CS135; Control Structures II (Repetition)

Gael Zarco

March 8, 2025

C++ has three repetition, or looping, structures that allow you to repeat a set of statement until certain conditions are men.

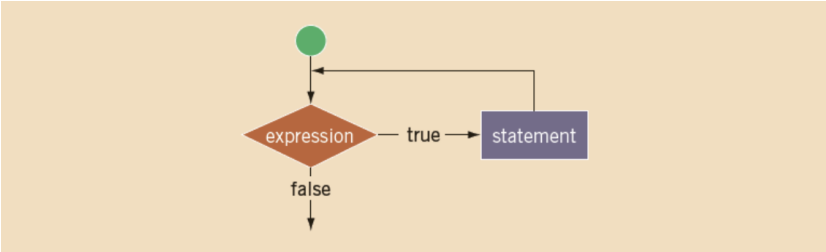
1 `while` Looping (Repetition) Structure

Listing 1: `while` Loop Syntax

```
while (expression)
        statement;
```

Statement can be simple or compound statement.

Execution Flow:



`expression` acts as a **Decision Maker** and is usually a logical expression.

- Provides an entry condition to the loop.
- If initially `true`, loop starts.
- Re-evaluates on each iteration, continues if `true`.
- Exits loop when `false`.

Infinite Loops are loops that execute endlessly (no exit case).

`statement` is the body of the loop.

Listing 2: `while` Loop Example Program

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    int calBurnedInADay;
7    int calBurnedInAWeek;
8    int day;
9
10   day = 1;
11   calBurnedInAWeek = 0;
12
13   while (day <= 7) {
14     cout << "Enter calories burned day " << day << ": ";
15     cin >> calBurnedInADay;
16     cout << endl;
17     calBurnedInAWeek = calBurnedInAWeek + calBurnedInADay;
18     day = day + 1;
19   }
20
21   cout << "Average number of calories burned each day: "
22   << calBurnedInAWeek / 7 << endl;
23
24   return 0;
25 }
```

`day` in the example program above is known as the **Loop Control Variable (LCV)**.

Loop Control Variables:

1. Must be initialized before the `while` loop.
2. Must result in a `false` expression evaluation at a certain point.

Be careful of where in the loop, LCVs are manipulated. This can change the behavior of the loop.

1.1 Case 1: Counter-Controlled `while` Loops

Counter-Controlled `while` Loop is a loop in which the LCV serves as a *counter* that dictates how many times the loop iterates or executes.

- That is, `statement` is executed n times.

Listing 3: Counter-Controlled `while` Loop Example

```
1  counter = 0;
2
```

```
3  while (counter < N) {
4    // Code that does something...
5    counter++
6  }
```

1.2 Case 2: Sentinel-Controlled `while` Loops

Last entry in a set of data is known as the **sentinel** and this value tells the loop to stop.

- if `item != sentinel` → `statement`.

The `while` loop continues to execute as long as the program has not read the `sentinel`.

Such a loop is known as a **Sentinel-Controlled `while` Loop**.

Listing 4: Sentinel-Controlled `while` Loop Syntax

```
1  cin >> variable;
2
3  while (variable != sentinel) {
4    // Code that does something...
5    cin >> variable;
6  }
```

1.3 Case 3: Flag-Controlled `while` Loops

Flag-Controlled `while` Loops use a `bool` variable to control the loop.

A **Flag Variable** is a `bool` variable that indicates whether a condition is `true` or `false`.

- Generally named for the `true` state of the condition. *i.e. isFound, isTallEnough, etc.*

Listing 5: Flag-Controlled `while` Loop Example

```
1  isFound = false;
2
3  while (!isFound) {
4    // Code that does something...
5    if (expression)
6      isFound = true;
7    }
8  }
```

1.4 Case 4: EOF-Controlled `while` Loops

Until now, input stream variables such as `cin` and the extraction operator, `>>`, have been used to read and store data in variables. However, The input stream variable can also return a value after reading data, as follows:

1. If the program has reached the end of the input data, the input stream var returns `false`.
2. If the program reads any faulty data, the input stream enters the fail state. Because of this, any attempt to use that stream is ignored. In this case, the input stream var returns `false`.
3. In any other case besides (1) and (2), the input stream var returns `true`.

EOF-Controlled `while` Loops are best used at times where selecting a good sentinel value is difficult or for programs where the data-file is altered.

- Someone could potentially erase the sentinel value or add past the value.

Listing 6: EOF-Controlled `while` Loop Example

```
1 cin >> variable;
2
3 while (cin) {
4   // Code that does something...
5   cin >> variable;
6 }
```

1.5 `eof` Function

The `eof` function allows you to check if you have reached the end-of-file status on an input stream variable.

- Member of the data type `istream`.
- Logical expression; returns a boolean value.

Listing 7: `eof` Function Syntax

```
istreamVar.eof()
```

istreamVar is an input variable stream, such as cin.

Listing 8: `eof` Example

```
1 ifstream infile;
2 char ch;
3
4 infile.open("inputDat.dat");
```

```
5
6  infile.get(ch);
7
8  while(!infile.eof()) {
9    cout << ch;
10   infile.get(ch);
11 }
```

2 `for` Looping (Repetition) Structure

A counter-focused looping structure.

Listing 9: `for` Loop Syntax

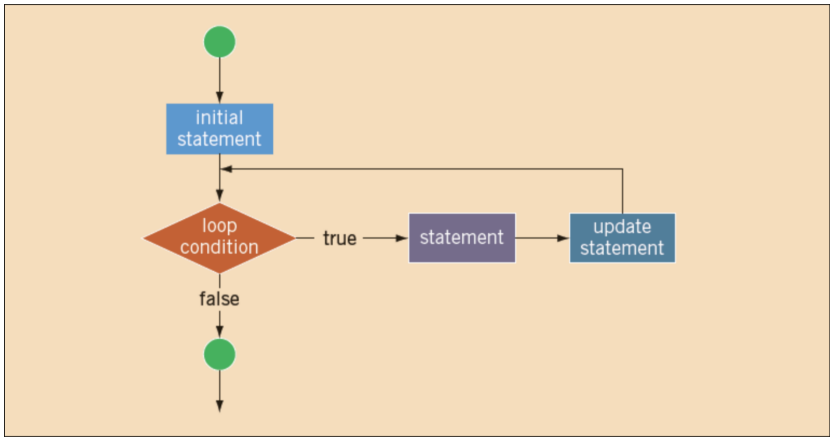
```
for(initial statement; loop condition; update statement)
  statement;
```

Statement can be a simple or compound statement.

The `initial statement`, `loop condition`, and `update statement` control the `statement` of the `for` loop.

- These are known as the `for` loop control statements.

Execution Flow:



The `for` loop executes as follows:

1. The `initial statement` executes.
2. The `loop condition` is evaluated. If the `loop condition` evaluates to `true`:
   - Execute the `for` loop statement.

- Execute the `update statement`.

3. Repeat *Step 2* until the `loop condition` evaluates to `false`.

The `initial statement` usually initializes a variable (called the `for` **loop control**, or `for` **indexed variable**).

Caveats:

- If the `loop condition` is initially `false`, the loop body does not execute.
- The `for` loop body executes indefinitely if the `loop condition` is always `true`.
- You can omit all three statements; however, the `for` statement must contain two semicolons *(Listing 10)*.

Listing 10: `for` Loop Gotcha

```
for (;;)
  cout << "Hello" << endl;
```

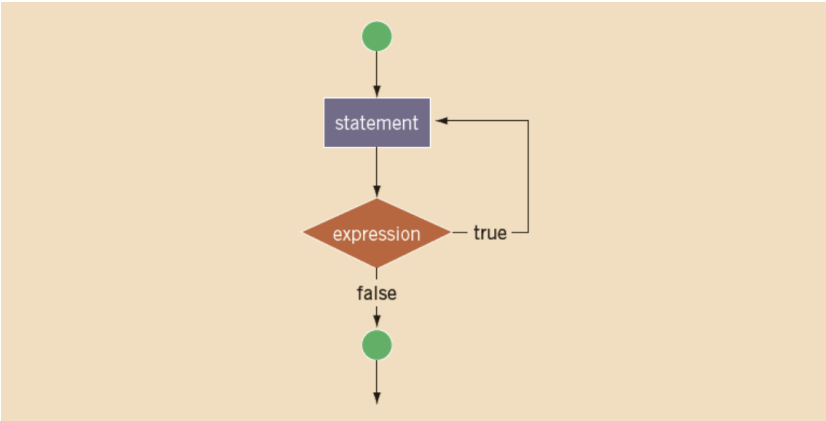
3 `do..while` Looping (Repetition) Structure

Listing 11: `do..while` Loop Syntax

```
do
  statement
while(expression);
```

Statement can be a simple or compound statement.

Execution Flow:



The `statement` executes first, then the `expression` is evaluated.

- As long as the `expression` evaluates to `true`, the loop continues.

In a `while` and `for` loop, the `loop condition` is evaluated *before* executing the body of the loop.

- `while` and `for` loops are called **Pretest Loops**.
- `do...while` loops are called **Posttest Loops**.

3.1 Choosing the Correct Looping Structure

- If you know the number of repetitions needed → `for` loop.
- If you do not know and it **could be 0** → `while` loop.
- If you do not know and it is **at least 1** → `do...while` loop.

4 `break` and `continue` Statements

The `break` statement is typically used for two purposes:

1. To exit early from a loop.
2. To skip the remainder of the `switch` structure.

After the `break` statement executes, the program continues to execute with the first statement after the structure.

The `continue` statement is used in all 3 loop structures:

1. `while`
2. `for`
3. `do...while`

When the `continue` statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop.

Differences between loops:

- `do..while` and `while` → `expression` is evaluated immediately after `continue` statement.
- `for` → `update statement` is executed after `continue` statement → `loop condition` executes.