Arithmetic, Logic, and Integration

http://people.sc.fsu.edu/∼jburkardt/isc/week02/lecture_04.pdf

....

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

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 20 May 2011

Arithmetic, Logic, and Integration

INTRO: Arithmetic Rules

In the previous lecture, we introduced the idea of variables, which were quantities with a **name**, a **type**, and a **value**.

Variables were set up with declaration statements, such as:

```
int x = 17;
float work, force, distance;
```

Values could be assigned as part of the declaration, or set later by an assignment statement.

A typical assignment statement has the form

```
x = y + z;
work = force * distance;
```

Today we will look at some arithmetic rules for the formulas used on the right hand sides of assignment statements.

INTRO: Logic

The variables we have discussed so far were numeric. The C++ numeric types we have discussed include **int**, **float** and **double**.

Sometimes, we want a variable to be able to store a logical value, that is, to hold a value of **true** or **false**.

C++ includes the bool type for this purpose.

We will look at how logical variables can be declared, assigned, and used.

In particular, we will introduce the **if** statement, which checks whether a certain condition is true.

INTRO: Integration

From calculus, we know that the integral of a function **f(x)** from **a** to **b** can be approximated by thinking of the integral as the area under the curve.

So we can divide the interval **[a,b]** into subintervals, and using each subinterval as a base, draw a rectangle that intersects the curve, perhaps using the left or right endpoint, or the midpoint of the subinterval, to determine how high to draw the rectangle.

Such a process can be automated. We will look at several ways to estimate an integral this way.

Your homework programming assignment will require you to compute a sequence of integral estimates, using more and more subintervals.

INTRO: 3D Plots

For our lab exercise on Tuesday, we made a simple plot with Gnuplot.

I was asked if we would also be doing 3D plots, and I said we would, eventually.

However, if you are interested in looking into this right now, I have put up a simple example on the web page.

The file **surface_grid.cpp** will compute a table of values Z(X,Y), which you can redirect to a file, say "data.txt".

If you start gnuplot, you can display the surface with the command

```
    splot "data.txt" with lines
```

Arithmetic, Logic, and Integration

ARITHMETIC: Assignment

A typical arithmetic calculation in C++ might be:

```
z = x + 42 * y;
```

C++ calls this line an example of an **assignment statement**.

It has the form

```
left value = right value;
```

where *left value* is the name of a variable, and *right value* is a literal number, or a variable, or some kind of formula.

An assignment statement is an example of an **executable statement**, that is, it describes an action to be carried out.

ARITHMETIC: Variable Values

We saw last time that variables must be set up in declaration statements at the beginning of the program.

We saw that it was possible, but not required, to give a variable a starting value as part of its declaration.

What about variables which are not given an initial value this way? An assignment statement is one way to put a value into a variable. The assignment statement overwrites any previous value stored in the variable.

Another way for a variable to receive a value is for the user to input the value. This is done with the **cin** input operator:

```
    cout << "Enter the value of x: ";
    cin >> x;
```

ARITHMETIC: Example Program

This example is called **formula.cpp**:

```
# include <iostream>
using namespace std;
int main ( )
{
  int x = 12;  <-- 3 Declaration statements
  int y;
  int z;

  cout << "Enter the value of y: ";
  cin >> y;
  z = x + 42 * y;
  cout << "Z = " << z << "\n";
  return 0;
}
```

ARITHMETIC: Variable Values

Notice that the variables **y** and **z** don't have a value when the program begins. They are **uninitialized**.

There's nothing wrong with an uninitialized variable, but until it gets a value, *you are not allowed to use it on the right hand side of a formula*.

If you try to do this anyway, you get an error. The compiler might see the error, or else it will occur when you run the program. Let's modify the previous program and see what happens!

ARITHMETIC: Example Program

This example is called **formula2.cpp**:

```
# include <iostream>
using namespace std;
int main ( )
{
  int x = 12;
  int y;
  int z;

  z = x + 42 * y;  <-- Y doesn't have a value yet.
  cout << "Enter the value of y: ";
  cin >> y;
  cout << "Z = " << z << "\n";
  return 0;
}
```

ARITHMETIC: Output

Here is what I got when I ran the modified program:

```
OSX> ./a.out
Enter the value of y: 4
x = 12
y = 4
z = 12 = 12 + 42 * 4
OSX>
```

The final printout doesn't make sense until you realize that, when the formula was actually evaluated, **y** must have had the value 0, not 4. In other words, the g++ compiler is silently initializing variables to 0 for us.

Not every compiler does this, and you should not rely on it! But in any case, remember that a formula is computed using whatever values the variables happen to have at the moment the assignment statement is executed.

ARITHMETIC: What Are the Values?

Try to fill in the values in the variables following each statement. The variables are not initialized, so they start with values of “?”:

```
                X       Y       Z
initial:     |__?___|___?__|___?__|
x = 1;       |______|______|______|
z = x + y;   |______|______|______|
y = 3;       |______|______|______|
z = x + y;   |______|______|______|
x = x + 1;   |______|______|______|
z = x + y;   |______|______|______|
y = 2 * y;   |______|______|______|
z = x + y;   |______|______|______|
```

ARITHMETIC: Output

Here is how you should have filled out the table:

```
                  X         Y        Z
initial:      |   ?    |    ?   |    ?   |
x = 1;        |   1    |    ?   |    ?   |
z = x + y;    |   1    |    ?   |    ?   |   <-- Compiler might
y = 3;        |   1    |    3   |    ?   |       complain.
z = x + y;    |   1    |    3   |    4   |
x = x + 1;    |   2    |    3   |    4   |
z = x + y;    |   2    |    3   |    5   |
y = 2 * y;    |   2    |    6   |    5   |
z = x + y;    |   2    |    6   |    8   |
```

ARITHMETIC: Operators

Simple arithmetic involves addition, multiplication, and so on. How do we specify these operations in a C++ program?

Addition:	people = children + adults;
Subtraction:	workdays = 365 - holidays - sundays;
Multiplication:	celsius = (fahrenheit - 32) * 5.0 / 9.0;
Division:	average = (x1 + x2 + x3) / 3;
Remainder:	leftover = 365 % 7;
Power:	area = pi * pow (radius, 2);

The expression **365 % 7** returns the remainder when 365 is divided by 7, namely 1. A integer **n** is even if **(n % 2)** is 0, and it is odd if **(n % 2)** is 1.

The expression **pow (radius, 2)** means $radius^2$. To use **pow()**, you must include <**cmath**>. We will wait til later to talk more about the rules for the power function.

ARITHMETIC: Integer versus Real Division

C++ makes a distinction between integer and real variables.

C++ also assumes that the result of the division operation depends on the type of the variables used.

In the expression **a/b**, if the top and bottom are both literal integers, integer variables, or formulas involving integer variables, then the result will be rounded down to an integer value!

The expression **20 / 3** contains literal integers. It evaluates to 6. The expressions **20.0 / 3.0** , **20.0 / 3** , and **20 / 3.0** all contain at least one float, and hence evaluate to 6.666...

Similarly, **1/2** results in the value 0!

We will later see how to guarantee a real number result from division when we want one.

ARITHMETIC: Precedence

Sometimes when we write a complicated formula, the result could depend on the order in which we carry out the operations.

Even a simple formula like z = x + 42 * y could be evaluated two different ways:

```
Do the addition first:   ( x + 42 )
Then the multiplication: ( ( x + 42 ) * y )

Do the multiplication first: ( 42 * y )
Then addition:               ( x + ( 42 * y ) )
```

To avoid problems, there is a rule for how operators are evaluated.

ARITHMETIC: Precedence

Operators in parentheses **()** are evaluated first. In cases of nested parentheses, the innermost pair of parentheses is evaluated first.

Multiplication *****, division **/**, and the modulus or remainder function **%** are applied next. If there are several such operators, they are evaluated from left to right.

Addition **+** and subtraction **-** are applied last, and are evaluated from left to right.

The most common thing people have trouble with is writing a fraction correctly.

These rules will need to be extended later when we meet some new operators.

ARITHMETIC: Precedence Examples

What happens, supposing a = 10, b = 20, c = 30 and d = 40? This program is **precedence.cpp**:

```
m = a + b * c + d / 2;

n = ( a + b * c ) + d / 2;

o = ( a + b ) * ( c + d ) / 2;

p = ( a + b * ( c + d ) ) / 2;

q = a + ( b * c ) + ( d / 2 );
```

ARITHMETIC: Evaluate a polynomial

If we want to evaluate a polynomial $ax^2 + bx + c$, the rules of precedence allow us to write:

```
value = a * x * x + b * x + c;
```

Another legal form is:

```
value = a * pow ( x, 2 ) + b * x + c;
```

Surprisingly, one more correct form is:

```
value = ( a * x + b ) * x + c;
```

ARITHMETIC: Evaluate a polynomial

Now suppose we want to write the fraction $\frac{a(b-c)}{d(e+f)}$.

Then which C++ statements below are correct?

```
1:     a *    b - c      /    d *    e + f     ;   _________?
2:     a * ( b - c )     /    d * ( e + f )    ;   _________?
3:     a * ( b - c )     / (  d * ( e + f ) ); _________?
4:  ( a * ( b - c ) ) / (  d * ( e + f ) ); _________?
5:  ( a * ( b - c )     /    d * ( e + f ) ); _________?
```

ARITHMETIC: Evaluate a polynomial

Suppose we want to write the fraction $\frac{a(b-c)}{d(e+f)}$.

Then which C++ statements below are correct?

```
1:     a *   b - c     /   d *   e + f     ;  _Wrong__!
2:     a * ( b - c )   /   d * ( e + f )   ;  _Wrong__!
3:     a * ( b - c )   / ( d * ( e + f ) );  _OK_____!
4:   ( a * ( b - c ) ) / ( d * ( e + f ) );  _OK_____!
5:   ( a * ( b - c )   /   d * ( e + f ) );  _Wrong__!
```

Arithmetic, Logic, and Integration

LOGIC: The IF Statement

So far our C++ programs executed statements **unconditionally**, in order. But an intelligent program might want to check before some operations.

To divide **x** by **y**, I write:

```
z = x / y;
```

But **y** might be 0. So perhaps I might prefer to write

```
if y is not 0.0, then
   z = x / y;
```

or, even better,

```
if y is not 0.0, then
   z = x / y;
else
   z = 0;
```

LOGIC: Logical Comparisons

C++ actually has **if** and **else** statements, and they can be used to check a condition before carrying out statements.

Given numbers **x** and **y**, C++ can check these conditions:

`if ( x == y )...`	*if x is equal to y then...*
`if ( x != y )...`	*if x is not equal to y then...*
`if ( x <  y )...`	*if x is less than y then...*
`if ( x <= y )...`	*if x is less than or equal to y then...*
`if ( x >  y )...`	*if x is greater than y then...*
`if ( x >= y )...`	*if x is greater than or equal to y then...*

Be careful to avoid a statement like **if (x = y)** *with just one equal sign. It looks like you are checking for equality, but it's a mistake, and what's worse, the compiler might not complain about it!*

LOGIC: if (condition) statement

A comparison or condition is either **true** or **false**. An **if** statement allows us to check a condition first, and then to carry out one or more statements only if the condition is true.

```
if ( condition ) a single statement;

if ( condition )
{
  statement 1;    <-- Only if condition is true
  statement 2;
  ...
}
```

Even if I only have one statement, I always use the second form!

LOGIC: if/else statement

A combined **if/else** statement lets us do one thing if a condition is true, and something else otherwise.

```
if ( condition )
{
   statement 1 for true case;
   statement 2 for true case
   ...
}
else
{
   statement 1 for false case;
   statement 2 for false case
   ...
}
```

There are also more complicated versions, called **if/elseif/else**

LOGIC: Examples:

Suppose we have picked a random point in the plane, which has coordinates **(x,y)**. The point is inside the unit circle if, and only, if, the square root of the sum of the squares of **x** and **y** is less than or equal to 1.

We can express that test this way:

```
if ( sqrt ( x * x + y * y ) <= 1.0 )
{
  cout << "(" << x << "," << y
       << ") is in the unit circle.\n";
}
```

Of course, I've had to surprise you yet again. **sqrt()** is how we compute the square root function in C++. To use this function, you must include <**cmath**>.

LOGIC: A Circle Program

This is the code CIRCLE.CPP

```
# include <iostream>
# include <cmath>
using namespace std;
int main ( )
{
  float x, y;
  cout << "Enter (x,y): ";
  cin >> x >> y;
  if ( sqrt ( x * x + y * y ) <= 1.0 )
  {
    cout << "(" << x << "," << y
         << ") is in the unit circle.\n";
  }
  return 0;
}
```

LOGIC: Check Your Input!

By the way, the command **cin** $>>$ **x** $>>$ **y** will read **x** and **y**, but only if you enter them the way the computer expects to see them, that is, as two numbers separated by spaces.

What happens if you enter:

```
4 3     <--- This is fine!
4, 3
(4 3)
4. 3
4.0 3.0
3+1 8-5  <--- Really not what you'd expect!
```

This is why the program I entered on the web prints out **(x,y)** even if it is not in the circle, so the user has a chance to notice if something went wrong with the input.

Arithmetic, Logic, and Integration

- Introduction
- Arithmetic Rules
- Logic
- **Marching**
- Integration
- Conclusion
- Programming Assignment #1

MARCH: Taking Steps

In many scientific applications, it is necessary to compute a sequence of equally spaced values.

We might be:

- plotting a function at equally spaced values of **x**;
- computing a differential equation at successive values of **t**;

Last week, we needed to "march" from 0 to 20 by steps of 0.04, to make a table of values of $\sin x$.

When we are marching, we know where we start **x1**, where we want to finish **x2**, and how many steps to take **n** or the size of the steps **dx**.

MARCH: The for Statement

We still don't officially know about the **for** statement, but we need to use it for our first programming assignment.

```
for ( x = initial value; check x; increment x )
{
  statements;
}
```

- The initial value initializes the counter variable **x**.
- The check is a logical condition that must be true in order to continue.
- If the check is satisfied, the statements are carried out, and then the counter is incremented.

MARCH: The FOR Statement

Case 1: Suppose we know the limits **x1** and **x2**, and the size of the steps **dx**.

Then our march will take the steps:

```
x1, x1+dx, x1+2dx, x2+3dx, ...
```

and we keep stepping as long as we are less than or equal to **x2**.

The following **for** statement will work:

```
for ( x = x2; x <= x2; x = x + dx )
{
  y = cos ( x );
  cout << x << "  " << y << "\n";
}
```

MARCH: The FOR Statement

Case 2: Suppose we know the limits **x1** and **x2**, and the number of steps **n**.

Unless **n** is 1, we can always compute **dx** and go back to Case 1:

```
dx = ( x2 - x1 ) / ( n - 1 );
```

Case 3: Otherwise, we can use an integer step counter **i**.

```
x = x1;
for ( i = 1; i <= n; i++ )
{
  y = cos ( x );
  cout << x << "  " << y << "\n";
  x = x + dx;
}
```

MARCH: The FOR Statement

Case 4: If we know the number of steps, then instead of adding **dx** repeatedly, we can compute the **i**-th **x** directly.

```
for ( i = 1; i <= n; i++ )
{
  x = ( ( n - i ) * x1 + ( i - 1 ) * x2 ) / ( n - 1 );
  y = cos ( x );
  cout << x << "  " << y << "\n";
}
```

MARCH: 500 points or 501 points?

Often, you want your **x** values to be "nicely spaced".

For the sine table we made last week, I first said I wanted 500 values between 0 and 20, but then I changed that to 501. Why was that? The formula for **dx** from **n** is

```
dx = ( x2 - x1 ) / ( n - 1 );
```

so using **n** $= 500$ results in a spacing of $20/499$, whereas 501 points will be spaced by $20/500 = 0.04$, a "nice" value.

In a similar way, if we want to divide an interval up into **n** subintervals, then we typically define **n+1** points to do this. To divide [0,1] up into 10 subintervals, we use the 11 points 0, 0.1, 0.2, ..., 0.9, 1.0.

Arithmetic, Logic, and Integration

INTEGRATION:

I have spent so much time on marching because it comes up in many different ways in scientific computing.

Today, we will need marching because we are going to try to estimate an integral. You may recall that the integral of a function **f(x)** between **a** and **b** is the area under the curve.

If **f(x)** is a polynomial or a simple function, there may be a formula for the area. But in scientific computing, we face a wide variety of functions, and we will settle for a method that approximates lots of integrals rather than getting the exact values for just a few.

INTEGRATION:

One way to estimate the integral of a function starts by dividing the interval **[a,b]** into **n** equal subintervals, each of width **dx**, picking a point **x** in each subinterval, computing the value of **f(x)** at these points, summing the values and multiplying by **dx**.

For this simple procedure, we can distinguish three common choices for **x**:

- left rule: each subinterval uses its left endpoint;
- center rule: each subinterval uses its midpoint;
- right rule: each subinterval uses its right endpoint;

INTEGRATION: Example

Just to be clear about this procedure, let's estimate the integral of x^2 from 1.0 to 3.0, using **n**=4 intervals and the left hand rule:

Our interval spacing is dx = (x2 - x1) / n = (3.0 - 1.0) / 4 = 0.5;

Our intervals are [1.0,1.5], [1.5,2.0], [2.0,2.5], [2.5,3.0].

INTEGRATION: Example

Let's estimate the integral of x^2 from 1.0 to 3.0, using **n**=4 intervals and the left hand rule:

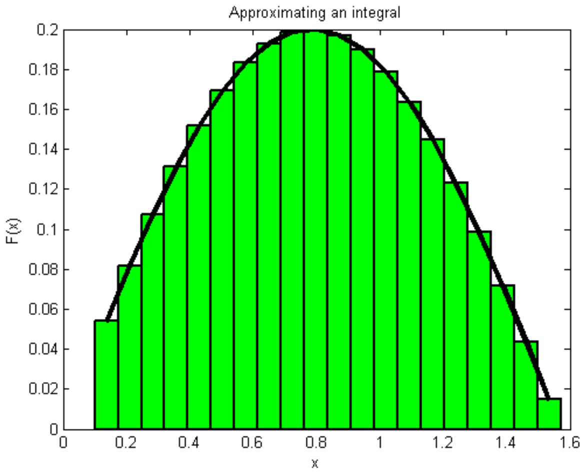
Our spacing is dx = (x2 - x1) / n = (3.0 - 1.0) / 4 = 0.5;

```
Interval      Left Point      F(x)         F(x)*dx

[1.0,1.5]            1.0      1.0          0.5
[1.5,2.0]            1.5      2.25         1.125
[2.0,2.5]            2.0      4.0          2.0
[2.5,3.0].           2.5      6.25         3.125
                                           -----
                                           6.750
```

The exact value is 8.666..., so our error was about 1.916. Presumably, we could do better with more intervals!

INTEGRATION: Using More Intervals Helps!



INTEGRATION:

Suppose I want to approximate the integral of the sine of x from x1=2 to x2=8, using n=60 intervals.

The distance I am traveling is x2 - x1 = 6.

The width of each interval, therefore, is dx = (x2 - x1) / n = 0.1.

The first interval goes from 2 to 2.1, the second from 2.1 to 2.2, and so on until the 60th interval, from 7.9 to 8.0.

If I use the left rule, my x's run from 2, 2.1, 2.2, up to 7.9.
If I use the center rule, my x's run from 2.05, 2.15, 2.25, to 7.95.
If I use the right rule, my x's run from 2.1, 2.2, 2.3, up to 8.0.

The only tricky thing is keeping track of your **x** values!

INTEGRATION:

Suppose I know **x1** and **x2**;

Suppose the user inputs the value of **n**,

From **x1**, **x2** and **n** I can compute **dx**.

Then initialize **value** to 0;

for **x** starting at **x1**, up to but not including **x2**, incrementing by **dx**
......compute the sine of **x**, multiply it by **dx**, and add it to **value**.

print **value**;

Can we sketch out a program on the board that would do this?

INTEGRATION: Variations

If we want to use the center rule, we start **x** at **x1+dx/2**.

If we want to use the right rule, we start **x** at **x1+dx** and we integrate up to, and including, **x2**.

I have shown you **for** loops in which **x** was used as the marching variable. Sometimes it is better to use an integer **i**, and to figure out the value of **x**. I have already shown you some ideas about how that works.

Arithmetic, Logic, and Integration

Conclusion:

We've been introduced to the basic arithmetic operators, and seen that parentheses can be helpful in correctly expressing formulas.

We've seen how the C++ **if** and **else** statements can be used to control whether we execute a group of statements, based on a logical condition.

We've talked about how to use the **for** statement to march through a set of values.

We've seen how an integral can be approximated using a sum that can be computed with a **for** statement.

Again, I apologize for making you see and use the **for** statement prematurely. We will get a chance to look at it more carefully very soon!

Conclusion:

Read: Chapter 2.5 to 2.8.

Next Class: The **if** and **while** Control Statements

Assignment: Today's Programming Assignment #1 is due in one week, on Thursday, May 26.

You are allowed to ask me or Detelina for advice on the programming assignment. You can ask anybody C++ questions, but the work on your assignment should be your own.

Arithmetic, Logic, and Integration

- Introduction
- Arithmetic Rules
- Logic
- Marching
- Integration
- Conclusion
- **Programming Assignment #1**

PROGRAM #1: Estimate an Integral

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.