The WHILE Statement

http://people.sc.fsu.edu/~jburkardt/isc/week03/lecture_05.pdf

.....

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

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 23 May 2011

The WHILE Statement

- **Introduction**
- A Review of Integration
- The WHILE Statement
- Using the WHILE Statement
- Exercise #3: Estimate a Square Root
- Extra: A Preview of Functions

INTRO: Repeat INTEGRATION

At the end of last week's lecture, I meant to go over some slides about approximating an integral before assigning our first homework program.

Instead, I presented the material using the whiteboard, and did not go over the details of the homework assignment either.

Today, I would like to outline the information carefully so that I am sure we can all see what we're trying to do, and how we are going to do it!

INTRO: WHILE Statements

For our new material today, we will talk about the **while** statement.

Like **if** and **for**, the **while** statement is an example of a **control** statement.

It controls whether or not executable statements are carried out.

It does this by checking a logical condition.

Once the statements are carried out, the **while** checks the condition again, and if it is still true, the statements are repeated.

The **while** statement allows us a natural way to express many computational algorithms that involve repetition.

INTRO: Schedule

Reading:

- Read Sections 3.7 to 3.9 of the text to understand more about the WHILE statement.

Next Class:

- The IF, ELSE and ELSEIF statements

Lab Exercise #3:

- at the end of today's class.

Programming Homework:

- Program 1 will be due on Thursday
- Program 2 will be assigned on Thursday.

The WHILE Statement

- Introduction
- **A Review of Integration**
- The WHILE Statement
- Using the WHILE Statement
- Exercise #3: Estimate a Square Root
- Extra: A Preview of Functions

INTEGRATION: Approximation

We have been given the formula for some function, **f(x)**, and the limits of an interval $x1 \leq x \leq x2$.

The integral of **f(x)** over this interval is the area between the **x**-axis and the curve. Calculus considers methods for finding the exact value...if the function is simple enough.

A computer user can always approximate an integral. This is done by approximating the area. We simply divide the interval **[x1,x2]** into **n** equal subintervals of width **dx**. Within each subinterval, we evaluate **f(x)** at some point. We have formed a rectangle of dimensions **dx** by **f(x)**.

To estimate the area under the curve, we add up the areas of these rectangles.

To improve the estimate, we try increasing the value of **n**.

INTEGRATION: Approximation

For your programming assignment, I have suggested you use the **left endpoint rule**, that is, that in each subinterval, you evaluate the function at the leftmost point.

In that case, you can use the C++ **for** statement, as follows:

```
dx = ( x2 - x1 ) / n;
value = 0.0;
for ( x = x1; x < x2; x = x + dx )
{
  value = ?;  <--  add the area of the next subtriangle
}
cout << "Using " << n <<
" subintervals, integral estimate is " << value << "\n";
```

INTEGRATION: Programming Assignment #1

Consider the function $f(x) = x^2 + 2x - 3$.

We wish to estimate the integral of $f(x)$ for $-4 \leq x \leq 5$.

The antiderivative function is $g(x) = \frac{x^3}{3} + x^2 - 3x$.

Therefore, the exact value of the integral is g(5)-g(-4) = 45.

Write a C++ program that estimates the value of this integral, using **n** intervals and the **left hand** approximation.

Run your program with values of **n** equal to 1, 2, 5, 10, 20 and 40.

Turn in: a copy of your program, and a table with 6 lines. Each line lists the value of **n**, your estimate for the integral, and the error of that estimate.

Your program and table are due by class time, Thursday, May 26. You can email it to Detelina or hand in a printed version.

The WHILE Statement

- Introduction
- A Review of Integration
- **The WHILE Statement**
- Using the WHILE Statement
- Exercise #3: Estimate a Square Root
- Extra: A Preview of Functions

WHILE: Repeat WHILE Condition Is True

Last week, we talked a little about logical conditions, and how an **if** statement can be used to control whether certain statements are carried out.

We also got an advance look at the **for** statement, which can repeat a set of statements.

Today, we will look at the **while** statement, which you can think of as a combination of those two control statements.

The **while** statement repeats a set of statements as long as some condition is true.

WHILE: Repeat WHILE Condition Is True

Suppose I'm driving a truck full of bricks over a bridge, but the bridge has a weight limit, which I don't know. If I exceed the weight limit, an alarm goes off and I can't cross.

Since I don't know the limit, I can toss a brick off the truck and try again. Unless the limit is very low, I will eventually be allowed to cross.

My strategy for crossing the bridge, therefore is

```
Try to cross the bridge.
If the alarm indicates my weight is too high
{
  decrease my weight by one brick.
  Try to cross again.
}
```

WHILE: Execute WHILE Condition Is True

Here's how to get Bill Gates to give you $1,000:
Start by betting $1,000 on the toss of a coin.
If you lose the toss, double your bet, and try again.

You have half a chance of winning $1,000 immediately. But if you lost, and you double, you have half a chance of getting it all back, plus the extra $1,000. If you lose again, you have half a chance of winning everything back, plus $1,000, and so on.

Our strategy is:

```
Bet $1,000.
Flip a coin.
if you lost the coin toss
{
  double your bet.
  flip again.
}
```

WHILE: Execute WHILE Condition Is True

C++ has a **while** statement that makes it easy to program these kind of operations, in which you might have to do something several times until a condition is no longer true.

The idea is that we have a logical condition, which we can express in numbers, and that we want to repeat a set of statements as long as this condition is true.

```
initial statements;
while ( condition )
{
  statement 1;
  statement 2;
  ...
  statement last; <-- now go back and check condition!
}
```

WHILE: The Bridge Weight Limit

For the bridge crossing problem, let's say the truck weighs 3,000 pounds, carries 1,000 bricks weighing 5 pounds each, and the bridge weight limit is 7,500 pounds.

We could express our strategy for satisfying the limit this way:

```
limit = 7500;
truck = 3000;
bricks = 1000;

while ( limit < truck + bricks * 5 )  <-- too heavy?
{
  bricks = bricks - 1;
}
```

WHILE: An Impossible Bridge

Now in some cases, the limit is so low that the truck can't pass. Our old program would compute negative bricks!
The **exit** function from <**cstdlib**> will let us stop at once.

```
limit = 2500;    <-- No way we can make it!
truck = 3000;
bricks = 1000;
while ( limit < truck + bricks * 5 )
{
  bricks = bricks - 1;
  if ( bricks < 0 )  <-- We are in trouble!
  {
    cerr << "No solution!\n";  <-- Warn the user!
    exit ( 1 );                <-- Terminate now!
  }
}
```

Notice that the **if** and **while** statements are **nested**.

WHILE: Betting with Bill

For our Bill Gates bet, we can use the **rand()** function from <**cstdlib**> to return a random integer.
If the integer is odd, we'll call that "heads" and say we won.

```
srand ( time ( NULL ) );  <-- Scrambles the random numbers.
won = 0;
bet = 1000;               <-- Our initial bet.
coin = rand ( );          <-- First toss
while ( coin % 2 == 0 )   <-- Check the condition
{                             Remember, TWO equal signs!
  won = won - bet;        <-- We lost some more money
  bet = 2 * bet;          <-- Double our bet.
  coin = rand ( );        <-- Try again!
}
won = won + bet;          <-- We won.  $1,000, in fact!
```

WHILE: Is N Prime?

An integer **n** is prime if it can't be divided by any smaller integer except 1. Can a program seek a divisor of a given number **n**?

```
cout << "Enter integer to check: ";
cin >> n;

test = n - 1;   <-- First number to try.
divisor = 0;    <-- Save divisor here, if found.

while ( what? )
{
   Did we find divisor?...
   Change test...
}
cout >> "What?\n";
```

How would you solve this by hand?

The WHILE Statement

- Introduction
- A Review of Integration
- The WHILE Statement
- **Using the WHILE Statement**
- Exercise #3: Estimate a Square Root
- Extra: A Preview of Functions

EXAMPLE: The SQRT Function

In C++, if you include the <**cmath**> file, you can get the square root of a real number **x** by typing:

```
y = sqrt ( x );
```

To check, you can compute the error:

```
error = x - y * y;
```

Actually, we should get the **absolute value** of the error, using the **fabs()** function, which is also part of <**cmath**>:

```
error = fabs ( x - y * y );  <-- Result 0 or positive.
```

If we try a few examples, we see the error is very small!

EXAMPLE: Where do SQRT values come from?

How does a computer know the square root of any number you type in? It cannot use a list or table of values.

It actually computes the square root. How is this possible?

Actually, it's similar to division. When you divide 10 by 7 you get 1.4285714285... and as you compute the next digit, your answer gets more accurate. We stop when the number of correct digits satisfies us.

The computer has a similar procedure for computing

```
sqrt(700) = 26.4575131...
```

producing a sequence of *approximations* to the correct answer.

EXAMPLE: Squaring a Rectangle

Suppose we have a rectangle of area **A** = 700 square inches

One example of such a rectangle would have width **W**=10, in which case we need height **H**=700/**W**=70.

But we wanted a **square** that had area 700, what would be the value of **W**, the width of one side? If 10*70 = 700, then 10*10 is too low, and 70*70 is too high. Even though they are wrong, let's remember these values as **W0** and **H0**.

If we don't know the exact answer, why not take a guess, and pick a number between 10 and 70. One good guess is the average. We'll call this new guess W1:

```
W1 = ( W0 + H0 ) / 2 = ( 10 + 70 ) / 2 = 40.
```

What's the error we are making?

```
error = | 700 - W1 * W1 | = | 700 - 1600 | = 900.
```

EXAMPLE: Squaring a Rectangle

Now, even if W1 isn't the correct value, we could still make a rectangle with area 700, and one side W1; we just have to set the height to 700 / W1:

```
H1 = A / W1 = 700 / 40 = 17.5
```

So we still have a rectangle, of dimensions 40 x 17.5, but...it's less rectangular.

What if we try again? We can average **W1** and **H1** to get **W2**, and then divide 700 by **W2** to get the new height **H2**.

```
W2 = ( W1 + H1 ) / 2 = ( 40 + 17.5 ) / 2 = 28.75.
H2 = 700 / 28.75 = 24.3478...
error = | 700 - W2 * W2 | = 126.56...
```

EXAMPLE: Squaring a Rectangle

This is still much too much error! What will our third step be?

```
W3 = ( W2 + H2 ) / 2;
H3 = 700 / W3;
error = | 700 - W3 * W3 |
```

and we will want to try keep going, **while** the error is too big.

I hope you can see that this can be turned into a program!

EXAMPLE: Squaring a Rectangle

```
include statements and other stuff!
int main ( )
{
  declarations;
  area = 700;  w = 70;  h = 1;
  error = area - w * w;
  while ( 0.000001 < fabs ( error ) )
  {
    compute new w;
    compute new h;
    error = area - w * w;
  }
  cout << "Estimated square root is " << w << "\n";
  return 0;
}
```

The WHILE Statement

- Introduction
- A Review of Integration
- The WHILE Statement
- Using the WHILE Statement
- **Exercise #3: Estimate a Square Root**
- Extra: A Preview of Functions

EXERCISE: Square Root

You are going to write a program to estimate the square root of a number, using the ideas we have talked about today.

We will want to use real arithmetic for our computations, so make sure you declare your variables to be of type **float**.

If you want more accurate results, you can try declaring variables to be **double** instead, and using a smaller error tolerance!

If you want to see more digits in a print out, include <**iomanip**> and try something like

```
cout << setprecision(12) << w << "\n";
```

EXERCISE: Outline of procedure

The user inputs a number x:

```
set w equal to 1.
set h equal to x / w;
set error = fabs ( x - w * w );

WHILE error is bigger than 0.0001:
  set w to the average of w and h.
  set h to x / w.
  set error to fabs ( x - w * w );

at the end, print
  x
  w,              your estimated square root;
  sqrt ( x ),     the computer's value.
  x - w * w,      the error using your estimate.
```

EXERCISE: Outline of program

```
# include <?>
using namespace std;
int main ( )
{
   declarations
   cout << "Enter X!";
   cin >> x;
   initializations
   while ( ? )
   {
      repeated actions
   }
   print out
   return 0;
}
```

EXERCISE: Test Data

Try your program using the value x = 700.

You will have to include <**cmath**> in order to use the **sqrt()** and **fabs()** functions.

If your program doesn't return an answer quickly, it is probably stuck in the **while** loop. Hit **Ctrl-C** to stop it. (Hold down the **Ctrl** key and hit **C**.) Ask me or Detelina to take a look.

When your program seems to be getting a good approximate answer, let Detelina know so she can give you credit for the exercise.

The WHILE Statement

- Introduction
- A Review of Integration
- The WHILE Statement
- Using the WHILE Statement
- Exercise #3: Estimate a Square Root
- **Extra: A Preview of Functions**

FUNCTIONS:

We've been approximating the integral of a function.

The example we used was $f(x) = x^2 + 1$. Inside the **for** loop, we multiplied the interval width by the function value using the statement:

```
value = value + dx * ( x * x + 1 );
```

But after class, I was asked *is it possible to write the function symbolically?*

The answer is **yes!**, but again, this is something we haven't had time to learn about yet. But I've been trying to hint that our C++ program, which has a **main()** function, can have more parts, also called functions, but behaving more like the functions of algebra, where they take input and return a value.

FUNCTIONS: Replace Formula by Function

So even though, in a sense, it's too soon, let me show you how we could use a function to represent $f(x)$.

In fact, the function looks just like a **main** program except that it has input, has the name **f**, and returns a **float** value. We write:

```
float f ( float x )
{
  float y;
  y = x * x + 1;
  return y;
}
```

Notice that the variable **x**, which is an input argument, is actually declared within the parentheses that list input.

The variable **y** is used to store the result of the function.
We send this back using the **return** statement.

FUNCTIONS: Changes Needed to use F(X)

We change our program in a few ways:

- We have to "declare" the function in the beginning of our program;
- We replace the formula by **f(x)**;
- We add the text of the function after the main function.

By the way, the same variable name **x** is used in the main program and in the function **f**, but that is not necessary.

Moreover, all the variables declared "inside" the function **f** are "invisible" to the main program. The only thing the main program knows is that there is a function called **f**, and it takes a float as input and returns a float as its value.

FUNCTIONS: Integrate Using F(X)

```
# include <iostream>
using namespace std;
float f ( float x );  <-- This "declares" our function.
int main ( )
{
  ...I left out some stuff here...
  for ( x = x1; x < x2; x = x + dx )
  {
    value = value + dx * f ( x );
  }
  cout << "Estimated integral = " << value << "\n";
}
float f ( float x )
{
  ..Text of function...
}
```