CSCE 155 - C

Lab 04 - Loops

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review the lecture notes on conditionals, or the following free textbook resources:
   - http://en.wikibooks.org/wiki/C_Programming/Control#Loops
   - http://www.cs.cf.ac.uk/Dave/C/node6.html

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming

to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- How loop structures work and how to use them
- The difference between for-loops, while-loops, and do-while loops
- How to use loops to solve a problem

2 Background

Loop control structures allow us to design algorithms and procedures that repeatedly execute a block of code. This allows us to repeatedly perform the same computation until some terminating condition is met, or to perform the same operation on a collection of data. Three common loop control structures are the for-loop, the while-loop, and the do-while loop. Though the syntax and behavior are slightly different, each one supports the same basic functionality. In fact, loops are usually interchangeable: for example any for-loop can be rewritten as a while-loop and vice versa.

A for-loop has three basic components: an initialization statement, a terminating condition, and an iteration statement. An example:

```
1  int i;
2  for(i=0; i<10; i++) {
3    printf("%d\n", i);
4  }
```

In this loop the first statement initializes the variable `i` to 0 while the third increments `i` by one on each iteration of the loop. The second statement is the terminating condition: it is evaluated at the start of each iteration of the loop. If the condition is true, then the loop executes; if it evaluates to false, the loop does not execute any further. The loop above would print out the values 0 thru 9. On the 10th iteration, the variable i would have a value of 10 and thus `i < 10` would no longer be true and the loop ends. Take careful note of the syntax and usage of semicolons.

A while-loop is similar, but does not collect all three components in the same line. With a while loop, the terminating condition is still checked at the beginning of the loop. Only if the condition is true does the loop execute.

```c
int i=0;
while(i<10) {
  printf("%d\n", i);
  i++;
}
```

In this example, the initialization is done outside, before the loop while the increment is done as part of the loop (inside the loop's code block).

Another loop structure, the do-while loop, is slightly different in how it behaves. Though it is similar to the while-loop, the key difference is that the do-while loop checks the terminating condition at the end of a loop's iteration. As a consequence, the code block in a do-while loop is always executed at least once.

```c
int i;
do {
  printf("%d\n", i);
  i++;
} while(i<10);
```

Note a slight syntax difference as well: there is a semicolon used at the end of the terminating condition.

3 Activities

We have provided partially completed programs for each of the following activities. Clone the lab's code from GitHub using the following URL: https://github.com/cbourke/CSCE155-C-Lab04.

3.1 Computing Sine

The standard math library provides a function to compute the sine of a given number. Complex functions such as these are usually approximated using some numeric analysis technique. One such technique is to use a Taylor series to approximate the sine function:

$$\sin x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Obviously, we cannot compute an infinite series. Instead, we will approximate $\sin(x)$ by computing the Taylor series above out to n terms.

We have provided you with an incomplete program, `sine.c` that reads in two values from the command line, x and n. We have also written a function to compute the factorial which you may find useful. You will need to complete the program by writing code to compute the approximation of $\sin(x)$.

Complete the program and answer the questions on your worksheet.

3.2 Number Guessing Game

Shall we play a game? In this game, a computer program randomly generates a number between 1 and 1,000. You, the player, then make guesses as to what that number is. If you guess correctly, the game ends and you win. For each wrong guess, the computer program gives you a hint by telling you whether your guess is too low or too high. The goal is to minimize the number of guesses you make.

We have provided an incomplete program, `guessingGame.c` that randomly generates a number between 1 and 1,000. You need to write a loop structure that prompts the user for a guess. While that guess is wrong, the loop should continue. It should also keep track of how many guesses the user makes and, for each wrong guess it should print the appropriate hint to the user.

Complete the program and answer the questions on your worksheet.

3.3 Nesting Loops

It is possible (and often necessary) to *nest* loops. That is, to perform a loop within another loop (just as you can nest conditional statements). The syntax is straightforward: you simply write another loop within a loop. For each iteration that the "outer" loop executes, the "inner" loop goes through its entire execution. As an example, consider the following code snippet with corresponding output.

```
1  int i, j;
2  for(i=0; i<3; i++) {
3    for(j=1; j<5; j++) {
4      printf("%d ", (i+j));
5    }
6    printf("\n");
7  }
```

Output:

```
 1 2 3 4
 2 3 4 5
```

```
3 4 5 6
```

Primality Testing

Recall that an integer is *prime* (ex: 2, 3, 5, 7, etc.) if it is only divisible by 1 and itself. Otherwise, it is called composite. Consider the following algorithm to determine if an integer x is prime: for each integer y from 2 up to the square root of x, if y divides x, then x is composite (not prime). Otherwise if no such integer in that range divides x, then x is prime.

Starter code has been provided in `primes.c` that reads in an integer n as a command line argument. Then, for each integer 2 up to n, it outputs the integer and whether it is prime or composite. You will need to complete the program by writing a nested loop that determines if a value is prime or composite. Demonstrate your working program to a lab instructor and have them sign your worksheet.

4 Advanced Activities (Optional)

Write a program that is the "reverse" of the number guessing game. That is, the computer is the player and the user is the one who picks a number between a certain range. The computer should then formulate an optimal guessing strategy and the user indicates if the actual number is correct, lower or higher. The program should be "smart enough" to know when the user is cheating.