variable with different values.

It works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long.

Syntax

```
switch (expression)
{
case x:
      // code
      break;
case y:
      // code
      break;
.
.
.
default:
      // code
}
```

Note: The case value must be literal or constant, and must be unique.

Let us look at this example:

Write a program using switch statements to check if the input lowercase character is vowel or consonant.

Code

```
switch (ch) {
case ‘a’:
      cout << “Vowel\n”;
      break;
case ‘e’:
      cout << “Vowel\n”;
      break;
case ‘i’:
      cout << “Vowel\n”;
      break;
case ‘o’:
      cout << “Vowel\n”;
      break;
case ‘u’:
      cout << “Vowel\n”;
      break;
default:
      cout << “Consonant\n”;
}
```

Case 1: ch = ‘e’
Output - Vowel

Case 2: ch = ‘w’
Output - Consonant

‘Break’ is a very important keyword when we are implementing switch statement (we will explore it in detail in the forthcoming lectures). Lets look at an example where we implement a logic with and without break statement.

Case 1: Without using Break Statement

```
int val = 5;


switch (val)
{
case (5):
      cout << "Five\n";
case (6):
      cout << "Six\n";
case (7):
      cout << "Seven\n";
}
```

Output:

Five
Six
Seven

Explanation

Surprised? You did not expect this, did you? Let us see why ?
Here, even when the first case is true, it will keep on executing all the subsequent statements until it encounters a break statement(which we have not used anywhere). After the break statement, it will break out of the switch statement.

To get the expected result, lets modify the program a little.

Case 2: Using Break Statement

```
int val = 6;


switch (val)
{
case (6):
      cout << "Six\n";
      break;
case (7):
      cout << "Seven\n";
      break;
case (8):
      cout << "Eight\n";
      break;
}
```

Output:
Six

Explanation
Using the break keyword here, helped the program to stop executing and come out of the switch Statement.

Try this
Write a program to print the day name based upon the day number.

1 - Monday, 2 - Tuesday, and so on.

Upcoming Class Teasers

- while loop
- The for loop
- The do-while loop
- Break keyword
- Continue keyword