C++ Foundation with Data Structures

Lecture 4 : Loops, Keywords, Associativity and Precedence

for loop

Loop statements allows us to execute a block of statements several number of times depending on certain condition. **for** loop is kind of loop in which we give initialization statement, test expression and update statement can be written in one line.

Inside for, three statements are written –
a. Initialization – used to initialize your loop control variables. This statement is executed first and only once.
b. Test condition – this condition is checked everytime we enter the loop. Statements inside the loop are executed till this condition evaluates to true. As soon as condition evaluates to false, loop terminates and then first statement after for loop will be executed next.
c. Updation – this statement updates the loop control variable after every execution of statements inside loop. After updation, again test conditon is checked. If that comes true, the loop executes and process repeats. And if condition is false, the loop terminates.

```
for (initializationStatement; test_expression; updateStatement) {
        // Statements to be executed till test_expression is true
}
```

Example Code :

```
int main(){
        for(int i = 0; i < 3; i++){
            cout<<"Inside for Loop : "<<i<<endl;
        }
        cout<<"Done";
}
```

Output:

```
Inside for Loop : 0
Inside for Loop : 1
Inside for Loop : 2
Done
```

In for loop its not compulsory to write all three statements i.e. initializationStatement, test_expression and updateStatement. We can skip one or more of them (even all three)

Above code can be written as:

```
int main () {
        int i=1;    // initialization is done outside the for loop
        for (; i <= 5; i++) {
                cout<<i<<end;
        }
}
```

OR

```
int main () {
        int i=1;     //// initialization is done outside the for loop
        for (; i <= 5; ) {
                cout<<i<<end;
                i++;  // updateStatement written here
        }
}
```

We can also skip the test_expression. See the example below :

Variations of for loop

- The three expressions inside for loop are optional. That means, they can be omitted as per requirement.

  Example code 1: Initialization part removed –

```
int main(){
        int i = 0;
        for(; i < 3; i++){
                cout<<i<<endl;
        }
}
```

  Output:

0
1
2

Example code 2: Updation part removed

```
int main(){
        for(int i = 0; i < 3; ){
                cout<<i<<endl;
                i++;
    }
}
```

Output:

0
1
2

Example code 3: Condition expression removed , thus making our loop infinite –

```
int main(){
        for(int i = 0 ; ;i++){
                cout<<i<<endl;
        }
}
```

Example code 4:

We can remove all the three expression, thus forming an infinite loop…

```
#include<iostream>
using namespace std;
int main(){
        for(; ;){
                cout<<"Inside for loop : "<<endl;
        }
}
```

- **Multiple statements inside for loop**

We can initialize multiple variables, have multiple conditions and multiple update statements inside a *for loop*. We can separate multiple statements using comma, but not for conditions. They need to be combined using logical operators.

Example code:

```
int main(){
        for(int i = 0,j = 4; i < 5 && j >= 0; i++, j--){
                cout<<i<<" "<<j<<endl;
        }
}
```

Output:

```
0 4
1 3
2 2
3 1
4 0
```

break and continue

1. **break statement**: The break statement terminates the loop (for, while and do. while loop) immediately when it is encountered. As soon as break is encountered inside a loop, the loop terminates immediately. Hence the statement after loop will be executed next.
2. **continue statement:** The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. (caution always update the counter in case of while loop else loop will never end)

```
while(test_expression) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

```
for (initializationStatement; test_expression; updateStatement) {
        // codes
        if (condition for break) {
                break;
        }
        //codes
}
```

❖ break

- Example: (using break inside for loop)

```
int main () {
        for (int i=1; i <= 10; i++) {
                cout<<i<<end;
                if(i==5)
                {
                        break;
                }
        }
}
```

Output:

```
1
2
3
4
5
```

- Example: (using while loop)

```
int main () {
        int i = 1;
        while (i <= 10) {
                cout<<i<<endl;
                if(i==5)
                {
                        break;
                }
                i++;
        }
}
```

Output:

1
2
3
4
5

- Inner loop break:

When there are two more loops inside one another. Break from innermost loop will just exit that loop.

Example Code 1:

```
int main () {
        for (int i=1; i <=3; i++) {
                cout<<i<<end;
                for (int j=1; j<= 5; j++)
                {
                        cout<<"in…" <<endl;
                        if(j==1)
                        {
                                break;
                        }
                }
        }
}
```

Output:

1
in…
2
in…
3
in…

Example Code 2:

```
int main () {
        int i=1;
```

```
        while (i <=3) {
                cout<<i<<end;
                int  j=1;
                while (j <= 5)
                {
                        cout<<"in…" <<endl;
                        if(j==1)
                        {
                                break;
                        }
                        j++;
                 }
                i++;
        }
}
```

Output:
1
in…
2
in…
3
in…

❖ Continue

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loopx.

- Example: (using for loop)

```
int main () {
        for (int i=1; i <= 5; i++) {
                if(i==3)
                {
                        continue;
                }
                cout<<i<<end;
```

```
        }
}
```

Output:
1
2
4
5

- **Example: (using while loop)**

```
int main () {
        int i=1;
        while (i <= 5) {
                if(i==3)
                {
                        i++;
                   // if increment isn't done here then loop will run
        infinite time for i=3
                        continue;
                }
                cout<<i<<end;
                i++;
        }
}
```

Output:
1
2
4
5

Scope of variables

Scope of variables is the curly brackets {} inside which they are defined. Outside which they aren't known to the compiler. Same is for all loops and conditional statement (if).

❖ Scope of variable --- for loop

```
for (initializationStatement; test_expression; updateStatement) {
        // Scope of variable defined in loop
}
```

Example:

```
int main() {
        for (int i=0; i<5; i++) {
                int j=2;        // Scope of i and j are both inside the loop they
                can't be used outside
}
```

❖ Scope of variable for while loop

```
while(test_expression) {
        // Scope of variable defined in loop
}
```

```
int main() {
        int i=0;
        while(i<5)
        {
                int j=2; // Scope of i is main and scope of j is only the loop
                i++;
        }
}
```

❖ Scope of variable for conditional statements

```
if(test_expression) {
        // Scope of variable defined in the conditional statement
}
```

```
int main () {
        int i=0;
        if (i<5)
        {
                int j=5;          // Scope of j is only in this block
```

```
    }
        // cout<<j;  → This statement if written will give an error because
                       scope of j is inside if and is not accessible outside if.
}
```

Increment Decrement operator

Explanation

Pre-increment and *pre-decrement* operators' increments or decrements the value of the object and returns a reference to the result.

Post-increment and *post-decrement* creates a copy of the object, increments or decrements the value of the object and returns the copy from before the increment or decrement.

Post-increment(a++):

This increases value by 1, but uses old value of a in any statement.

Pre-increment(++a):

This increases value by 1, and uses increased value of a in any statement.

Post-decrement(a--):

This decreases value by 1, but uses old value of a in any statement.

Pre-decrement(++a):

This decreases value by 1, and uses decreased value of a in any statement.

```
int main () {

        int I=1, J=1, K=1, L=1;

        cout<<I++<<' '<<J-- <<' '<<++K<<' '<< --L<<endl;

        cout<<I<<' '<<J<<' '<<K<<' '<<L<<endl;

}
```

Output:

1 1 2 0

2020

Bitwise Operators

Operator	Name	Example	Result	Description
a & *b*	and	4 & 6	4	1 if both bits are 1.
a | *b*	or	4 | 6	6	1 if either bit is 1.
a ^ *b*	xor	4 ^ 6	2	1 if both bits are different.
~*a*	not	~4	-5	Inverts the bits. (Unary bitwise compliment)
n << *p*	left shift	3 << 2	12	Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions.
n >> *p*	right shift	5 >> 2	1	Shifts the bits of *n* right *p* positions.

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

int main() {
   int a = 19;  // 19 = 10011
   int b = 28;  // 28 = 11100
  int c = 0;

  c = a & b;           // 16 = 10000
  cout << "a & b = " << c << endl ;

  c = a | b;           // 31 = 11111
  cout << "a | b = " << c << endl ;

  c = a ^ b;           // 15 = 01111
  cout << "a ^ b = " << c << endl ;

  c = ~a;              // -20 = 01100
  cout << "~a = " << c << endl ;

  c = a << 2;          // 76 = 1001100
  cout << "a << 2 = " << c << endl ;
```

```
  c = a >> 2;          // 4 = 00100
  cout << "a ≫ 2 = " << c << endl ;

  return 0;
}
```

Output

```
a & b = 16
a | b = 31
a ^ b = 15
~a = -20
a << 2 = 76
a >> 2 = 4
```


Precedence and Associativity

- **Operator precedence** determines which operator is performed first in an expression with more than one operators with different precedence.
  For example, 10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30.
- **Associativity** is used when two operators of same precedence appear in an expression. Associativity can be either **L**eft **t**o **R**ight or **R**ight **t**o **L**eft. For example, '*' and '/' have same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Precedence and Associativity are two characteristics of operators that determine the evaluation order of subexpressions in absence of brackets.

Note : We should generally use add proper brackets in expressions to avoid confusion and bring clarity.

1) Associativity is only used when there are two or more operators are of same precedence.

The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program,

associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of following program is in-fact compiler dependent.

```
// Associativity is not used in the below program. Output is compiler dependent.
 x = 0;
int f1() {
  x = 5;
  return x;
}
int f2() {
  x = 10;
  return x;
}
int main () {
  int p = f1() + f2();
  cout<<x;
  return 0;
}
```

2) All operators with same precedence have same associativity
This is necessary, otherwise there won't be any way for compiler to decide evaluation order of expressions which have two operators of same precedence and different associativity. For example, + and – have same associativity.

3) There is no chaining of comparison operators in C++
In C++ expression (a>b>c) will be treated as (a>b)>c .For example, consider the following program.. For example, consider the following program. The output of following program is "FALSE".

```
int main ()
{
  int a = 10, b = 20, c = 30;

  // (c > b > a) is treated as ((c > b) > a), associativity of '>'
  // is left to right. Therefore, the value becomes ((30 > 20) > 10). Since
(30>20) is true thus its answer is 1. So the expression becomes (1 > 20).

  if (c > b > a)
  cout<<"TRUE";
  else
```

```
  cout<<"FALSE";
 return 0;

}
```

Following is the Precedence table along with associativity for different operators.

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( ) [ ] . –>> ++ —	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left–to–right
++ — + – ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right–to–left
* / %	Multiplication/division/modulus	left–to–right
+ –	Addition/subtraction	left–to–right
<< >>	Bitwise shift left, Bitwise shift right	left–to–right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left–to–right
== !=	Relational is equal to/is not equal to	left–to–right
&	Bitwise AND	left–to–right

^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
| |	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right