

C++ Looping Structures

Contents

1	Overview of Looping Structures	1
2	The <code>while</code> Loop Structure	1
3	The <code>for</code> Loop Structure	2
4	The <code>do-while</code> Loop Structure	3
5	Nesting	4
6	A Note on Input Validation	5

1 Overview of Looping Structures

- Frequently we'll need a program to do things repeatedly, in a way that we could not capture without *looping structures*
 - Sometimes we'll want to do the *exact same thing* repeatedly
 - Other times we'll want to do something very similar repeatedly
 - C++ provides two looping structures, `while` and `for`
 - Similar to an `if` statement, looping structures consist of a C++ statement followed by a set of code enclosed in curly brackets
 - The code within the curly brackets is called the *body of the loop*
-

2 The `while` Loop Structure

- The C++ `while` loop provides a way of repeatedly executing a set of statements for as long as some boolean condition is true
- A `while` loop looks like the following...

```
while ( SomeCondition )
{
    statements;
}
```

- Once the `while` loop is encountered, the condition `SomeCondition` is evaluated...if it evaluates to `true`, then `statements;` is executed
 - After `statements;` is executed, `SomeCondition` is evaluated again, and if it still evaluates to `true`, `statements;` is again executed
-

- In short, *as long as the condition is true, C++ will keep executing the body of the while loop*

```
While SomeCondition is true...
    Keep executing this set of statements
```

- *examples...*
-

- What if the condition is always true? Will the `while` loop keep executing indefinitely?
- Answer: Yes. This is called an *infinite loop*, and is almost always undesirable
- It usually indicates a misunderstanding about what the looping condition should be, or a missing statement within the body of the loop

- *examples...*

- Remember... the goal is for the `while` loop to eventually halt
-

3 The for Loop Structure

- The C++ `for` loop provides another way of repeatedly executing a set of statements
 - Unlike the `while` loop, the `for` loop is used primarily for executing a set of code a fixed number of times
 - That is, *when you know how many times the loop will execute before the loop is encountered*.
 - *(Although as we'll see shortly, it's really just short-hand for a `while` loop)*
-

- A `for` loop looks like the following

```
for ( PreStatement; Condition; PostStatement )
{
    statements;
}
```

- Once the `for` loop is encountered...
 1. `PreStatement` (which is any valid C++ statement) is executed first.
 2. Next, `Condition` (which is any boolean expression) is evaluated. If it evaluates to `true`, then `statements;` is executed, followed by `PostStatement`
 3. Afterwards, `Condition` is again evaluated. If it evaluates to `true`, `statements;` is executed, followed by `PostStatement`
 4. The execution of these two statements continues for as long as `Condition` evaluates to `true`
- So in general...

```
Execute PreStatement;
For as long as Condition is true...
    Execute statements;
    Execute PostStatement;
```

- An example of using the `for` loop for doing something a fixed number of times...

```
for (int count = 0; count < 10; count++)
{
    cout << "Hello World!" << endl;
}
```

- *examples...*
-

- As hinted at above, the `for` loop can be viewed as shorthand for a `while` loop...
- For example, the following `for` loop...

```
for ( PreStatement; Condition; PostStatement; )
{
    statements;
}
```

- behaves *exactly* like the following `while` loop construct...

```
PreStatement;
while ( Condition )
{
    statements;

    PostStatement;
}
```

- If this is the case, then why provide such an odd looping structure?
- Answer: We frequently use a `for` loop in the following fashion, and proves useful to provide such a structure

```
for (int count = 0; count < someNumber; count++)
{
    // Do something with count
}
```

- *example...*
-

4 The do-while Loop Structure

- There is one more looping structure, called the `do-while` loop
- It's essentially a `while` loop, with one difference
- *It gaurantees the execution of its body at least once*

- The general format is

```
do
{
    statements;
} while ( Condition );
```

- NOTICE: The keyword `do` precedes the loop body
 - After the loop body comes the `while` statement with its condition
 - *There is a semicolon after the `while` statement!*
-

- The difference between the `do-while` and the `while` should be obvious from where the `while` statement appears.
 - In a regular `while` loop, the condition appears *before* the loop body, and is evaluated *before* each execution of the loop body
 - In a `do-while` loop, the condition appears *after* the loop body, and is evaluated *after* each execution of the loop body
 - The only real difference in terms of effect is that the `do-while` *will always* execute the loop body at least once!
-

- This behavior makes it well suited for instances where you want to get an input from the user and ensure it is of some specific value
- For example, a *menu selection* or some range of numbers...
- The following code repeatedly prompts the user until they enter a positive whole number...

```
int input;

do
{
    cout << "Enter a positive whole number: ";
    cin >> input;
} while (input <= 0);
```

5 Nesting

- Like control structures, looping structures may be nested within one another
 - Similarly, looping and control structures may (and frequently will) be nested/mixed within one another
 - *examples*
-

6 A Note on Input Validation

- Now that we have various *looping* structures, we know how to do things over (and over. . . and over. . . and over. . .) until some condition is true
- As mentioned in the `do-while` discussion and various examples, we can use loops when reading an input from the user to prevent the program from proceeding *until they have entered a valid value!*
- For example, you have a program where you need a value greater than zero from the user
- Rather than just prompting them for a value once, you repeatedly prompt them *until they enter a valid value!*
- This is called *input validation*
- ***From now on, in all homeworks and labs, all inputs from the user must be validated!***