Loops in C++

Dr. Usman Habib

Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - The name of a control variable (or loop counter).
  - The initial value of the control variable.
  - The condition that tests for the final value of the control variable (i.e., whether looping should continue).
  - The increment (or decrement) by which the control variable is modified each time through the loop.
- Example:

```
int counter =1;                //initialization
while (counter <= 10){         //repetition
condition
    cout << counter << endl;
    ++counter;                 //increment
}
```

Essentials of Counter-Controlled Repetition

- The declaration

```
int counter = 1;
```

  - Names **`counter`**
  - Declares **`counter`** to be an integer
  - Reserves space for **`counter`** in memory
  - Sets **`counter`** to an initial value of **`1`**

The for Repetition Structure

- The general format when using **`for`** loops is

```
for ( initialization; LoopContinuationTest;
      increment )
   statement
```

- Example:

```
for( int counter = 1; counter <= 10; counter++ )
   cout << counter << endl;
```

  – Prints the integers from one to ten

No semicolon after last statement

The for Repetition Structure

- **`For`** loops can usually be rewritten as **`while`** loops:

```
initialization;
while ( loopContinuationTest){
    statement
    increment;
}
```

- Initialization and increment as comma-separated lists

```
for (int i = 0, j = 0;  j + i <= 10;
  j++, i++)
    cout << j + i << endl;
```

Examples Using the for Structure

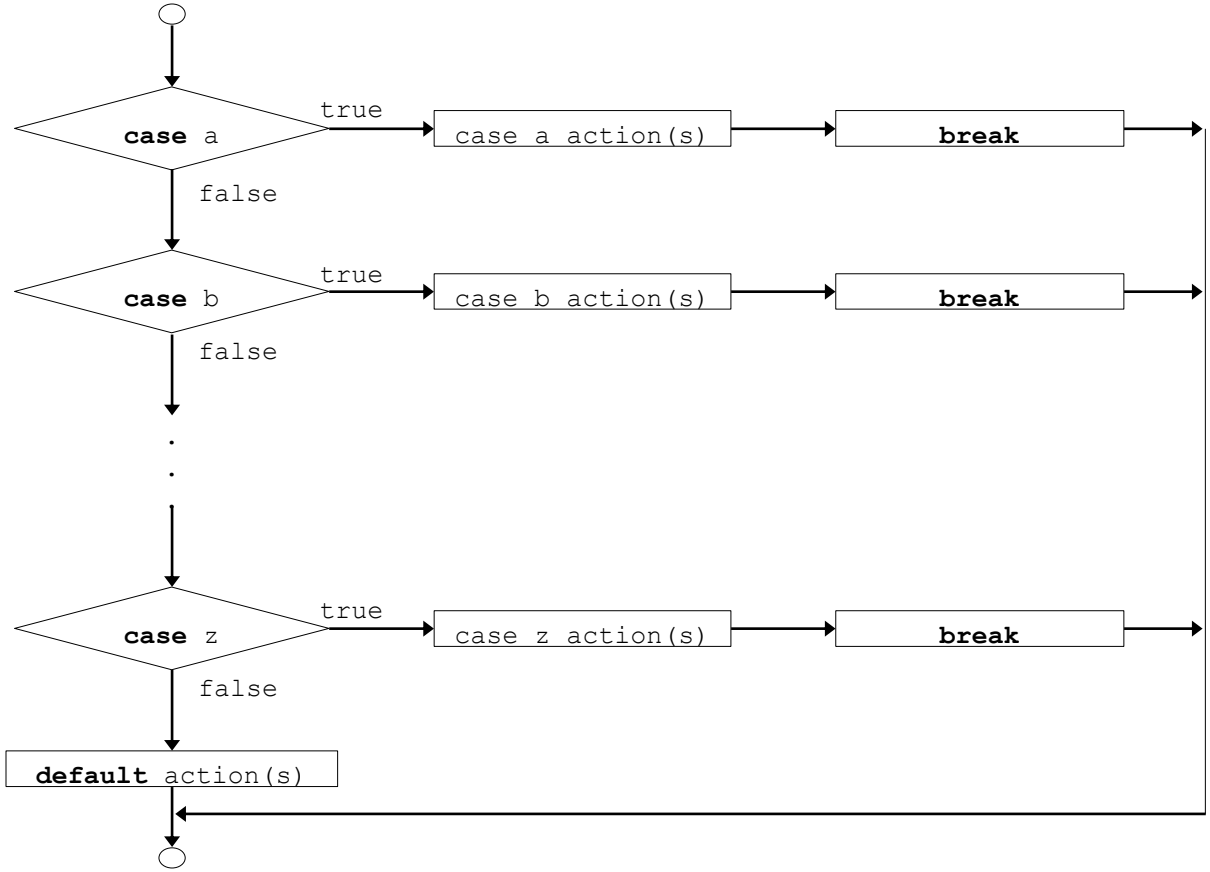
- Program to sum the even numbers from 2 to 100

```
// Fig. 2.20: fig02_20.cpp
// Summation with for
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int sum = 0;

   for ( int number = 2; number <= 100; number +=
      sum += number;

   cout << "Sum is " << sum << endl;

   return 0;
}
```

```
Sum is 2550
```

The switch Multiple-Selection Structure

- **`switch`**
  - Useful when variable or expression is tested for multiple values
  - Consists of a series of **`case`** labels and an optional **`default`** case



```cpp
// Fig. 2.22: fig02_22.cpp
// Counting letter grades
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int grade,        // one grade
       aCount = 0,   // number of A's
       bCount = 0,   // number of B's
       cCount = 0,   // number of C's
       dCount = 0,   // number of D's
       fCount = 0;   // number of F's

   cout << "Enter the letter grades." << endl
        << "Enter the EOF character to end input." << endl;

   while ( ( grade = cin.get() ) != -1 ) {

      switch ( grade ) {

         case 'A':  // grade was uppercase A
         case 'a':  // or lowercase a
            ++aCount;
            break;  // necessary to exit switch

         case 'B':  // grade was uppercase B
         case 'b':  // or lowercase b
            ++bCount;
            break;

```

Notice how the **case** statement is used

```
35          case 'C':  // grade was uppercase C
36          case 'c':  // or lowercase c
37             ++cCount;
38             break;
39
40          case 'D':  // grade was upper
41          case 'd':  // or lowercase d
42             ++dCount;
43             break;
44
45          case 'F':  // grade was upper
46          case 'f':  // or lowercase f
47             ++fCount;
48             break;
49
50          case '\n': // ignore newlines,
51          case '\t': // tabs,
52          case ' ':  // and spaces in
53             break;
54
55          default:   // catch all other characters
56             cout << "Incorrect letter grade entered."
57                  << " Enter a new grade." << endl;
58             break;  // optional
59       }
60    }
61
62    cout << "\n\nTotals for each letter grade are:"
63         << "\nA: " << aCount
64         << "\nB: " << bCount
65         << "\nC: " << cCount
66         << "\nD: " << dCount
67         << "\nF: " << fCount << endl;
68
69    return 0;
70 }
```

`break` causes **`switch`** to end and the program continues with the first statement after the **`switch`** structure.

Notice the **`default`** statement.

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

The do/while Repetition Structure

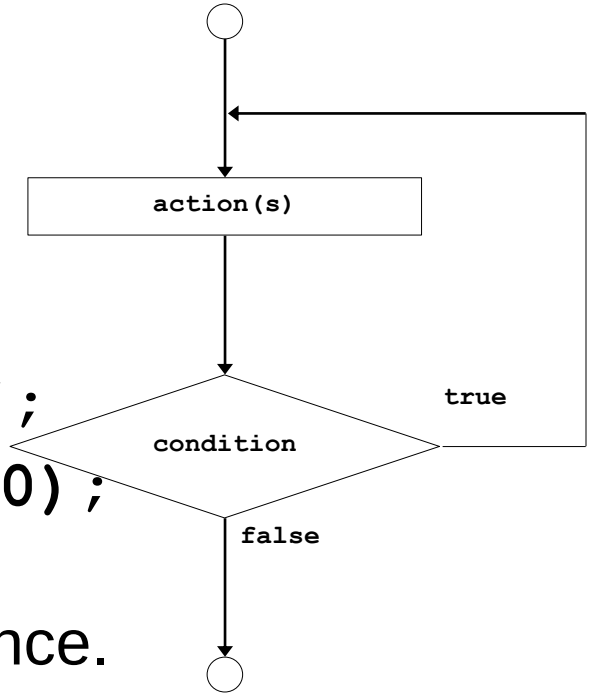
- The **`do/while`** repetition structure is similar to the **`while`** structure,
  - Condition for repetition tested after the body of the loop is executed
- Format:

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

- Example (letting counter = 1):

```
do {
  cout << counter << " ";
} while (++counter <= 10);
```

  - This prints the integers from **`1`** to **`10`**
- All actions are performed at least once.



The break and continue Statements

- **`Break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do/while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement:
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure

The break and continue Statements

- **`Continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure and proceeds with the next iteration of the loop
  - In **`while`** and **`do/while`**, the loop-continuation test is evaluated immediately after the **`continue`** statement is executed
  - In the **`for`** structure, the increment expression is executed, then the loop-continuation test is evaluated

Logical Operators

- **`&&`** (logical **`AND`**)
  - Returns **`true`** if both conditions are **`true`**
- **`||`** (logical **`OR`**)
  - Returns **`true`** if either of its conditions are **`true`**
- **`!`** (logical **`NOT`**, logical negation)
  - Reverses the truth/falsity of its condition
  - Returns **`true`** when its condition is **`false`**
  - Is a unary operator, only takes one condition
- Logical operators used as conditions in loops

Expression	Result
`true && false`	`false`
`true || false`	`true`
`!false`	`true`

Confusing Equality (==) and Assignment (=) Operators

- These errors are damaging because they do not ordinarily cause syntax errors.
  - Recall that any expression that produces a value can be used in control structures. Nonzero values are **`true`**, and zero values are **`false`**
- Example:

```
if ( payCode == 4 )
    cout << "You get a bonus!" << endl;
```

  - Checks the paycode, and if it is **`4`** then a bonus is awarded
- If **`==`** was replaced with **`=`**

```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

  - Sets **`paycode`** to **`4`**
  - **`4`** is nonzero, so the expression is **`true`** and a bonus is awarded, regardless of **`paycode`**.

Confusing Equality (==) and Assignment (=) Operators

- Lvalues
  - Expressions that can appear on the left side of an equation
  - Their values can be changed
  - Variable names are a common example (as in `x = 4;`)
- Rvalues
  - Expressions that can only appear on the right side of an equation
  - Constants, such as numbers (i.e. you cannot write `4 = x;`)
- Lvalues can be used as rvalues, but not vice versa

Structured-Programming Summary

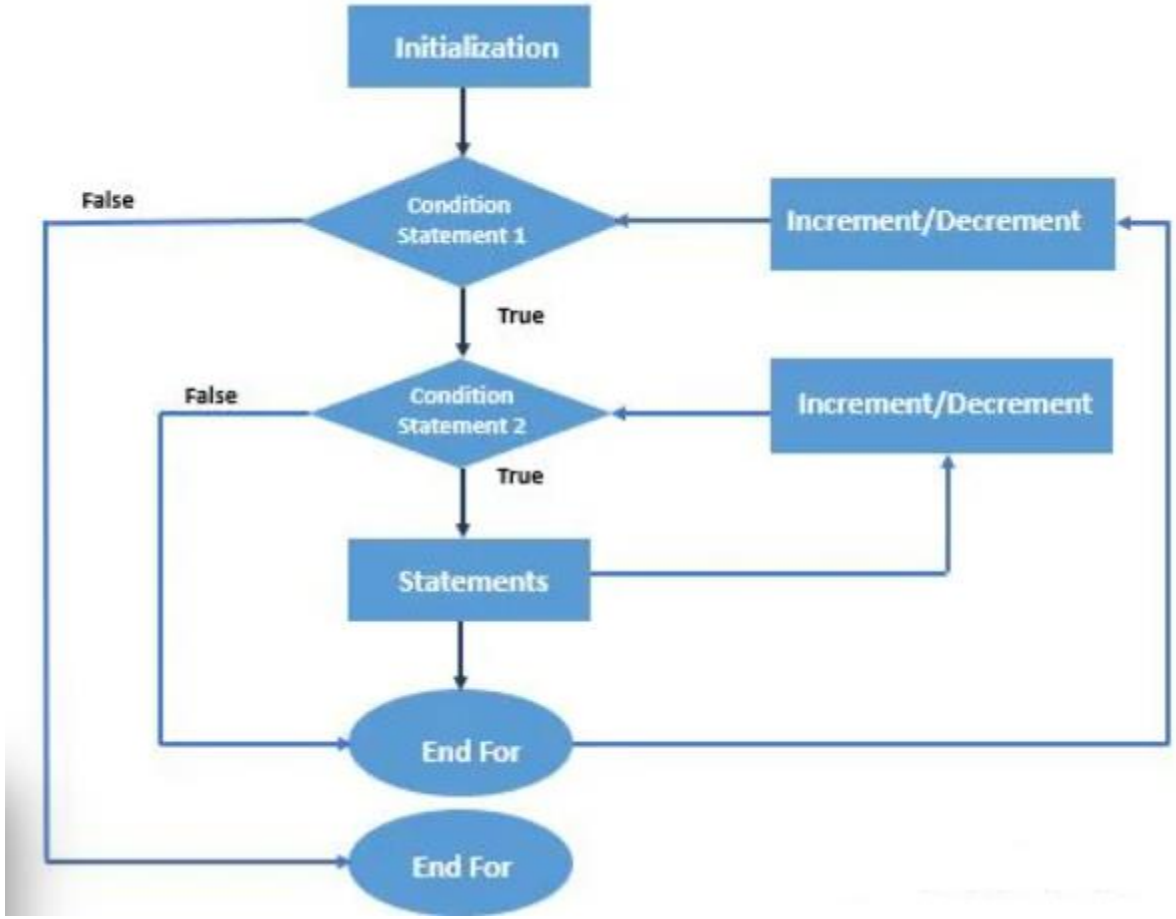
- Structured programming
  - Programs are easier to understand, test, debug and, modify.
- Rules for structured programming
  - Only single-entry/single-exit control structures are used
  - Rules:

    1) Begin with the “simplest flowchart”.
    2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
    3) Any rectangle (action) can be replaced by any control structure (sequence, if, if/else, switch, while, do/while or for).
    4) Rules 2 and 3 can be applied in any order and multiple times.

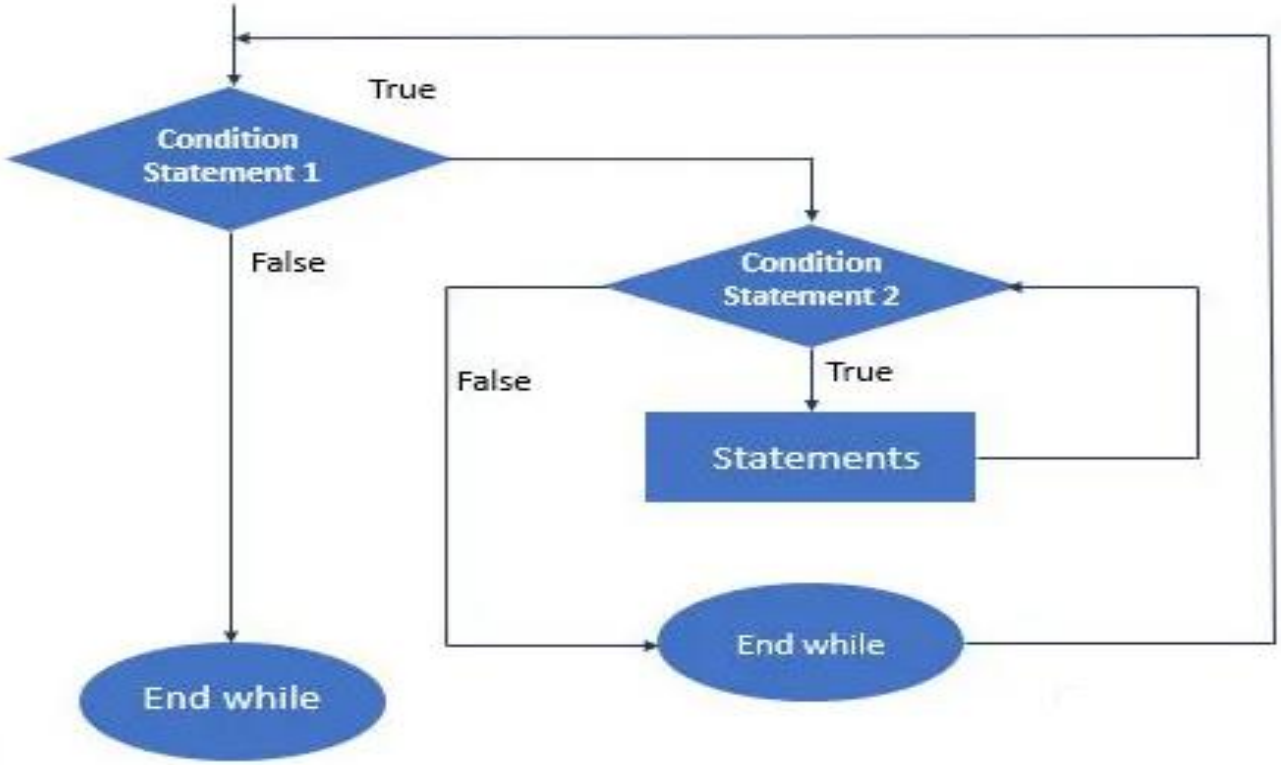
Structured-Programming Summary

- All programs can be broken down into
  - Sequence
  - Selection
    - **`if`**, **`if/else`**, or **`switch`**
    - Any selection can be rewritten as an **`if`** statement
  - Repetition
    - **`while`**, **`do/while`** or **`for`**
    - Any repetition structure can be rewritten as a **`while`** statement

Nested For Loop



Nested While Loop



Break Statement

```
#include <iostream>
using namespace std;

int main() {
    int weeks = 3, days_in_week = 7;
    for (int i = 1; i <= weeks; ++i) {
        cout << "Week: " << i << endl;

        for (int j = 1; j <= days_in_week; ++j) {
            // break during the 2nd week
            if (i == 2) {
                break;
            }
            cout << "    Day:" << j << endl;
        }
    }
}
```

```
Week: 1
    Day:1
    Day:2
    ... .. ...
Week: 2
Week: 3
    Day:1
    Day:2
    ... .. ...
```

Continue Statement

```
#include <iostream>
using namespace std;

int main() {
    int weeks = 3, days_in_week = 7;
    for (int i = 1; i <= weeks; ++i) {
        cout << "Week: " << i << endl;

        for (int j = 1; j <= days_in_week; ++j) {
            // continue if the day is an odd number
            if (j % 2 != 0) {
                continue;
            }
            cout << "    Day:" << j << endl;
        }
    }
}
```

```
Week: 1
    Day:2
    Day:4
    Day:6
Week: 2
    Day:2
    Day:4
    Day:6
Week: 3
    Day:2
    Day:4
    Day:6
```

Nested Loops

```
#include <iostream>
using namespace std;

int main() {
for (int i = 1; i <= 5; ++i) {
        for (int j = 1; j <= i; ++j) {
             cout <<  j ;
        }
       cout <<  “\n” ;

    }
}
```