CSI 1430
Week 5 Resource

Colin Burdine

2/14/21

Major Topics:

1. Variable Scope
2. Loops

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Variable Scope

Last week, we discussed some of the conditional branching structures (`if`, `if-else`, etc.), which are quite useful for executing mutually exclusive lines of code depending on some condition. However, we do have to be careful about what variables are used within a conditional branch, as it can create problems that result in unexpected behavior. For example, consider the code segment below:

```
1 bool someCondition = true;
2 int x = 42;
3
4 if(someCondition){
5     int x = 999;
6 }
7 // what will be printed? Will this program even compile?
8 cout << "The value of x is: " << x << endl;
```

Notice that if we put this into a C++ source file, it will compile and run just fine; the program output will be

```
The value of x is:  42
```

So what exactly is going on here? It seemed that on line 2 and 5 we declared the same variable twice, yet it still compiled. Furthermore, if we were to remove the `int` type specifier on line 5 above, the program outputs:

```
The value of x is:  999
```

The reason that this behavior emerges is due to the *scoping rules* in C++. As you saw above, we are allowed to redeclare the same variable twice, but the catch is that we can only redeclare variables if they have not yet been declared in the same *scope*. But what is a *scope*, exactly? a scope is a segment of code within which a variable can be accessed by name. In the code above, our first declaration of `x` has the entire program as its scope, while the second declaration of `x` only has scope within the brackets of line 5. Note that the two scopes for `x` are nested inside each other.

In C++, whenever a variable is accessed or modified, the variable that has the *innermost* scope is the one that is actually accessed or modified. In C++ we enclose the scopes of variables within brackets, as you have seen before with `if` statements and other structures. We notice in the code above that once the if statement is complete, the instance of `x` we declared inside the `if` statement is no longer accessible, and so the variable name `x` is now associated with the initially declared value of 42.

1.2 Loops

In the past couple of weeks we introduced conditional branching structures in C++. This week we will be diving into another key structure- loops! Loops allow for the repeated execution of a sequence of instructions or lines of code. Typically a loop has a *loop condition* and and an *exit condition*. A loop condition is the condition upon which a loop continues to execute. An exit condition, is the condition on which a loop stops executing. These two conditions should be the logical NOT of eachother (think about why), which means we only need to specify one to know how long to keep looping. In C++ we specify the loop condition (the condition under which we continue to loop) only.

Below, we introduce two of the basic loop structures in C++, the `while` and the `do-while` loop:

```
1
2  // the while loop:
3  while( [ loop condition ] ){
4      // This is the loop body that is executed until
5      // the loop condition evaluates to false. If
6      // the loop condition was false to begin with,
7      // this loop body is not executed.
8  }
9
10 // the do-while loop:
11 do{
12     // This is the loop body that is executed until
13     // the loop condition evaluates to false.
14     // In a do-while loop, the body is always executed
15     // at least once, and then the loop condition is
16     // checked
17 } while( [ loop condition ] );
```

Another useful loop structure used ubiquitously in C++ is the `for` loop. It has a more complex structure than the previous loops. It has three segments, an *initialization* segment, a *loop condition* segment, and a *post-body* segment. The initialization segment is always executed exactly once before the loop body is entered. The loop condition is the condition for looping, which behaves in a manner similar to the `while` loop (not the `do-while`). Finally, the post-body segment is executed at the very end of the loop body, right before the loop condition segment is checked once again. The syntax for the `for` loop is given below:

```
1
2  // the for loop:
3  for( [ initialization ]; [ loop condition ]; [ post-body ] ){
4      // this is the loop body. Its execution is similar to
5      // the while loop.
6  }
```

In a typical for loop execution, the order in which the segments are executed resembles the following pattern:

```
[initialization]
[loop condition (is found to be true)]
[loop body]
[post-body]
[loop condition (is found to be true)]
    ...
[loop condition (is found to be false)]
    (done)
```

2 Programming Example Problem

To illustrate how we can use loops to solve problems, consider the following example programming problem:

We can use loops to do some interesting things, such as print out ASCII art, such as the following 16×4 checkerboard (left) or 3×3 checkerboard (right):

```
------------------          -----
|#X#X#X#X#X#X#X#X|          |#X#|
|X#X#X#X#X#X#X#X#|          |X#X|
|#X#X#X#X#X#X#X#X|          |#X#|
|X#X#X#X#X#X#X#X#|          -----
------------------
```

Write a program that prompts the user for both a positive width w and a positive height h and prints out a $w \times h$ checkerboard. If either of the values are invalid, print an error message and exit.

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to use the empty space below to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```cpp
/* Your proper header comments should go here :) */
#include <iostream>
using namespace std;

int main(){
    int w, h;

    // prompt user for a height and width:
    cout << "Enter a width and a height: ";

    // read in the width and height:
    cin >> w >> h;

    // ensure input width and height are positive:
    if(w <= 0 || h <= 0){
        cout << "Error- both width and height must be positive"
             << endl;
    } else {
        // print out checkerboard top:
        for(int c = 0; c < w+2; ++c){
            cout << '-';
        }
        cout << endl;

        // print out checkerboard middle rows:
        for(int r = 0; r < h; ++r){

            // print out full row:
            cout << '|';
            for(int c = 0; c < w; ++c){
                // determine whether we should
                // print a '#' or a 'X' via the
                // modulus operator:
                if((c+r)%2 == 0){
                    cout << '#';
                } else {
                    cout << 'X';
                }
            }
            cout << '|' << endl;
        }

        // print out checkerboard bottom:
        for(int c = 0; c < w+2; ++c){
            cout << '-';
        }
        cout << endl;
    }
}
```