More About WHILE Loops

http://people.sc.fsu.edu/~jburkardt/isc/week04_lecture_07.pdf

.....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 30 May 2011

More About WHILE Loops

- **Introduction**
- Counting with WHILE Statements
- Reading Input with WHILE Statements
- The DO...WHILE Statement
- The Fibonacci Sequence
- Lab Exercise #4

INTRO: Schedule

Read:

- Sections 3.7, 3.8, 3.9, 3.10

Next Class:

- FOR Loops

Assignment:

- Today: in-class lab exercise #4
- Thursday: Programming Assignment #2 is due.

INTRO: Today's Topics

Today we will spend some time looking at examples of how the **while** statement can be used.

We'll start by looking at how useful a *counter variable* can be, which keeps track of how many times the loop has been carried out.

Then we will consider three ways of using a **while** statement to add an unspecified number of values. The last way will also work for reading data from a file.

Our in-class lab exercise will have you use a **while** statement to compute some of the numbers in the Fibonacci sequence.

More About WHILE Loops

- Introduction
- **Counting with WHILE Statements**
- Reading Input with WHILE Statements
- The DO...WHILE Statement
- The Fibonacci Sequence
- Lab Exercise #4

COUNT: How Many Times Has This Loop Run?

The **while** statement doesn't know how many times it repeats the statements in the loop, unless you tell it.

The easiest way to do this is to define a *counter variable* of type **int**, perhaps called **step**, which is initialized to zero, and increased by 1 (or "incremented") at the beginning of the loop.

Sometimes the counter variable is useful as a label (we use it in the output of the popcorn calculation).

Also, since a **while** statement is a loop, you might accidentally have a loop which runs forever. If you limit the maximum number of steps to some large value, you will be able to stop the progr and report where the problem occurred.

COUNT: Watch for Too Many Iterations!

sqroot_check.cpp:

```
x = 700.0;
h = 700.0;
w = 1.0;
error = 700.0 - h * h;
step = 0;
while ( 0.000001 < fabs ( error ) )  <-- Seek small error.
{
  step = step + 1;
  if ( 100 < step )                  <-- Too many steps!
  {
    cerr << "Too many steps!\n";
    exit ( 1 );                      <-- Quit right now!
  }
  h = ( h + w ) / 2.0;
  w = x / h;
  error = 700.0 - h * h;
}
cout << "Square root estimate is " << h << "\n";
```

COUNT: Run the Loop Exactly N Times

Another way to use the counter variable is if you simply want to repeat the loop a specific number of times.

To run the loop 10 times, the **while** statement can simply check whether the value of **step** is less than 10.

COUNT: Add 7 Values

lunch.cpp:

```
day = 0;
total = 0.0;
while ( day < 7 )  <--  How many days done so far?
{
  day = day + 1;
  cout << "Enter amount for lunch on day " << day << ":
  cin >> lunch;
  total = total + lunch;
}
cout << "This week's lunch bill: " << total << "
```

COUNT: Seek a Certain Number of Wins

A different kind of counter is used something like a fishing limit, because each time you run the loop, you might, or might not, get something you want. You start out with nothing, and you want to carry out the loop repeatedly until you've had a given number of successes, when you can stop.

We could still keep a loop counter as well, to let us know how many times we tried.

In that case, we initialize the counter (let's call it **wins**) to zero, but inside the loop you only increase it when you win, or find what it is you are looking for.

COUNT: Play til You Win 10 Times

three_heads.cpp:

```
  int coin1, coin2, coin3, tries, wins;
  float cost;
//
//  Flip 3 coins, get three heads to win.
//
  tries = 0;
  cost = 0.0;
  wins = 0;
  srand ( time ( 0 ) );
  while ( wins < 10 )
  {
    tries = tries + 1;
    cost = cost + 0.25;  <-- Costs 25 cents a game.
    coin1 = rand ( );    <-- Random integers.
    coin2 = rand ( );
    coin3 = rand ( );
    if ( coin1 % 2 == 1 && coin2 % 2 == 1 && coin3 % 2 == 1 )
    {
      win = win + 1;
    }
  }
  cout << "Winning " << wins << " times, I spent $" << cost << "\n";
```

More About WHILE Loops

- Introduction
- Counting with WHILE Statements
- **Reading Input with WHILE Statements**
- The DO...WHILE Statement
- The Fibonacci Sequence
- Lab Exercise #4

INPUT: Add Unknown Number of Values

You might want to write an interactive program that computes a sum, but unlike the **lunch.cpp** program, you might want to use it in cases were more than 7 numbers are involved.

It is natural to have the program read a number, add it to the total, and try to read the next one, until something tells the program the summing is done and it's time to print the total.

It's easy to have the program read a number over and over; we simply put a **cin** statement inside a **while** loop.

But how do we tell the **while** loop when the input is complete?

INPUT: Three Ideas

Here are several ideas we could try so that we can write a program that will add as many numbers as we want:

1. the user first enters **n**, the number of values to be summed;
2. or, a special value of -1 (for instance) is used as a signal that the input is complete;
3. or, the user does something to signal that the input is done.

INPUT: 1: User Enters N

sum_counter.cpp

```
cout << "How many numbers to add?: ";
cin >> n;

sum = 0.0;
step = 0;

while ( step < n )
{
  cin >> x;
  sum = sum + x;
  step = step + 1;
}
cout << "Sum of " << n << " numbers is " << sum << "\n";
```

INPUT: 2: User Enters Special Value

sum_minus.cpp

```
cout << "Enter values, end with -1\n";

sum = 0.0;
n = 0;
cin >> x;                <-- Have to read once here!

while ( x != -1.0 ) <-- then check for -1
{
  sum = sum + x;     <-- then add
  n = n + 1;
  cin >> x;          <-- now get next number
}
cout << "Sum of " << n << " numbers is " << sum << "\n";
```

INPUT: End of File

The bad thing about using -1.0 as a special value is that we have to remember what the special value is, and we're in trouble if -1.0 is actually one of the numbers we want to add.

A better solution is available. The special character CTRL-D is called the **eof** or "end of file" signal. Instead of using a -1.0 value, we can use CTRL-D. This always works, and it will have an extra benefit!

If we just performed an input operation with **cin**, then the special function **cin.eof()** is **true** if the program detected an end of file, but **false** if we actually read a value.

So our while condition is changed to

```
while ( ! cin.eof ( ) )
```

INPUT: User Enters CTRL-D

sum_eof.cpp

```
cout << "Enter values, end with CTRL-D\n";

sum = 0.0;
n = 0;
cin >> x;                    <-- Have to read once here!

while ( !cin.eof ( ) ) <-- check for end of file
{
  sum = sum + x;             <-- then add
  n = n + 1;
  cin >> x;                  <-- now get next number
}
cout << "Sum of " << n << " numbers is " << sum << "\n";
```

INPUT: End-of-File Works for Files

I mentioned a mysterious "extra benefit" of using the **eof** function instead of a special input value like -1.

The benefit is that, if the program reads input from a file instead o from the user, when it reaches the end of the file, the **cin** command will get an **eof** signal just as though an interactive user had typed CTRL-D.

In other words, we can use the same program to sum the numbers in a file:

```
sum_eof < grades.txt
```

More About WHILE Loops

- Introduction
- Counting with WHILE Statements
- Reading Input with WHILE Statements
- **The DO...WHILE Statement**
- The Fibonacci Sequence
- Lab Exercise #4

DO/WHILE: Variation on the WHILE Statement

We have learned about the **while** statement, which allows us to repeat statements using the following form:

```
initialization;
while ( condition to check )
{
   statements to be repeated;
}
```

The standard **while** statement checks the condition before carrying out the statements.

We are using a **while** statement in our second homework programming assignment, for instance, to check whether we have reached the value 1.

DO/WHILE:

The **do...while** statement is almost the same as the **while** statement, except that the condition is checked **after** the statements are executed:

```
initialization;
do
{
   statements to be repeated;
}
while ( condition to check );    <-- semicolon required here!
```

DO/WHILE:

Since the **do...while** statement is so closely related to the **while** statement, in a sense, it's not really necessary, just convenient.

Moreover, to use the **do...while** statement, you have to be very careful to put a semicolon at the end of the final **while**. This means it's easy to use the statement incorrectly!

One feature of the **do...while** statement is that the statements inside the loop will *always be executed at least once*.

I don't need you to learn very much about the **do...while**, but I want you to be familiar with it.

Let's look at how some examples we've done before might be rewritten this way.

DO/WHILE: Square Root with WHILE

square_root_while.cpp:

```
h = x;
w = x / h;                  <-- Same as in loop
error = x - h * h;          <-- Same as in loop
while ( 0.001 < fabs ( error ) )
{
  h = ( h + w ) / 2.0;
  w = x / h;
  error = x - h * h;
}
```

DO/WHILE: Square Root with DO...WHILE

square_root_do_while.cpp:

```
h = x;                          <-- only one initialization

do
{
  w = x / h;
  error = x - h * h;
  h = ( h + w ) / 2.0;
}
while ( 0.001 < fabs ( error ) ); <-- Pesky semicolon!
```

More About WHILE Loops

- Introduction
- The Increment Operator
- Counting with WHILE Statements
- Reading Input with WHILE Statements
- The DO...WHILE Statement
- **The Fibonacci Sequence**
- Lab Exercise #4

FIBONACCI: Homework from 1202 AD

In the year MCCII, Leonardo Fibonacci published one of the earliest mathematics textbooks, in which he suggested that Europeans should try using the latest discovery from India and the Arabs, called the decimal system.

His book was full of examples of how to compute interest rates, balance accounts and convert currency using the decimal system.

One homework exercise in the book told the story of a boy who bought a pair of baby rabbits on New Year's Day. One month later, the rabbits were old enough to reproduce, and a month after that, a new pair was born.

A month after that, the first pair had another pair of babies, although the second pair was only just becoming old enough to reproduce, so now there were three pairs.

A month later, there were 5 pairs.

FIBONACCI: Homework from 1202 AD

Counting the pairs of rabbits each month, we have the sequence

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

Each month, the population increases by births to all the rabbits who have just turned two months old. Why did October's 21 pairs become November's 34? Because in September we had 13 pairs, and so two months later, we get exactly 13 new pairs added.

Once you see the pattern, it's easy to write a formula:

```
F(next month) = F(this month) + F(last_month);
              = population    + pregnant;
```

Of course, being mathematical rabbits, they always reproduce exactly on schedule, always have two babies, one of each sex, and never die!

FIBONACCI: Homework from 1202 AD

On paper, it's easy to compute the Fibonacci sequence.

But in a computer, what do we do? It's easy to see that we have to add two numbers to get the next one. But if we want to compute 100 values, does that mean we have to define 100 variables, called, perhaps, f1, f2, f3, ..., f100?

Certainly, we could do it that way, but it would mean we had to type a lot of declaration statements. And our program would be very long.

```
f1 = 1;
f2 = 2;
f3 = f2 + f1;
f4 = f3 + f2;
...
```

FIBONACCI: Homework from 1202 AD

The formula for the next value essentially says:

```
next = current + old;
```

That uses just three variables. We only need three variables, if we are willing to move the values around as we go!

```
                                Old Cur Nex
old = 1;                        (1   ?   ?)
current = 1;                    (1   1   ?)
next = current + old;           (1   1   2) <--Computed F3
old = current;                  (1   1   2)
current = next;                 (1   2   2)
next = current + old;           (1   2   3) <--Computed F4
old = current;                  (2   2   3)
current = next;                 (2   3   3)
next = current + old;           (2   3   5) <--Computed F5
```

FIBONACCI: Homework from 1202 AD

If you think about it, we have exactly the form of a **while** loop:

```
old = 1;                          <-- initialization
current = 1;
step = 0;
while ( condition? )              <-- repetition condition
{
  next = current + old;           <-- statements to repeat
  old = current;
  current = next;
  step = step + 1;                <-- Count the steps
}
```

The only thing missing is a repetition condition. We might continue as long as **step** is less than 10, for instance.

FIBONACCI: Homework from 1202 AD

The program we have outlined can compute an unlimited number of Fibonacci numbers, but it only uses three variables. If we had had to declare a variable for every single Fibonacci number, we could never have written this program!

An important point of this exercise is that a variable is just a place to store a result. As soon as you don't need that result any more, the variable can be re-used. Many computations involve stepping forward in a very long sequence, but really only need to keep track of the most recently calculated values.

You can carry out such a calculation using just a few variables; the price you pay, though, is that you have to (carefully!) move the data from the **next** variable to the **current** variable to the **old** variable yourself. And if you do this incorrectly, your program will fail.

More About WHILE Loops

- Introduction
- Counting with WHILE Statements
- Reading Input with WHILE Statements
- The DO...WHILE Statement
- The Fibonacci Sequence
- **Lab Exercise #4**

EXERCISE 1: Print the Fibonacci Sequence

Write a C++ program that can compute and print out the elements of the Fibonacci sequence up to the 20th number.

In particular, your first five lines of output should be

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

EXERCISE 1: Program Outline

```
# include <stuff>
using namespace std;
int main ( )
{
   declarations;
   initialization;
   print first values;
   while ( ? )
   {
      statements to repeat;
      print new value;
   }
   return 0;
}
```

EXERCISE 2: First Number Over 10,000

Make a new copy of your first program. Modify the program so that the user enters a number **n**, and your program computes the elements of the Fibonacci sequence until it finds the first element that is equal to or bigger than **n**.

Print the step and the Fibonacci number when this happens.

In particular, if the user types 100, your program might print out:

Fibonacci number 12 is 144 which is bigger than 100.

Once you get your program working, try to compute the first Fibonacci number bigger than 10,000. (When you enter the value 10,000 into the computer, don't use a comma!)

Once you have computed this value, please show your work to Detelina so you can get credit for the exercise!