**ITCS 1212L**

**Lab Lessons 7**

**Objectives:**

* **To work with the while loop**
* **To introduce the do-while loop**
* **To work with the for loop**
* **To work with nested loops**
* **To briefly introduce the concept of files**
* **Debugging**

**Data Validation:**

One nice application of the while loop is data validation. The user can input data (from the keyboard or a file) and then a while loop tests to see if the value(s) is valid. The loop is skipped for all valid input but for invalid input the loop is executed and prompts the user to enter new (valid) input. The following is an example of data validation.

cout << "Please input your choice of drink " << "(a number from 1 to 4 or 0 to quit)" << endl;

cout << " 1 - Coffee" << endl <<"2 - Tea"<<endl <<"3 - Coke"<<endl << " 4 - Orange Juice" << endl << endl <<"0 - QUIT"<<endl<<endl;

cin >> beverage;

while (beverage < 0 || beverage > 4) {

cout << "Valid choices are 0 - 4. Please re-enter: ";

cin >> beverage; }

What type of invalid data does this code test for? If beverage is an integer vari- able, what happens if the user enters the character ‘$’ or the float 2.9?

**Sentinel Values**

We can also control the execution of a loop by using a sentinel value which is a special value that marks the end of a list of values. In a variation of the previ- ous program example, if we do not know exactly how many test scores there are, we can input scores which are added to total until the sentinel value is input. Sample Program 5.4 revises Sample Program 1 to control the loop with a sen- tinel value. The sentinel in this case is -1 since it is an invalid test score. It does not make sense to use a sentinel between 0 and 100 since this is the range of valid test scores. Notice that a counter is still used to keep track of the number of test scores entered, although it does not control the loop. What happens if the first value the user enters is a -1?



Sample Program 1

#include <iostream>

using namespace std;

int main() {

int score, test = 1 ;

float total = 0, average;  // total and average of scores

// the individual score read in

cout<<"Enter your score on test "<<test<<"(or -1 to exit):" <<endl;

cin >> score;

while (score != -1) {

total = total + score; //running total is updated

test++;

cout<<"Enter your score on test "<<test<< "(or -1 to exit):"<< endl;

cin >> score; // Read the next score

}

if (test > 1) // If test = 1, no scores were entered

{

average = total / (test - 1);

cout<<"Your average based on "<<(test - 1)<<" test scores is "<< average << endl;

}

return 0;

**do-while Loop:**

A do-while loop is very similar to a while loop. The distinction is the condition is placed under the block of code to repeat. This means the block of code will execute one time before the condition is checked. After the block executes the condition is checked. If the condition evaluates to true, the block repeats execution. If the condition evaluates to false, the program execution continues with the code under the block.

Do while loops are excellent for data validation and menu-driven program.

**for Loop:**

A for-loop is used when you or the user knows ahead of time how many iterations (repetitions) of the block should execute. The number of times to repeat is specified in the loop header:

int index;  
//the for-loop header consists of initialization of a //loop control variable, an expression that evaluates //to true or false and an update statement:

for(index = 0; index < 5; index = index+1) {

//block of statements you want to repeat

//In this example this block will execute 5 times }

STEP 1: index = 0, initializes an integer variable, index, to 0, the initialization is performed only one time.

STEP 2: Then the condition is checked. This condition evaluates to either true or false. If the value is true, the block executes.

STEP 3: After the block executes, the update statement is executed.

In this example, 1 is added to index. The condition is then checked again. If the condition evaluates to true, then the block executes. If the condition evaluates to false, the program execution continues after the block.

You, as the programmer should determine the initial value of your loop control variable, the condition and the update depending on the logic of your program.

**Nested Loops**

Often programmers need to use a loop within a loop, or nested loops. Sample Program 2 below provides a simple example of a nested loop. This program finds the average number of hours per day spent programming by each student over a three-day week- end. The outer loop controls the number of students and the inner loop allows the user to enter the number of hours worked each of the three days for a given student. Note that the inner loop is executed three times for each iteration of the outer loop.

Sample Program 2:

// This program finds the average time spent programming by

// a student each day over a three day period.

#include <iostream>

using namespace std;

int main() {

int numStudents;

float numHours, total, average;

int count1 = 0, count2 = 0; //these are the counters for the loops

cout<<"This program will find the average number of hours a day" <<

" that each given student spent programming over a long weekend" <<

endl << endl;

cout << "How many students are there ?" << endl << endl;

cin >> numStudents;

for (count1 = 1; count1 <= numStudents; count1++) {

total = 0;

for (count2 = 1; count2 <= 3; count2++) {

cout << "Please enter the number of hours worked by student " <<

count1 << " on day " << count2 << "." << endl;

cin >> numHours;

total = total + numHours; //running total is updated

}

average = total / 3;

cout << endl;

cout << "The average number of hours per day spent programming by"

<< " student " << count1 <<" is " << average <<endl<<endl << endl;

}

return 0.

}

**Files:**

So far all our input has come from the keyboard and our output has gone to the monitor. Input, however, can come from files and output can go to files. To do either of these things we should add the #include <fstream> directive in the head- er to allow files to be created and accessed. A file containing data to be input to the computer should be defined as an ifstream data type and an output file should be defined as ofstream.

Sample Program 3

Suppose we have a data file called grades.dat that contains three grades, and we want to take those grades and output them to a file that we will call final-grade.out. The following code shows how this can be done in C++.

#include <fstream> //This statement is needed to use files

using namespace std;

int main() {

float grade1, grade2, grade3; // This defines 3 float variables

ifstream dataFile; // This defines an input file stream.

 // dataFile is the "internal" name that is used in the program for

// accessing the data file.

ofstream outFile; // This defines an output file stream.

// outFile is the "internal" name that is used in the program for

// accessing the output file.

outFile << fixed << showpoint; // These can be used with output

// files as well as with cout.

dataFile.open("grades.dat"); // This ties the internal name,

// dataFile, to the actual file, grades.dat.

outFile.open("finalgrade.out"); // This ties the internal name,

// outFile, to the actual file, finalgrade.out.

// This reads the values from the input file into the 3 variables.

dataFile >> grade1 >> grade2 >> grade3;

// These 3 lines write the values stored in the 3 variables to the

// output file

outFile << grade1 << endl;

outFile << grade2 << endl;

outFile << grade3 << endl;

return 0;

}

//

**Debugging:**

Many IDEs (Integrated Development Environments) have software debuggers, which are used to help locate logic errors; however, programmers often use the concept of stubs and drivers to test and debug programs that use functions and procedures. A **stub** is nothing more than a dummy function that is called instead of the actual function. It usually does little more than write a message to the screen indicating that it was called with certain arguments. In structured design, the programmer often wants to delay the implementation of certain details until the overall design of the program is complete. The use of stubs makes this possible.

#include <iostream>

using namespace std;

int findSqrRoot(int x); //prototype for a user defined function that

// returns the square root of the number passed to it

int main() {

int number;

cout << "Input the number whose square root you want." << endl;

cout << "Input a -99 when you would like to quit." << endl;

cin >> number;

while (number != -99) {

cout << "The square root of your number is " << findSqrRoot(number) << endl;

cout << "Input the number whose square root you want." << endl; cout << "Input a -99 when you would like to quit." << endl; cin >> number;

}

return 0;

}

**int findSqrRoot(int x) {**

**cout<<"findSqrRoot function was called with "<<x<< " as its argument\n";**

**return 0; // This bold section is the stub.**

**}**

This example shows that the programmer can test the execution of main and the call to the function without having yet written the function to find the square root. This allows the pro- grammer to concentrate on one component of the program at a time. Although a stub is not really needed in this simple program, stubs are very useful for larger programs.

A **driver** is a module that tests a function by simply calling it. While one programmer may be working on the main function, another programmer may be developing the code for a particular function. In this case the programmer is not so concerned with the calling of the function but rather with the body of the function itself. In such a case a driver (call to the function) can be used just to see if the function performs properly.

#include <iostream>

#include <cmath>

using namespace std;

int findSqrRoot(int x); // prototype for a user defined function that

// returns the square root of the number passed to it

int main() {

int number;

cout << "Calling findSqrRoot function with a 4" << endl; cout << "The

result is " << findSqrRoot(4) << endl;

return 0; }

int findSqrRoot(int x) {

return sqrt(x);

}

**Codeblocks Debugger:**

The following wiki explains briefly about the debugging in codeblocks.

<http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks#Build_debug_version_of_your_project>

Basically, in any IDE you can build your application in debug mode and you can set up break points to stop the execution of your program and examine the contents of the variables. In this lab, you will be asked to debug your programs. Make sure to work on your prelab and visit the above web link.