CONDITIONAL AND CONTROL STATEMENT

C++ Conditional Statements: IF, ELSE IF, ELSE

Conditional statements allow us to build programs that can make decisions based on a set of parameters. With the help of conditional statements, you can write code that executes an action depending on whether a specific condition evaluates to true or false.

Similar to other programming languages, C++ also offers the basic conditional statements; if, else if and else. Note that every conditional statement operates on an expression that should evaluate to a Boolean result; true or false. Failure to comply with this will obviously result in an error.

Without further ado, let's take a look at examples of C++ conditional statements.

IF Statement

We use **if** statement in the scenarios where a certain block of code needs to be executed if and only if a given condition evaluates to true. The basic syntax of if statement in C++ is as follows:

```
if (your_condition)
{
  // the code chunk which will execute if your_condition is true.
}
```

Let's look at an example to understand it.

```
if (55 > 45)
{
  cout << "I am in the <if> block of code" << endl;
}
```

In the above example, since 55 is greater than 45, the code will print I am in the <if> block of code as the output.

If the expression above is written as if (55 < 45), the code will not output anything because the expression evaluates to false.

Note: we can use multiple if statements one after another. Each statement will run independently.

```
if (0 < 10)

{

  cout << "I will study." << endl;

}

if (20 < 10)

{

  cout << "I will party." << endl;

}

if (30 > 20)

{

  cout << "I will go for a fine dine." << endl;

}
```

The above code running in sequence will result in the following statements as output.

I will study.

I will go for a fine diner.

The statement I will party. is not printed because its expression evaluates to false.

Ever heard of nested code segments? We can also use if statements as a nested code block under another code block. Basically, an if statement within another if statement.

ELSE IF Statement

The **else if** statement is a kind of if statement but with optional execution. The else if statement only runs when the preceding conditional statements evaluate to false.

```
if (55 < 45)

{

  cout << "I am in the <if> block of code." << endl;
```

```
}

else if (55 > 45)

{

  cout << "I am in the <else if> block of code." << endl;

}
```

The above code snippet will print I am in the <else if> block of code. Only because the preceding if statement evaluates to false. If the preceding if statement had already evaluated to true, the else if code chunk would not have been executed.

Just like multiple if statements, you can use multiple else if statements in sequence as well. Also, note that else if statement cannot be written stand-alone. It must only be written when there exists at least one preceding if statement.

ELSE Statement

The **else** statement is a resting statement and executes only when all the preceding if and else if statements evaluate to false.

```
if (55 < 45)

{

  cout << "I am in the <if> block of code." << endl;

}

else if (55 == 45)

{

  cout << "I am in the <else if> block of code." << endl;

}

else

{

  cout << "I am in the <else> block of code." << endl;

}
```

The above code prints I am in the <else> block of code. As output. Just like else if statements, the else statements cannot be written stand-alone.

Control Statements

A C++ control statement redirects the flow of a program in order to execute additional code. These statements come in the form of conditionals (if-else, switch) and loops (for, while, do-while). Each of them relies on a logical condition that evaluates to a boolean value in order to run one piece of code over another.

Boolean Logic

Boolean values on their own do not appear to be very useful, however when combined with control statements they become one of the most powerful tools that any programing language has to offer. As stated in the data types notes, boolean variables (bool) can be either true or false. There are also binary operators that return a boolean value depending on specific conditions. Some operators compare, such as ==, !=, >, etc., the values on either side of it and return true if the entire statement is true. Others, work with boolean values to return another boolean, such as && and ||.

Operator	Symbol	Usage	Example
Equals	==	Returns true if the statements on either side of the operator are the same value, otherwise it returns false. Can be used with int, double/float, char, boolean, and string.	x == 'A'
Not Equal	!=	Returns true if the statements on either side of the operator are not the same, otherwise it returns false.	x != 2
Greater Than	>	Returns true if the first term is greater than the second, otherwise it returns false. Note: If both terms are equal it returns false.	x > 0
Less Than	<	Returns true if the first term is less than the second, otherwise it returns false.	x < 10
Greater Than or Equal	>=	Returns true if the first term is greater than or equal to the second, otherwise it returns false.	x >= 0
Less Than or Equal	<=	Returns true if the first term is less than or equal to the second, otherwise it returns false.	x <= -2
And	&&	Returns true if both terms on either side of the operator are true, otherwise it returns false. It is commonly used to concatenate statements together.	x && True (x == 2) && (y >= 0)
Or	||	Returns true if any of the terms on either side of the operator are true, otherwise it returns false.	x || y

Not	!	Negates the value of the statement that follows it.	!(x > 2)

If-Else

If-Else statements allow the program to execute different blocks of code depending on conditionals. All If statements have the following form:

```
if ( condition ) {
   //body
}
```

An If statement executes the code in the body section if the condition evaluates to True, otherwise it skips the body. When the condition evaluates to False, the program will either test another condition with a following Else-if, run the code inside an Else statement, or continue as normal if neither an Else-if nor Else block exist.

An Else statement acts as a default case for If statements. In the case that an If is directly followed by an Else and the condition is false, the code in the Else is executed. Else statements do not have a condition themselves.

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

An If statement can followed by any number of Else-if blocks. Each Else-if has its own conditional statement and body of code. If an Else-if conditional is False, then its body of code is skipped and the program will check the next Else-if in order that they appear. An If followed by a long list of Else-IF's is usually referred to as an If-Else ladder.

```
if ( condition ) {
   //body
} else if ( 2nd condition ) {
   // else if body
} else {
   // else body
}
```

For Loop

A for loop allows for a block of code to be executed until a conditional becomes false. For loops are usually used when a block of code needs to executed a fixed number of times. Each loop consists of 3 parts, an initialization step, the conditional, and an iteration step. The initialization is run before entering the loop, the condition is checked at the beginning of each run through the loop (including the first run), and the iteration step executes at the end of each pass through the loop, but before the condition is rechecked. It is usual practice to have the iteration step move the loop on step closer to making the condition false, thus ending the loop, but this does not need to be the case.

```
for( initialization ; conditional ; iteration ) {
   // loop body
}
```

While Loop

A while loop is a simple loop that will run the same code over and over as long as a given conditional is true. The condition is checked at the beginning of each run through the loop (including the first one). If the conditional is false for the beginning, the while loop will be skipped all together.

```
while ( conditional ) {
   // loop body
}
```

Do-while Loop

A do-while loop acts just like a while loop, except the condition is checked at the end of each pass through the loop body. This means a do-while loop will execute at least once.

```
do {
   // loop body
} while ( condition );
```

Break

Break is a useful keyword that allows the program to exit a loop or switch statement before the expected end of a that code block. This is useful in error checking or if the outcome of a loop is not certain. For example, the following code will break out of the for loop if a user asks to leave.

```
string inputs[10];
string input;
```

```
for (int i = 0; i < 10; i++ ) {
    cin >> input;
    if ( input == "quit" ) {
        break;
    }
    inputs[i] = input;
}
```