Lecture 3.2

Topics

1. Conditional Statements – Revisited
2. Logical Operators – Revisited
3. Repetitive/Loop Structures – Review

---

1. Conditional Structure – Revisited

Recall that the most basic conditional structure is given below.

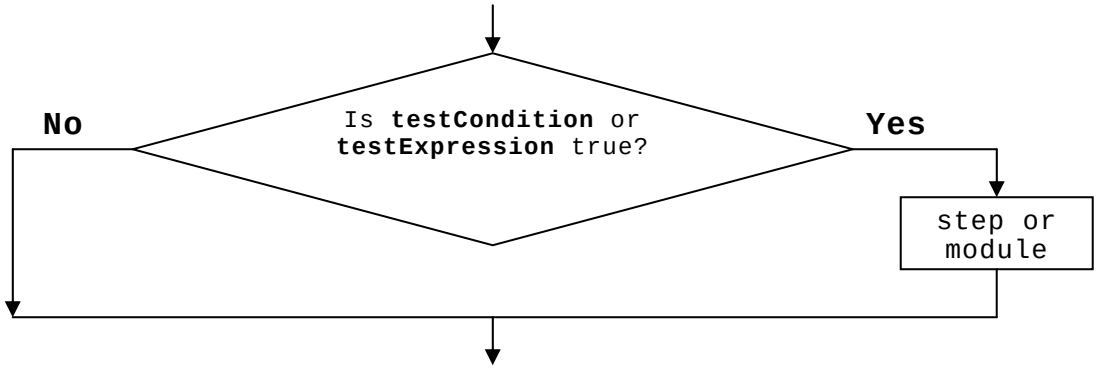


Figure 1 Conditional structure with one option **`if()`**

1.1 `if()` Structure

```
if ( testExpression ) //header of the if-statement/block
{
  //if true, do something here.
}
```

In the above structure, if the `testExpression` is **true** then the actions will take place; and this is the only one choice.

In many cases, we may want to do ***something else*** when `testExpression` is **false**. This can be depicted as follows,

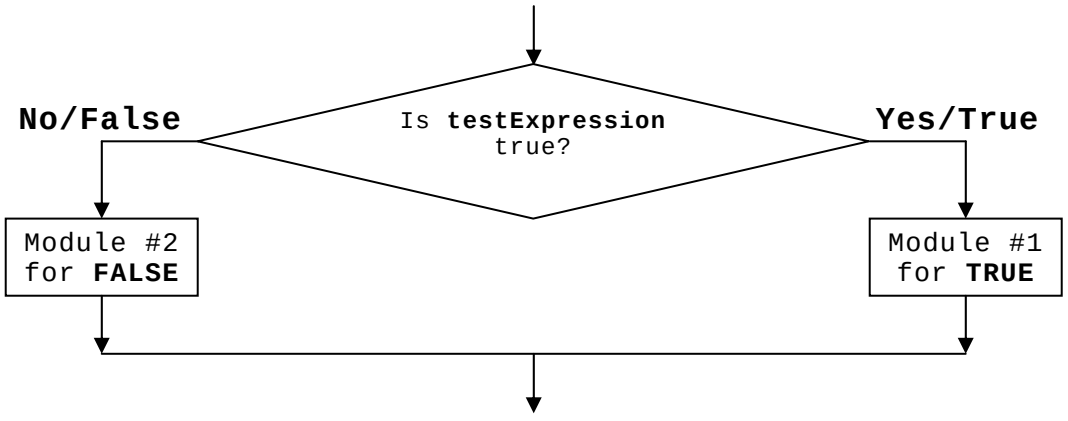


Figure 2 **`if ()-else`** conditional structure

1.2 `if()-else` Structure

```
if ( testExpression ) {
  //if true, do something here.
}
else {
  //else (i.e., false), do other thing here
}
```

Let's consider the following code fragment,

```
bTest = isEven();

if ( bTest == true ) {
  cout << "\nThe number is even!" << endl;
}

if ( bTest == false ) {
  cout << "\nThe number is odd!" << endl;
}
```

This can be rewritten as follows,

```
bTest = isEven();

if ( bTest ) {
  cout << "\nThe number is even!" << endl;
}
else {
  cout << "\nThe number is odd!" << endl;
}
```

1.3 Extended Structures

In some other cases, there may be more than two choices for the `testExpression`.

For examples, seven (7) days in a week, twelve (12) months in a year, etc. In these cases, the extended form of **`if-else`** can be used.

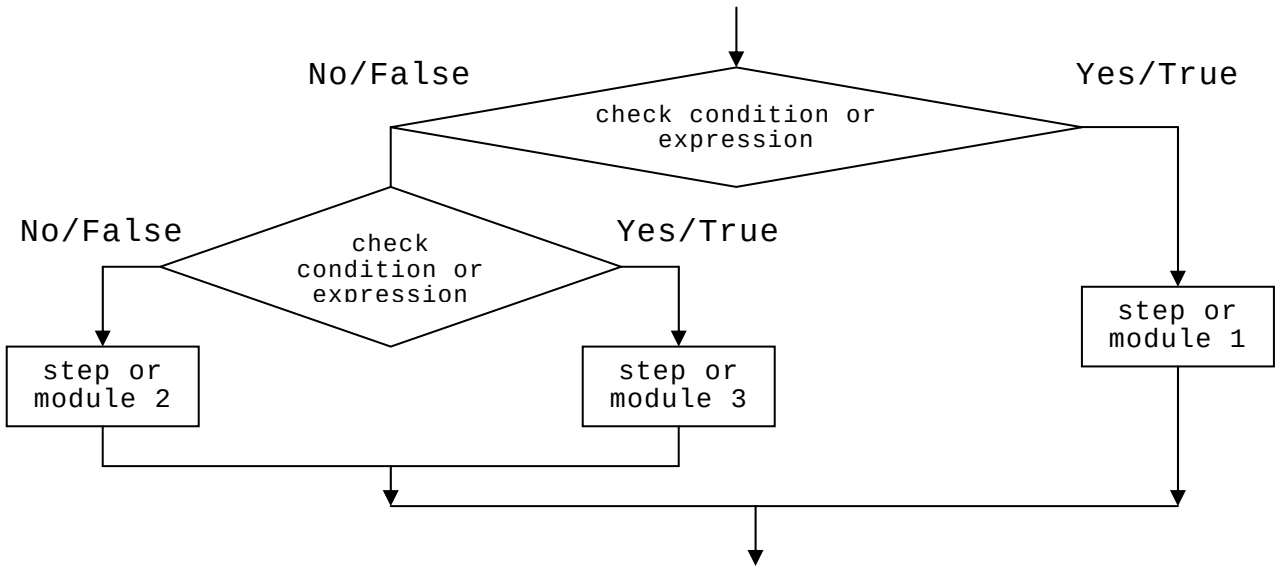


Figure 3 Extended conditional structure with multiple `if-else` components

The general syntax is given as follows,

`Extended if-else` Structure

```
if ( testExpression1 ) {
  //if testExpression1 is true, do action #1 here.
}
else if ( testExpression2 ) {
  //if testExpression2 is true, do action #2 here
}
else if ( testExpression3 ) {
  //if testExpression3 is true, do action #3 here
}
else {
  //if testExpression3 is false, do this last action
}
```

As seen in the above structures, there are several test expressions being checked out. These test expressions may be formed using logical operators.

2. C++ Logical Operators -- Revisited

Logical data of values true and false are used in expressions that involve with **`yes/no`** (or **`true/false`**) answer.

The C++ **`bool`** type is one that can be used to represent these values (that means Boolean like with values of **`true`** or **`false`**).

2.1 Logical Data

In general, a data value of zero (**`0`**) is considered as **false**, and data value of nonzero is considered as **true**.

A numeric representation of the **true-false** behavior is depicted in **Figure 4** below.

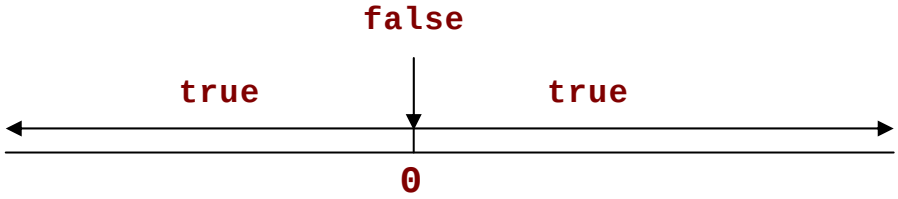


Figure 4 `True-False` scale in a numeric representation

2.2 Logical Operators

There are three logical operators of **`AND`**, **`OR`**, and **`NOT`**. The ranking and meaning of these operators are given in Table 1 below.

Operator	Meaning	Rank
!	Logical **`NOT`**	2
&&	Logical **`AND`**	11
||	Logical **`OR`**	12

Table 1 Logical Operators

2.3 Truth Table

The results of logical expressions are of logical value **`true`** or **`false`**.

The simplest expressions for the above three operators will have

- One single operand for the **`NOT (!)`** operator, and
- Two operands for the **`AND (&&)`** and **`OR (||)`** operators.

Note that operators enclosed in the parentheses are legal operators used in C++ programs.

Truth tables for these logical operators are given below

`NOT` : Logical Expression

a	!a
T	F
F	T

`NOT` : C++ Programming

a	!a
0	Nonzero
Nonzero	0

`AND` : Logical Expression

a	b	a && b
T	T	T
T	F	F
F	T	F
F	F	F

`AND` : C++ Programming

a	b	a && b
Nonzero	Nonzero	Nonzero
Nonzero	0	0
0	Nonzero	0
0	0	0

`OR` : Logical Expression

a	b	a || b
T	T	T
T	F	T
F	T	T
F	F	F

`OR` : C++ Programming

a	b	a || b
Nonzero	Nonzero	Nonzero
Nonzero	0	Nonzero
0	Nonzero	Nonzero
0	0	0

Table 2. `True-False` truth table

Using these logical operators, complex test expressions can be formed.

For examples,

(1) To test if the integer `iValue` is positive AND even, the expression may be written as

```
( iValue > 0 ) && ( iValue % 2 == 0 )
```

(2) To test if the integer is greater than 10 OR odd, the expression may be written as

```
( iValue > 10 ) || ( iValue % 2 )
```

3. Repetitive/Loop Structures – Review

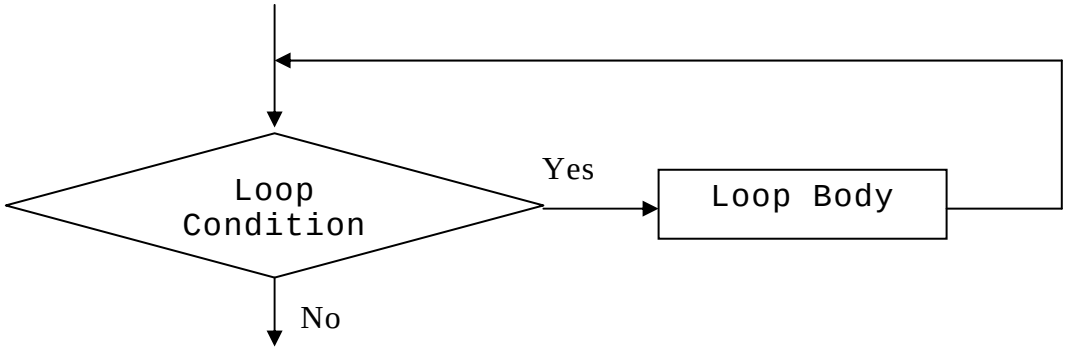
There are times when the same tasks/steps would be repeatedly applied to new sets of data. Then, the repetition or loop should be used.

- It is important to note that the size of the data set may be just one or several values, and
- Every set of data may have at least one different value from all other sets of data.

Recognizing and using a loop will eliminate a lengthy and repeated the same code fragment that would have been written many times in the program.

A loop will allow the code (instructions) to be written once and used with different sets of data.

General loop structures are depicted in Figures 6&7 below.



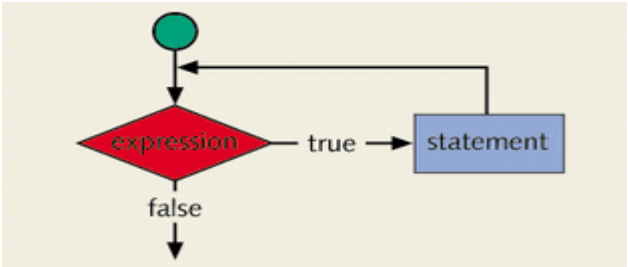
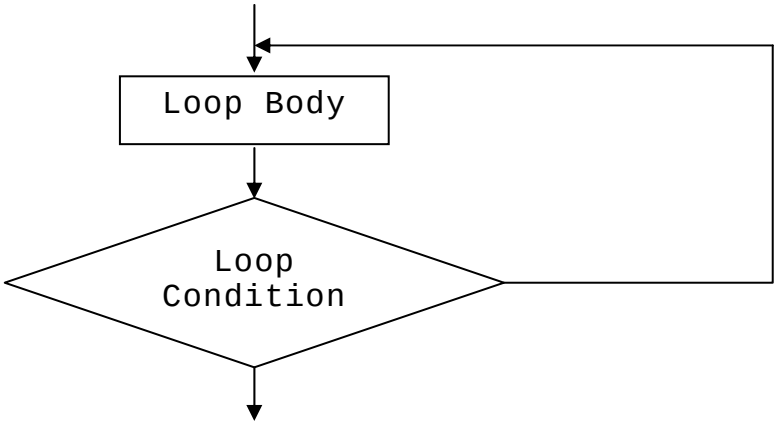


Figure 5 A general flowchart depicts a loop



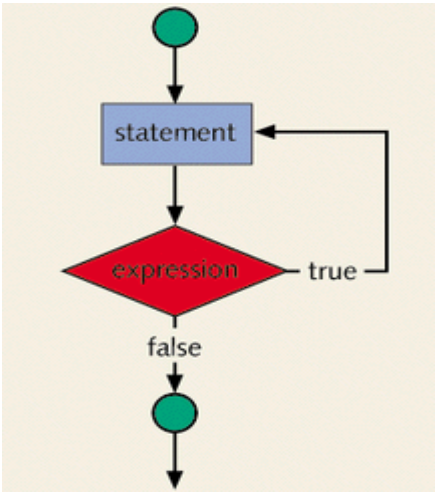


Figure 6 Loop with body executed at least once

Looking at flowchart for a loop,

- There is at least one conditional block and other processing blocks, and
- A loop must end through the result of this conditional block.

There are several different structures of setups for looping.

Let's look at these structures next.

3.1 Loop Structures

There are two basic looping requirements for the execution of a loop:

(1) Loop body must be executed at least once, and

(2) Loop body may not be executed at all.

Figure 5 depicts a loop with the loop body may not be executed at all. **Figure 6** depicts a loop that must be executed at least once

In the current C, C++, Java, C# and many other computer languages, there are several loop statements/structures that one can use.

And each loop can be thought of a process where three steps (or groups of tasks) are preformed after **initializing**:

(1) Testing/Conditioning,

(2) Loop Body, and

(3) Update.

They are given below.

Loops that may not have body executed at all:

Loop #1

```
//Initial Condition
while ( loopCondition ) {
  //Loop Body

  //Update
}
```

Loop #2

```
//Initial Condition
while ( loopCondition ) {
  //Update

  //Loop Body
}
```

Loop #3

```
for ( initialization; testExpression; update ) {
  //Loop Body
}
```

Loops that must be executed once:

Loop #1

```
//Initialization
do {
  //Loop Body

  //Update
} while ( loopCondition );
```

Loop #2

```
//initialization
do {
  //Update

  //Loop Body
} while ( loopCondition );
```

3.2 Example – Finding the Smallest Value

Problem Statement

Write a program to determine the smallest value from a given sequence of values. As a user is entering values, the program will keep track of the smallest value entered so far. The program will print this smallest value when user enters **`eof (ctrl+z)`**.

Analysis:

What do we have?

A sequence of values

What do we want?

The smallest value

What do we assume about the input values?

To be numeric and entered from keyboard

What may be done?

Read in one value.
Compare with the previous smallest value (an initial value stored previously).
Swap, if necessary, and store new smallest value.

Design:

How to get the smallest value?

- Read in one value.
- Compare with the previous smallest value (an initial value stored previously).
- Swap, if necessary, and store new smallest value.
- Read in another value.
- Compare with the previous smallest value (an initial value stored previously).
- Swap, if necessary, and store new smallest value.
- Keep repeating the process until `ctrl+z`.

Code Fragment – A Sample

```
int testEOF;

cin >> testEOF;
while ( testEOF != EOF ) {
  /*Perform operations to retain the smallest value*/

  cin >> testEOF;
}
```