CS 11 Data Structures and Algorithms

Lesson 4: Loops 1

Skip to Main Content

Section 4.1: Question Type Loops

Let's go back to our employee paycheck problem (from lesson 1 and lesson 3). At the end of the week when we are computing paychecks for our employees we have to run our program once for each employee. If we have 200 employees this would mean running the program 200 times. It would be quite an improvement if we could instead simply have the program repeat itself as long as we had more employees to process. In other words, we would like to be able to say something like this:

```
as long as there are more employees to process
        [compute one employee's paycheck]
```

How will we know when there are no more employees to process? We will have to ask the user whether there are more employees. Refining our plan for our program to include this fact, we get this:

```
ask the user if there is an employee to process
as long as the user answers "yes",
        [compute one employee's paycheck]
        ask the user if there is another employee to process
```

C++ provides us with a statement that does exactly this. It is called a **while loop.** The general form of a while loop is:

```
while (<condition>){
    <statements>
}
```

The <condition> part is exactly the same as the <condition> we saw in the if-statement. Each time C++ reaches a while statement during execution of a program, it checks to see whether the condition is true or not. If it is true, the statements inside the while loop (called the **body** of the while loop) are executed. If the condition is false, the body of the loop is not executed. C++ then continues on with the program, beginning with the next statement after the end of the while loop. This sounds an awful lot like an if-statement so far. The big difference is that when C++ finishes executing the body of the while loop, it returns to the top of the while loop, checking the condition again. This continues until the condition becomes false.

Using the while loop now to write our program so that it repeats itself, we get this:

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

int main()
{
    int hours;
    int paycheck;
    int payRate;
    char response;

    cout << "Is there an employee to process (Y/N)? ";
    cin >> response;
    while (response == 'Y'){
        cout << "Enter hours worked: ";
        cin >> hours;
        cout << "Enter rate of pay: ";
        cin >> payRate;

        if (hours <= 40) {
            paycheck = hours * payRate;
        } else {
            paycheck = 40*payRate + (hours-40)*payRate*1.5;
        }
```

```
            cout << "The amount of the paycheck is " << paycheck
                 << " dollars." << endl;

            cout<< "Is there another employee to process (Y/N)? ";
            cin >> response;
        }
    }
```

```
    Is there an employee to process? Y
    Enter hours worked:  15
    Enter rate of pay:  10
    The amount of the paycheck is 150 dollars.
    Is there another employee to process (Y/N)? Y
    Enter hours worked:  50
    Enter rate of pay:  20
    The amount of the paycheck is 1100 dollars.
    Is there another employee to process (Y/N)? N
```

Notice that in this program we ask the user if there is an employee to process before we even start executing the loop. In other words, **we are assuming that the user may decide not to process any employees at all**. This may seem strange. You might ask why the user would run the program at all if he wasn't planning on processing any employees. This is a good point, but consider the fact that this while loop might not be our entire program. It may be in the middle of a program that does several other things besides computing employees' paychecks. In this case, the user might be running the program for some other reason (besides computing paychecks) and might want to just skip this while loop. Also, while not likely, it is possible that a user could execute the program and then change his mind, deciding that there are no employees to process after all. In either of these two cases it would be highly undesirable to force the user to process an employee. **It is good practice to always assume, unless there is a specific reason to believe otherwise, that the user may decide not to execute the body of the while loop even once.**

The while loop program segment given above is an example of a **question-type loop**. It is called this because the decision about whether to continue executing the loop is made by asking the user a question ("is there another employee...."). There will be many instances where you will want to have a question-type loop in a program you are writing. When this is the case, there is no reason to try to figure out from scratch how to write the loop. You can simply use the example above as a pattern, replacing only the body of the loop and a few other words. The pattern looks like this:

```
cout << "Is there a ....? ";
cin >> response;
while (response == 'Y'){
    <body of loop>

    cout << "Is there another ....? ";
    cin >> response;
}
```

Section 4.2: Special-Value Type Loops

A second way to decide whether to continue executing a loop is to tell the user to enter a special value when she is done. We call this a **special-value type loop.** This is how we would like the screen output to look when using a special-value-type loop:

```
Enter hours worked (negative number to quit):  10
Enter rate of pay:  20
The amount of the paycheck is 200 dollars
Enter hours worked (negative number to quit):  50
Enter rate of pay:  20
The amount of the paycheck is 1100 dollars
Enter hours worked (negative number to quit): -4
```

Here's our first try at a program to implement a special-value-type loop, along with the screen output:

```
    #include <iostream>
    using namespace std;
    int main()
    {
        int hours;
        int paycheck;
        int payRate;

        while (hours >= 0){
            cout << "Enter hours worked (negative number to quit): ";
            cin >> hours;
            cout << "Enter rate of pay: ";
            cin >> payRate;

            if (hours <= 40) {
                paycheck = hours * payRate;
            } else {
                paycheck = 40 * payRate + (hours-40) * payRate * 1.5;
            }

            cout << "The amount of the paycheck is " << paycheck
                 << " dollars." << endl;
        }
    }
```

```
    Enter hours worked (negative number to quit): 10
    Enter rate of pay:  20
    The amount of the paycheck is 200 dollars.
    Enter hours worked (negative number to quit): -3
    Enter rate of pay: 20
    The amount of the paycheck is -20 dollars.
```

Did our program work as we had intended? Not quite. We wanted it to stop immediately after the user enters a negative number for the hours worked. Unfortunately, our program went on to ask the user for a rate of pay, and calculated the amount of the paycheck one last time before quitting. This illustrates a point that beginning computer programmers often find confusing:

A while loop does not immediately stop executing if the condition becomes false somewhere in the middle of the loop body.

The only time C++ looks at the condition is when the loop body is done executing and it is time to check the condition again. This fact leads us to our first important observation about our program.

The cin statement at which the user may enter the special-value (meaning she wants to stop executing the program) must come **immediately** before we check the condition.

Let's take another look at the screen output we want our program to produce. If you look carefully you will notice that we execute the statement

```
cout << "Enter hours worked (negative number to quit)";
```

one more time than we execute all of the other statements in the body of the loop. In our sample of how we would like our screen output to look, the line "Enter hours worked (negative number to quit)" appears 3 times, while all of the other lines appear only twice. This is because it must be both the **first** line of screen output and the **last** line. This is called a "fence-post" problem because it is like building a fence. If you build a fence in a straight line, you will always need one more fencepost than you need planks going between the fenceposts, because there must be a fencepost at the very beginning of the fence and there must also be a fencepost at the very end of the fence. So our second important observation about our program is this:

Because the cin statement at which the user may enter the special-value must occur one more time than the rest of the statements in our loop body, that statement must appear once outside the loop as well as inside the loop with the rest of the loop body statements.

Putting these two important observations together, here is our working program:

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

int main()
{
    int hours;
    int paycheck;
    int payRate;

    cout << "Enter hours worked (negative number to quit): ";
    cin >> hours;
    while (hours >= 0){
        cout << "Enter rate of pay: ";
        cin >> payRate;

        if (hours <= 40) {
            paycheck = hours * payRate;
        } else {
            paycheck = 40 * payRate + (hours-40) * payRate * 1.5;
        }

        cout << "The amount of the paycheck is " << paycheck
             << " dollars." << endl;

        cout << "Enter hours worked (negative number to quit): ";
        cin >> hours;

    }
}
```

```
Enter hours worked (negative number to quit):  10
Enter rate of pay:  20
The amount of the paycheck is 200 dollars
Enter hours worked (negative number to quit):  50
Enter rate of pay:  20
The amount of the paycheck is 1100 dollars
Enter hours worked (negative number to quit): -3
```

In the same way that the question type loop introduced in section 1 of this lesson may be used as a pattern, this special-value type loop may be used as a pattern.

Section 4.3: Counter Controlled Loops

Thus far we have discussed two types of loops: question-type loops and special-value-type loops. Before we progress too much further, let me clarify that these are not in any sense "official" C++ types of loops or even generally known types of loops. They are just loop patterns that I made up in an attempt to help you get used to working with loops. You will read about more loop patterns in the text. Special-value type loops correspond roughly to what the text calls "sentinel-controlled" loops.

Here are some hints for those of you who often have trouble thinking about how to begin solving a problem. Often you will get a problem and you won't know how to start but you will be able to identify the problem as one that requires a loop. Once you have realized this, the next question should be, "which loop pattern can I use to solve the problem?" Once you've answered this question, you can usually get a pretty good start on your program by using the loop pattern as a kind of template and then filling in the missing pieces.

We will now discuss a third type of loop, the **counter controlled** loop. The loops that we have discussed so far share the characteristic that when we initially enter the loop we do not know how many times the loop will be executed. In these two types of loops, the user may choose to stop executing the loop at any time. A counter-controlled loop is the opposite of this; that is,

a counter controlled loop is used when we know before entering the loop how many times it will be executed.

For example, let's write a program that will write out a name 10 times. Of course, we could do this by simply using 10 cout statements, like this:

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

    int main()
    {
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
        cout << "Dave" << endl;
    }
```

However, being the clever students that we are, we would like to discover a better way of doing this. Let's try using a loop that has as its body the `cout << "Dave" << endl;` statement. In other words, something like this:

```
    while ( ??? ) {
        cout << "Dave" << endl;
    }
```

What do we want to use as the condition for the while loop? We want to say

```
    while (number of times through the loop is less than 10) {
        cout << "Dave" << endl;
    }
```

In other words, we want to keep track of how many times we have executed the loop. We will introduce a new variable to keep track of this and call it `count`. We will initialize count to 0 before the loop (indicating that we have gone through the loop 0 times). We will increase the value of count by 1 each time we enter the loop, so the value of count will always tell us how many time we have gone through the loop.

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

    int main()
    {
        int count;

        count = 0;
        while (count < 10) {
            cout << "Dave" << endl;
            count = count + 1;
        }
    }
```

```
    Dave
    Dave
    Dave
    Dave
    Dave
    Dave
    Dave
    Dave
    Dave
    Dave
```

This program is an example of a counter-controlled loop. You should trace through the execution of this program on your own to make sure that you understand how it works. Are you convinced that it really does write out the name exactly 10 times? Are you sure it's not 9 times? Or 11 times?

It is actually quite difficult to write a counter-controlled loop that is exactly right. In the example above, we might have used `(count <= 10)` instead of `(count < 10)` as the condition. What would have happened? How did we know to use the latter condition? Should we have initialized the count to 1 instead of 0? How do we know? The hard way to approach counter-controlled loops is to start thinking about it from scratch each time you need to use one. The easy way is to use the program above as a pattern. Almost every counter-controlled loop you ever use will have this as the basic pattern:

```
count = 0;
while (count < howManyTimes){
    <body of loop>

    count = count + 1;
}
```

Section 4.4: Counter Controlled Loop Examples

Example 1

problem: Write a program which writes your name out n times, where n is a number entered by the user.

solution: We just need to ask the user for a number before we get started, and then use the number entered in the position where we used the number "10" before:

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

int main()
{
    int count;
    int numTimes;
    cout << "Enter number of times you want the name written: ";
    cin >> numTimes;

    count = 0;
    while (count < numTimes){
        cout << "Dave" << endl;
        count = count + 1;
    }
}
```

```
Enter number of times you want the name written:  7
Dave
Dave
Dave
Dave
Dave
Dave
Dave
```

Incidentally, C++ provides a shortcut for the statement `count = count + 1;`. This statement can be replaced with the statement `count++;`. These two statements are interchangeable, but be careful when you use the latter form. Many students get confused and write things like `count = count++;` which is completely wrong! If you aren't sure whether you are using `count++;` correctly or not, try taking it out and replacing it with `count = count + 1`. If it still looks right, then you are probably using it correctly.

Example 2:

Problem: Write a program that writes out the numbers from 1 to n, where n is entered by the user.

Solution: Each time through our loop we should write out the number corresponding to how many times we have gone through the loop, plus one. In other words, when we have gone through the loop 0 times, we should print out a 1. When we have gone through the loop one time, we should print out a 2. When we have gone through the loop 2 times, we should print out a 3, and so on. Since the variable `count` represents how many times we have gone through the loop, we just need to print out the value of `count + 1`.

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

int main()
{
    int count;
    int maxNumber;

    cout << "Enter number you want to count up to: ";
    cin >> maxNumber;

    count = 0;
    while (count < maxNumber){
        cout << count +1 << endl;
        count++;
    }
}
```

```
Enter number you want to count up to:  9
1
2
3
4
5
6
7
8
9
```

Section 4.5: More Counter Controlled Loop Examples

Example 3:

Problem: Write a program that adds up all the numbers from 1 to n where n is entered by the user.

Solution: This is also a counter-controlled loop very similar to our previous examples. We just need to have an additional variable which keeps track of the sum of the numbers so far. We need to start it off at 0 (since the sum of the numbers so far is 0 when we haven't seen any yet). Each time through the loop we need to add `count + 1` to our sum-so-far.

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

int main()
{
    int count;
    int maxNumber;
    int sumSoFar;

    cout << "Enter number you want to add up to: ";
    cin >> maxNumber;

    sumSoFar = 0;
    count = 0;
    while (count < maxNumber){
        sumSoFar = sumSoFar + (count + 1);
        count++;
    }
    cout << "The sum of the numbers from 1 to " << maxNumber
         << " is " << sumSoFar;
}
```

```
Enter number you want to add up to:  100
The sum of the numbers from 1 to 100 is 5050.
```

Example 4

Problem: Write a program that finds the average of all the numbers entered by the user. The user will indicate that there are no more numbers by entering a negative number. We will assume that the user enters only non-negative integers.

Solution: What kind of loop is this? Is it a counter-controlled loop? No, this is a special-value-type loop, because the loop is ended by a special value. We do **not** know before we enter the loop how many times it will be executed.

In order to compute the average we need to know two things: the sum of the numbers entered (as we computed in the last example), and how many numbers were entered. So, while this is a special-value-type loop, we **also** need to have a count so that when we finish the loop we know how many times it was executed.

We will use our basic special-value-type loop pattern, but we will insert some statements that will allow us to count the numbers.

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

int main()
{
    int num;
    int count;
    int maxNumber;
    int sumSoFar;

    sumSoFar = 0;
    count = 0;
    cout << "Enter a number (negative number to quit): ";
    cin >> num;
    while (num >= 0){
        sumSoFar = sumSoFar + num;
        cout << "Enter a number (negative number to quit): ";
        cin >> num;
        count++;
    }

    cout << "The average of the numbers is"
         << sumSoFar/count;
}
```

```
Enter a number (negative number to quit): 50
Enter a number (negative number to quit): 40
Enter a number (negative number to quit): 60
Enter a number (negative number to quit): 45
Enter a number (negative number to quit): 55
Enter a number (negative number to quit): -6
The average of the numbers is 50.
```

This program, unfortunately, has one little problem. What will happen if the user enters a negative number the first time? The average will be 0 divided by 0, which is undefined. The computer will not like this. So, in order to avoid this possible problem, we will add an if statement:

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

int main()
{
    int num;
    int count;
    int maxNumber;
    int sumSoFar;

    sumSoFar = 0;
    count = 0;
    cout << "Enter a number (negative number to quit): ";
    cin >> num;
    while (num >= 0){
        sumSoFar = sumSoFar + num;
        cout << "Enter a number (negative number to quit): ";
        cin >> num;
        count++;
    }

    if (count == 0){
        cout << "There is no average, because you did not "
             << "Enter any numbers!";
    } else {
        cout << "The average is " << sumSoFar/count;
    }
}
```

```
Enter a number (negative number to quit): -6
There is no average because you did not enter any numbers!
```

Section 4.6: One Last Counter Controlled Loop Example

Example 5

Problem: Write a program which finds the smallest of all the numbers entered by the user. The user will enter only non-negative integers, and will indicate that she is done entering numbers by typing a negative number.

Solution: The structure of our program will be similar to that of example 4: a special-value-type loop. We will have to introduce a new variable which will represent the smallest number we have seen so far. We'll call it `smallest`. When we read our first number, it is the smallest number we have seen so far, so we'll set smallest equal to that number (for example, if the first number we read is 7, then 7 is the smallest number we have seen so far). After that, each time we read a new number, we must check to see if it is smaller than our smallest-so-far. If it is, then we must replace our smallest-so-far with the new number.

Notice that, just as in example 4, we must check at the end to see if the user typed a negative number the first time. If she did, we should print an appropriate message.

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

int main()
{
    int num;
    int smallest;

    cout << "Enter a number (negative number to quit): ";
    cin >> num;
    smallest = num;

    while (num >= 0){
        if (num < smallest){
            smallest = num;
        }

        cout << "Enter a number (negative number to quit): ";
        cin >> num;
    }

    if (smallest < 0){
        cout<< "There is no smallest because you "
            << "did not enter any numbers!";
    } else {
        cout << "The smallest number you entered was "
             << smallest << endl;
    }
}
```

```
Enter a number (negative number to quit): 47
Enter a number (negative number to quit): 8
Enter a number (negative number to quit): 37
Enter a number (negative number to quit): 5
```

```
     Enter a number (negative number to quit): 9
     Enter a number (negative number to quit): -4
     The smallest number you entered was 5
```