CSCI1300 - Introduction to Computer Programming
Instructor: Dr. Knox

Recitation 5: Nested While Loops

Nested While Loops

In this recitation we will cover nested while loops. Let's remind ourselves what while loops look like and why we use them:

Syntax and Form:

```
while (condition)
{
        //statement to do something;
}
```

where *while* is a C++ reserved word, *condition* is a boolean-expression that will evaluate to a true or false statement, and *statement to do something* is enclosed by curly brackets. If the condition is true, the specified statement within the loop is executed. After running once, the boolean-expression is then re-evaluated. If the statement is true, then it is executed again. This process of evaluation and execution is repeated until the condition becomes false.

Example:

```
int userChoice = 1;
while (userChoice != 0)
{
        cout << "Do you want to see this question again? Press 0 for no, any other number for
        yes.";
        cin >> userChoice;
}
```

Could we write this piece of code without a loop? No. We have no way of knowing how many times a user will enter non-zero values before the program begins. Many tasks are difficult or impossible to accomplish without using some kind of looping mechanism.

A "nested loop" is a loop within a loop. The inner loop is executed once for every iteration of the outer loop. Consider the following example

Example:

```
int i = 0;
while (i < 5)
{
        int j = 0;
        while (j <= i)
        {
                cout << j << " ";
                j++;
        }
        cout << endl;
        i++;
}
```

The outer loop in this example is going over values of i from 0 to 5. The inner loop goes over values of j from 0 up to the current value of i and prints the values of j separated by spaces. Each time the inner loop finishes a new line is started. The output in this case would be:

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

Nested loops are extremely common and are often used when dealing with multiple lists or sets of information. They are also somewhat difficult to understand the first time they are encountered. The key to becoming comfortable with these kinds of patterns is practice.

Character Case Conversion

When dealing with strings it is often useful to be able to convert between upper and lower case characters. If we are looking for matches of the word "hello" in a long text document, we might not care about capitalization. There are two built-in C++ functions which help us with this task, **tolower** and **toupper**. These functions work almost exactly like you would expect. However, there is one important subtlety. **tolower** has the following function signature:

```
int tolower(int c);
```

This means that **tolower** actually takes an **int**, not a char, as input and returns an **int**. It turns out that C++ treats **ints** and **chars** very similarly. Indeed, we saw a conversion table from **int**s to

chars using the ASCII encoding in the previous recitation.

Here are a few examples of how to use these functions:

```
string str = “HELLO, world!”;
int i = 0;
while (i < str.length()) {
        str[i] = tolower(str[i]);
        i++;
}
```

This results in str containing the value “hello, world!”

```
string str = “HELLO, world!”;
string newStr = “”;
int i = 0;
while (i < str.length()) {
        newStr = newStr + char(tolower(str[i]));
        i++;
}
```

In this example we build a new string, newStr, instead of altering the value of str. Notice that we have added **char(...)** around the call to **tolower**. This is called “casting”. It tells the computer to interpret the value returned by **tolower**, which is an **int**, as a character. In this case, the result of **tolower** is interpreted as the ASCII encoding of some character. This character is then the result of the full expression **char(tolower(str[i]))**.

Removal of a Substring

Last week one of the recitation questions involved removing all occurrences of a substring from a given string. Here is one way of accomplishing this task using nested while loops:

```
string removeSubstring(string str, string substring){
   bool stringModified = true;

   while (stringModified){
      stringModified = false;
      int i = 0;
      while (i < str.length()){
         string s = str.substr(i, substring.length());
```

```
            if (s == substring) {
                str.replace(i, substring.length(), "");
                stringModified = true;
            }
            i++;
        }
    }
    return str;
}
```

Activity

Download **recitation5pattern.cpp** and **recitation5pattern2.cpp** files from Moodle.
Use **recitation5pattern.cpp** to implement the first pattern of increasing length output lines.
Use **recitation5pattern2.cpp** to implement the second pattern of decreasing length lines.

For this activity you will have to generate a tree of asterisks like the following:

```
*
* *
* * *
* * * *
* * * * *

and

* * * * *
* * * *
* * *
* *
*
```

Implement a program that asks the user for the number of rows and outputs to the console a tree made out of spaces, " ", and asterisks, "*", depending on the number of rows.

Once you have your code working, zip your two .ccp files together and **submit your .zip file to Moodle**. Don't forget your name, recitation section, and TA in a comment in the header and comments for each of your files. **Then complete the Recitation 5 Coding Activity on Moodle**.