Lecture 3: Jump statements in C++

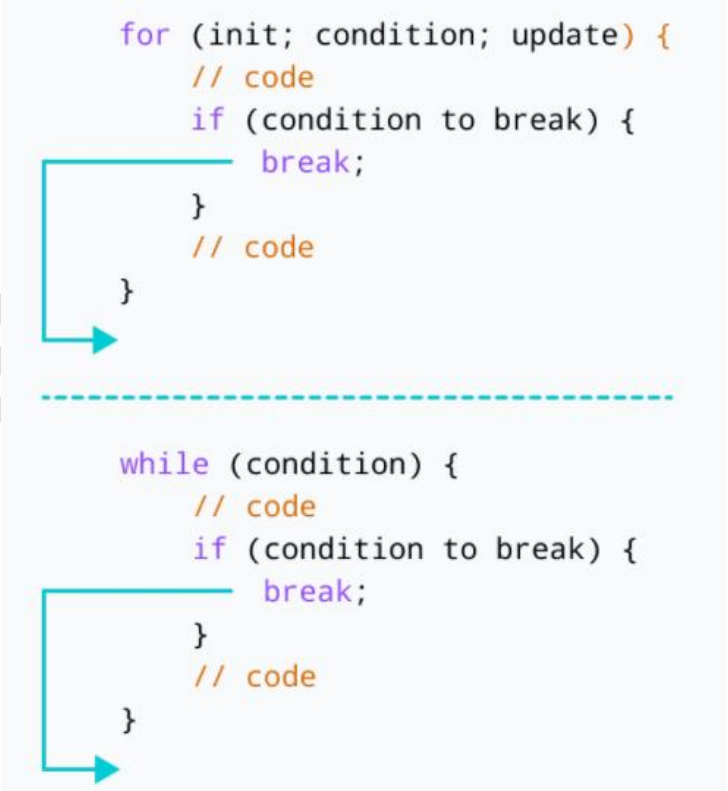
Jump statements are used to manipulate the flow of the program if some conditions are met. It is used to terminate or continue the loop inside a program or to stop the execution of a function. In C++ there is four jump statement: **break**, **continue**, **goto** and **return**.

C++ Break

In C++, the **break** statement terminates the loop when it is encountered.

Syntax

```
break;
```



Example: break with for loop

```
// program to print the value of i

#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // break condition
        if (i == 3) {
            break;
        }
        cout << i << endl;
    }

return 0;
}
```

Output

```
1
2
```

In the above program, the for loop is used to print the value of i in each iteration.

```
if (i == 3) {
    break;
}
```

This means, when i is equal to 3, the break statement terminates the loop. Hence, the output doesn't include values greater than or equal to 3.

> Note: The break statement is usually used with decision-making statements.

Example: break with while loop

// program to find the sum of positive numbers
// if the user enters a negative numbers, break ends the loop
// the negative number entered is not added to sum

```
// program to find the sum of positive numbers
// if the user enters a negative numbers, break ends the loop
// the negative number entered is not added to sum

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    while (true) {
        // take input from the user
        cout << "Enter a number: ";
        cin >> number;

        // break condition
        if (number < 0) {
            break;
        }

        // add all positive numbers
        sum += number;
    }

    // display the sum
    cout << "The sum is " << sum << endl;

    return 0;
}
```

Output

```
Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: -5
The sum is 6.
```

In the above program, the user enters a number. The while loop is used to print the total sum of numbers entered by the user. Here, notice the code,

```
if(number < 0) {
    break;
}
```

This means, when the user enters a negative number, the break statement terminates the loop and codes outside the loop are executed.

The while loop continues until the user enters a negative number.

break with Nested loop

When break is used with nested loops, break terminates the inner loop. For example,

```
// using break statement inside
// nested for loop

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // nested for loops

    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (i == 2) {
                break;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

    return 0;
}
```

Output

```
i = 1, j = 1
i = 1, j = 2
```

```
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3
```

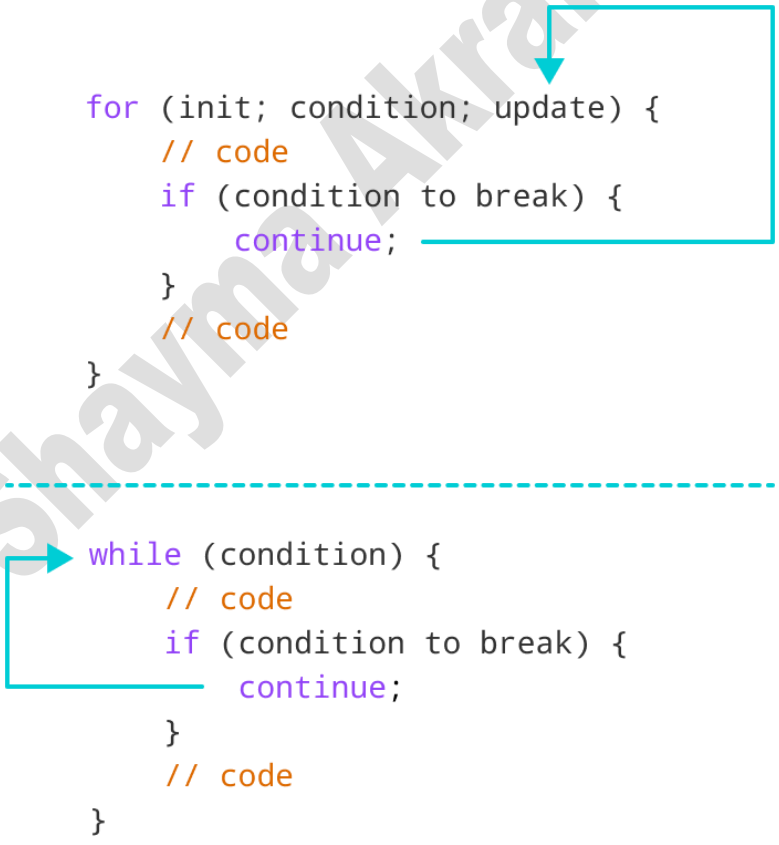
In the above program, the break statement is executed when i == 2. It terminates the inner loop, and the control flow of the program moves to the outer loop. Hence, the value of i = 2 is never displayed in the output.

C++ Continue

In computer programming, the continue statement is used to **skip the current iteration** of the loop and the control of the program goes to the next iteration.

Syntax

```
continue;
```



Example: continue with for loop

In a for loop, continue skips the current iteration and the control flow jumps to the update expression.

```
// program to print the value of i

#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // condition to continue
        if (i == 3) {
            continue;
        }

        cout << i << endl;
    }

    return 0;
}
```

Output

```
1
2
4
5
```

In the above program, we have used the the for loop to print the value of i in each iteration. Here, notice the code,

```
if (i == 3) {
    continue;
}
```

This means

- When i is equal to 3, the continue statement skips the current iteration and starts the next iteration
- Then, i becomes 4, and the condition is evaluated again.
- Hence, 4 and 5 are printed in the next two iterations.

Example: continue with while loop

In a while loop, continue skips the current iteration and control flow of the program jumps back to the while condition.

```cpp
// program to calculate positive numbers till 50 only
// if the user enters a negative number,
// that number is skipped from the calculation

// negative number -> loop terminate
// numbers above 50 -> skip iteration

#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    int number = 0;

    while (number >= 0) {
        // add all positive numbers
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;

        // continue condition
        if (number > 50) {
            cout << "The number is greater than 50 and won't be
calculated." << endl;
            number = 0;  // the value of number is made 0 again
            continue;
        }
    }

    // display the sum
    cout << "The sum is " << sum << endl;

    return 0;
}
```

Output

```
Enter a number: 12
Enter a number: 0
Enter a number: 2
Enter a number: 30
Enter a number: 50
Enter a number: 56
The number is greater than 50 and won't be calculated.
Enter a number: 5
Enter a number: -3
The sum is 99
```

In the above program, the user enters a number. The while loop is used to print the total sum of positive numbers entered by the user, as long as the numbers entered are not greater than 50.

Notice the use of the continue statement.

```
 if (number > 50){
    continue;
}
```

When the user enters a number greater than 50, the continue statement skips the current iteration. Then the control flow of the program goes to the condition of while loop.

- When the user enters a number less than 0, the loop terminates.

> **Note**:
>
> The continue statement works in the same way for the do...while loops.

continue statement with Nested loop

When **continue** is used with nested loops, it skips the current iteration of the inner loop. For example,

```
// using continue statement inside
// nested for loop

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // nested for loops

    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (j == 2) {
                continue;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
```

```
    }

    return 0;
}
```

Output

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

In the above program, when the continue statement executes, it skips the current iteration in the inner loop. And the control of the program moves to the update expression of the inner loop.

Hence, the value of j = 2 is never displayed in the output.

> **Note**:
>
> The **break statement** terminates the loop entirely. However,
>
> the continue statement only skips the current iteration.

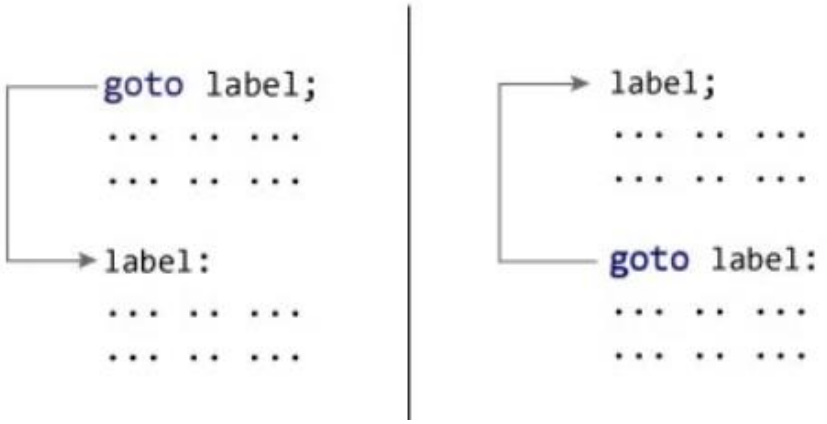
C++ goto Statement

In C++ programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto Statement

```
goto label;
... .. ...
... .. ...
... .. ...
label:
statement;
... .. ...
```

In the syntax above, label is an identifier. When goto label; is encountered, the control of program jumps to label: and executes the code below it.



Example: goto Statement

```
// This program calculates the average of numbers entered by the
user.
// If the user enters a negative number, it ignores the number and
// calculates the average number entered before it.

# include <iostream>
using namespace std;

int main()
{
    float num, average, sum = 0.0;
    int i, n;

    cout << "Maximum number of inputs: ";
    cin >> n;

    for(i = 1; i <= n; ++i)
    {
        cout << "Enter n" << i << ": ";
        cin >> num;

        if(num < 0.0)
        {
           // Control of the program move to jump:
            goto jump;
        }
        sum += num;
    }

jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
```

```
}
```

Output

```
Maximum number of inputs: 10
Enter n1: 2.3
Enter n2: 5.6
Enter n3: -5.6

Average = 3.95
```

Note:

You can write any C++ program without the use of **goto** statement and is generally considered a good idea not to use it.

Reason to Avoid goto Statement

The goto statement gives the power to jump to any part of a program but, makes the logic of the program complex and tangled.

In modern programming, the goto statement is considered a harmful construct and a bad programming practice.

The goto statement can be replaced in most of C++ program with the use of break and continue statements.

C++ return statement

The **return** statement returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and return the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value is must be returned.

Syntax:

```
return[expression];
```

```cpp
// C++ code to show not using return
// statement in void return type function

#include <iostream>
using namespace std;

// void method
void Print()
{
    cout << "Hello World";
}

// Driver method
int main()
{

    // Calling print
    Print();

    return 0;
}
```

Output:

```
Hello World"
```

Using return statement in void return type function:

Now the question arises, what if there is a return statement inside a void return type function?

Since we know that, if there is a void return type in the function definition, then there will be no return statement inside that function. But if there is a return statement inside it, then also there will be no problem if the syntax of it will be:

Correct Syntax:

```cpp
void func()
{
return;
}
```

This syntax is used in function just as a jump statement in order to break the flow of the function and jump out of it.

```
// C++ code to show not using return
// statement in void return type function

#include <iostream>
using namespace std;

// void method
void Print()
{
    cout << "Hello World";

    return; // void method using the return statement
}

// Driver method
int main()
{

    // Calling print
    Print();

    return 0;
}
```

But if the return statement tries to return a value in a void return type function, that will lead to errors.

Incorrect Syntax:

```
void func()
{
    return value;
}
```

Warnings:
warning: 'return' with a value, in function returning void

```
// statement in void return type function

#include <iostream>
using namespace std;

// void method
void Print()
{
    cout << "Hello World";

// void method using the return
// statement to return a value

    return 10;
}

// Driver method
int main()
{

    // Calling print
    Print();

    return 0;
}
```

Warnings:

```
prog.c: In function 'Print':
prog.c:12:9: warning: 'return' with a value, in function returning void
  return 10;
         ^
```