The Math Library, Functions and Parameters

http://people.sc.fsu.edu/~jburkardt/isc/week05_lecture_10.pdf

.....

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

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 06 June 2011

The Math Library, Functions and Parameters

- **Introduction**
- The <cmath> Functions
- User Functions
- The CHOOSE Function
- The Month Length
- Homework Program #4

INTRO: Schedule

Read:

- Today's class covers Sections 5.1, 5.2, 5.3, 5.4, 5.5

Next Class:

- Function prototypes
- C++ Standard Library headers
- Random Numbers

Assignment:

- Programming Assignment #3 is due today.
- Programming Assignment #4 will be due June 16.

Functions

- Introduction
- **The <cmath> Functions**
- User Functions
- The CHOOSE Function
- The Month Length
- Homework Program #4

<CMATH>:

In several programs that we have looked at, I've told you to add the following line at the beginning:

```
# include <cmath>
```

That was because we wanted to use one of the C++ built-in mathematical functions, in particular the square root function **sqrt()** or the absolute value function **fabs()**:

```
float x, y, z;

x = 700.0;
y = sqrt ( x );
error = fabs ( x - y * y );
```

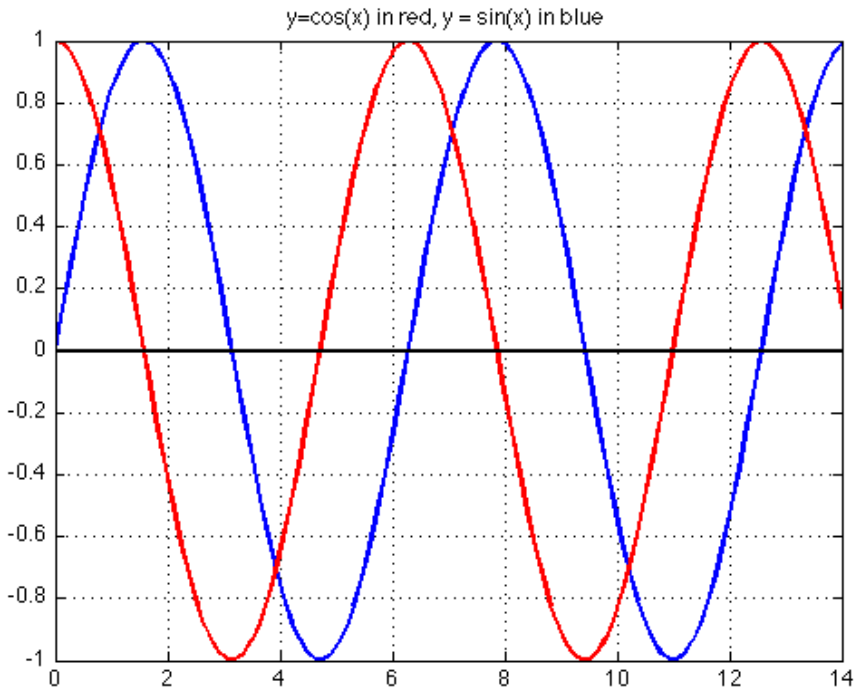
<CMATH>:

We will be needing other functions from <**cmath**> soon, so here is a list of some of the more useful things there:

```
y = f(x)                    x = inverse function(y)
--------------------------  ------------------------
y = cos(x);                 x = acos(x);
y = sin(x);                 x = asin(y);
y = tan(x);                 x = atan(y);
y = tan(height/width);      x = atan2 ( height, width );

y = exp(x);                 x = log(y);
y = pow(10.0,x);            x = log10(y);
y = sqrt(x);                x = y * y = pow ( y, 2.0 );
y = pow(x,power);           x = pow(y,1.0/power);
y = fabs(x);
y = ceil(x);
y = floor(x);
```

<CMATH>: Plots of Sin(x) and Cos(x)



<CMATH>: Trig Functions

The trig functions measure angles in radians. To convert an angle from degrees to radians, divide by 180 and multiply by π.

To compute the cosine of 30 degrees, write

```
degrees = 30.0;
radians = degrees * pi / 180.0;
y = cos ( radians );
```

The **arctan2()** function will report the angle corresponding to a given slope. If a road rises 1000 feet over a distance of 1 mile (5,280 feet), then the angle is **atan2 (1000.0, 5280.0)**. The angle will be reported in radians, so if we want degrees, we divide by π and multiply by 180:

```
radians = atan2 ( 1000.0, 5280.0 );
degrees = radians * 180.0 / pi;
```

which works out to about 10.7 degrees.

<CMATH>: Exponentials and Powers

The **exp()** function is used to model exponential growth. It is sometimes written $y = exp(x) = e^x$.

The **log10()** function uses 10 as the base. The **log()** function uses e as the base.

The **pow(base,power)** function requires two arguments, and computes x^y. Some combinations of base and power are illegal, because they represent meaningless quantities such as 0^{-1} or $(-4.0)^{\frac{1}{2}}$

<CMATH>: Remaining Functions

We have seen that the **fabs()** function returns the absolute value. **fabs(-17.3)=fabs(17.3)=17.3**.

The **ceil()** function rounds numbers "upwards". While **ceil(4.9)** is 5.0, you may be surprised that **ceil(-4.9) = -4.0**. Rounding "up" means rounding towards positive infinity!

Similarly, **floor()** rounds numbers "downwards", that is, towards negative infinity. **floor(4.9)=4.0** and **floor(-4.9)=-5.0**.

<CMATH>: Arguments Should be Real

The numbers you send into these functions (that is, the "arguments") should always be real variables (**float** or **double**), or real constants (numbers that have a decimal point).

To compute the square root of 225, you must type

```
y = sqrt ( 225.0 );
```

To compute the square of 15, you must type

```
y = pow ( 15.0, 2.0 );
```

<CMATH>: Arguments Should be Real

What exactly goes on when you "include" <**cmath**>?

Essentially, <**cmath**> is a list of *function declarations*, similar to variable declarations, but with extra information. A function h as a type, because it returns a value, but also has input variables, whose number and type must be declared.

Instead of

```
double x;
```

we have things like

```
double sin ( double x );    <-- one double in
double pow ( double x1, double x2 );  <--two doubles in
```

so the include statement gives the rules for how you are allowed to use the <**cmath**> functions.

Functions

- Introduction
- The <cmath> Functions
- **User Functions**
- The CHOOSE Function
- The Month Length
- Homework Program #4

USER:

You may have noticed that the list of functions available in <**cmath**> does not include a function to compute the maximum value of two numbers. You can always do this yourself by writing the necessary code:

```
x1 = pow ( e, pi );
x2 = pow ( pi, e );
if ( x1 < x2 )
{
  biggest = x2;
}
else
{
  biggest = x1;
}
```

To find the longest word in a file, you'd also have to write something like:

```
record_length = 0;

while ( more to read )
{
  new_length = length of next word;
  if ( record_length < new_length )
  {
    record_length = new_length;
  }
}
```

USER: Write Your Own Max Function

Now it might turn out that you had to find the maximum value in many places in your program, in which case it starts to become a problem having to write several lines to express a simple thought.

It would be so much nicer if you could define your own function, just like a <**cmath**> function, which might have the declaration:

```
double my_max ( double x1, double x2 );
```

meaning that **my_max** is a function that takes two real numbers as input, and returns a real number as a result.

USER: Write Your Own Max Function

We would like our programs to replace the several lines of code by a single line, reading something like:

```
y = my_max ( x1, x2 );
```

where, we assume, **x1** and **x2** are two real numbers. Then our user function **my_max()** would return the maximum of the two.

In fact, C++ allows you to write exactly such a user function, and to use it as described.

So far, we have only written programs involving a single function, called **main()**. Now we will see, with user functions, how a larg computer program is built out of many, cooperating function

USER: Form of a Function

The form of a user function is something like this:

output type `my_max ( double x1, double x2 )` <-- *interface*

```
{
  double x;   <--an "internal" variable
  ...
  middle part computes x from x1 and x2
  ...
  return x;   <-- how the result is returned
}
```

USER: Form of the MY_MAX Function

Here is a possible form of the **my_max** function:

```
double my_max ( double x1, double x2 )
{
  double x;

  if ( x2 < x1 )
  {
    x = x1;
  }
  else
  {
    x = x2;
  }
  return x;
}
```

USER: In and Out of the Function

The "middle" of the function makes sense. It's just what we said before. The unusual parts come at the beginning and end.

Note that the beginning and end are similar to a main function:

```
double my_max ( double x1, double x2 )  <-- the interface
{
  double x;  <--an "internal" variable
  ...
  middle part computes x from x1 and x2
  ...
  return x;  <-- how the result is returned
}
```

USER: In and Out of the Function

The first line tells us several things:

```
double my_max ( double x1, double x2 )
```

- this function is named **my_max**;
- this function takes two inputs, which we'll call **x1** and **x2**;
- both inputs are of **double** type;
- the output is also a **double** value;

The last line

```
return x;
```

identifies the point at which the result is ready to be returne

USER: Using the Function

compare.cpp:

```
beginning stuff
int main ( )
{
  double my_max ( double x1, double x2 );
  double x, y, z;

  cout << "Enter two numbers to compare: ";
  cin >> x >> y;

  z = my_max ( x, y );

  cout << "The maximum of " << x << " and " << y
       << " is " << z << "\n";
  return 0;
}         <-- The text of my_max follows the main program -->
double my_max ( double x1, double x2 )
{
  double x;
  if ( x1 < x2 )
  {
    x = x2;
  }
  else
  {
    x = x1;
  }
  return x;
}
```

USER: In and Out of the Function

Some interesting questions should come up in your mind, especially about the names of variables.

- When we use **my_max()**, must our input be called **x1** and **x2**?
- There is an **x** variable in **my_max()**.
  What if my program already has a variable called **x**?

USER: In and Out of the Function

The variables named **x1** and **x2** are sometimes called *dummy variables*. The name is supposed to capture the idea that these are simply stand-ins for whatever variables the user actually supplies. The user function has to give them names too, but these names are simply a convenience, and are local to the function.

Another way of saying this is that the scope of the variable names **x1** and **x2** is limited to the function **my_max**. The specific names are attached to specific values, but only as long as we are working in that function.

This answers the second question as well. Both the main program and the function **my_max()** can have variables called **x** but this really means we have two declarations:

1. the variable **x** as defined and used in **main()**;
2. the variable **x** as defined and used in **my_max()**.

USER: What Happens During Execution?

To try to understand how memory is used, and how the main program and function cooperate, let's walk through the execution of a program that calls **my_max()** which I will roughly sketch as:

initial stuff

```
int main ( )
{
  double my_max ( double x1, double x2 ); <-- Declare
  double x, y, z;                            function!
  x = 1;
  y = 2;
  z = my_max ( y, x );
  cout << "my_max ( " << y
       << "," << x << ") = " << z << "\n";
  return 0
}
```

USER: What Happens During Execution?

```
                        main scope   | my_max scope
                        -------------|----------------
                        X  Y  Z      |  X1  X2  X
                        -------------+----------------
x=1;                    -  -  -      |   -   -  -
y=2;                    -  -  -      |   -   -  -
z=my_max(y,x);          -  -  -      |   -   -  -
  if ( x2 < x1 )        -  -  -      |   -   -  -
    x = x1;             -  -  -      |   -   -  -
  else                  -  -  -      |   -   -  -
    x = x2;             -  -  -      |   -   -  -
  return x;             -  -  -      |   -   -  -
z=my_max(y,x);          -  -  -      |   -   -  -
```

USER: What Happens During Execution

```
                        main scope   | my_max scope
                        -------------|----------------
                        X  Y  Z      |  X1  X2  X
                        -------------+----------------
x=1;                    1  -  -      |
y=2;                    1  2  -      |
z=my_max(y,x);          1  2  ?      |   2   1  -   <-- created
  if ( x2 < x1 )                     |              <-- true
    x = x1;             1  2  ?      |   2   1  2
  else                               |
    x = x2;                          |
  return x;             1  2  -      |   2   1  2
z=my_max(y,x);          1  2  2      |   -   -  -   <-- terminated
```

Functions

- Introduction
- The <cmath> Functions
- User Functions
- **The CHOOSE Function**
- The Month Length
- Homework Program #4

CHOOSE: How Many Ways?

One way to estimate the probability of an event is to count how many ways it could happen out of how many total choices there are. This works when all the ways are equally probable.

If I am thinking about the game of poker, in which five cards are drawn from a deck of 52, I may want to know how many different hands are possible. (I assume that the order in which I draw a certain set of cards is not important, just which five I got.)

There is a formula that allows us to count such quantities, and it is called *n-choose-k*. This is sometimes symbolized by $C(n, k)$ or $\binom{n}{k}$.

The definition is:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

CHOOSE: How Many Ways?

Here, the symbol **n!** stands for the *factorial function*, whose simple definition for positive n is:

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

and we add the definition that $0! = 1$.

Now that we know what a factorial is, we can go back and look at the *n-choose-k* function to count poker hands:

$$\begin{aligned}\binom{52}{5} =& \frac{52!}{5!47!} = \frac{52 * 51 * 50 * \ldots * 3 * 2 * 1}{(5 * 4 * 3 * 2 * 1) * (47 * 46 * 45 * \ldots * 3 * 2 * 1)} \\ =& \frac{52 * 51 * 50 * 49 * 48}{5 * 4 * 3 * 2 * 1} \\ =& 2,598,960 \text{ possible poker hands.}\end{aligned}$$

which is, in some ways, a surprisingly small number!

USER: Using FACTORIAL to Define CHOOSE

So, if we needed to compute $\binom{n}{k}$, we have to compute three different factorials, of n, of k and of $n - k$. Rather than do that three times, it would be much easier to work out the computation once, make **factorial()** a user function, and then call it like this:

```
choose = factorial ( n ) / factorial ( k )
         / factorial ( n - k );
```

We've chosen a name for our user function, we know it has one integer input and one integer output, so we're ready to write it!

CHOOSE: The factorial() Function

```
int factorial ( int n )
{
  int i;
  int value;

  value = 1;
  for ( i = 1; i <= n; i++ )
  {
    value = value * i;
  }
  return value;
}
```

CHOOSE: The choose() Function

I mentioned that the value of *n-choose-k* comes up in probability calculations, so it might be nice to make that a function too! We know the formula, but now we need to "dress it up" with the interface information and declarations:

```
int choose ( int n, int k )
{
  int factorial ( int n );  <-- Declare factorial()
  int value;

  value = factorial ( n ) / factorial ( k )
          / factorial ( n - k );

  return value;
}
```

CHOOSE: Using choose()

Now if our main program wants to evaluate the number of poker hands, it has to declare **choose()**, and then call it. Notice that it does not have to declare **factorial()**, because the main program does not call that function directly.

beginning stuff

```
int main ( )
{
  int choose ( int n, int k );  <-- Declare choose()
  int n, k, value;

  n = 52;
  k = 5;
  value = choose ( n, k );
  cout << "Number of poker hands is " << value << "\n";
  return 0;
}
```

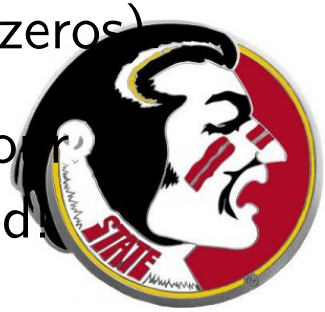
CHOOSE: Using choose()

This program is available as **poker_hands.cpp**. It looks correct. Mathematically, it is correct. But it is unable to get the right answer.

Is there anything peculiar about our computation that would explain the problem? Notice that when we did the problem by hand, we did not actually compute **52!**; instead, we used cancellation to simplify the problem.

If we had actually multiplied out **52!** first, (which would be acceptable), how big would that number be? Remember, there is a maximum possible integer in C++, and for the regular integers we are using, that number is about 2 billion: 2,000,000,000 (9 zeros).

So to fix this program, we'd have to think of how to teach our **choose()** program to use the same cancellation idea we used!



Functions

- Introduction
- The <cmath> Functions
- User Functions
- The CHOOSE Function
- **The Month Length**
- Homework Program #4

MONTH: Our Irregular Calendar

Thirty days hath September,
April, June and November,
Thirty one for all the rest –
Except February, which I detest;
It usually has twenty eight,
Or twenty nine at any rate.

Our irregular calendar makes it difficult to count. A month, more often than not, does not equal 30 days; a year is 12 months, but not 12*30 days, and so on. Every year begins on a different day of the week.

So how hard is it to answer the following question, which obviously has an exact, simple answer:

Exactly how many days old are you?

MONTH: How Shall I Count the Days?

If we decide that such a problem is worth answering using C++, we have to think about the procedure that would be necessary to do it.

Let's imagine that the user types in the month, day and year of birth as three numbers, such as 5, 4, 1776 for July 4th, 1776, for example. Let's call these values **m1**, **d1** and **y1**.

Let's suppose the computer can figure out the month, day, and year for today. Let's call these values **m2**, **d2** and **y2**.

How do we count the days between these two dates?

MONTH: A Formula Would be Great

One way to answer this question is to come up with a formula.

The formula can somehow work out from any year, month and day the total number of days since some day in the past, which we can call, maybe, **jd(y,m,d)**.

If we can do that, then:

```
my age in days = jd ( today ) - jd ( my birthday );
```

It is possible to do this, but it takes some thought. (Look up the "Julian Day Number" if you want to know more.)

MONTH: Imagine Tearing Pages Off

day_count.cpp:

A second way is to imagine tearing off pages from a calendar, one day at a time, until we reach today. Seems pretty simple.

So we have variables called **m**, **d** and **y**, which start at **m1**, **d1** and **y1**, and end at **m2**, **d2** and **y2**.

We start the day count at 0, and **m/d/y** at **m1/d1/y1**.

When we tear off a page of the calendar, the day count goes up by one. But also, **d** goes up by one:

```
while ( m != m2 || d != d2 || y != y2 )
{
  days = days + 1;
  d = d + 1;
  ...
```

MONTH: Fixing an Illegal Day

But the day number **d** doesn't go up forever. How do we know if it has gotten too high? If it is bigger than the number of days in the month. Ahh, so we have to keep track of the days in the month. And what do we do if **d** gets too big?

```
if ( month_length ( m, y ) < d )
{
  m = m + 1;
  d = 1;
}
```

Why does the **month_length** function need the value of **y**?

MONTH: Fixing an Illegal Month

But the month could have gotten too high. So we have to make sure it didn't go over 12. If it did, we have to correct it:

```
  if ( 12 < m )
  {
    y = y + 1
    m = 1;
  }
}
```

And that's the end of the **while** loop.

MONTH: The Month Length

The **month_length()** function might look like this:

```
int month_length ( int m, int y )
{
  bool leap_year ( int y );
  int value;

  if ( m == 4 || m == 6 || m == 9 || m == 11 )
  {
    value = 30;
  }
  else if ( m == 1 || m == 3 || m == 5 || m == 7 || m == 8 || m == 10 || m == 12 )
  {
    value = 31;
  }
  else
  {
    if ( leap_year ( y ) )
    {
      value = 29;
    }
    else
    {
      value = 28;
    }
  }
  return value;
}
```

(A **bool** *variable is a logical true/false variable.)*

MONTH: Remarks

In this overview, I've left some things out:

- What is the leap year function?
- Could the program tell what day of the week I was born (yes!);
- Could the program count forward to a future event?
- Could the program be more efficient (counting years at a time)?

However, I hope just thinking about how to solve this problem shows you that it can be natural to say "I'll handle that part of the calculation by writing another function."

This example, computing the number of days since you were born, would be an acceptable kind of final project. If you are interested in a calendar-based project, let me know.

Functions

- Introduction
- The <cmath> Functions
- User Functions
- The CHOOSE Function
- The Month Length
- **Homework Program #4**

ASSIGNMENT #4: The Body Mass Index

The Body Mass Index (BMI) is an attempt to summarize in a single number the amount to which a person is overweight or underweight. For many reasons, it is just a ballpark figure. We are only interested in this computation as an example.

The BMI is defined as

$$BMI = \frac{weight\ in\ kilograms}{(height\ in\ meters)^2}$$

Since we don't use the metric system, our weights are in pounds, and our heights are in feet. That means that before we can use this formula, we need to convert our weights and heights.

ASSIGNMENT #4: English to Metric Conversion

A formula to convert a weight **lb** in pounds to kilograms **kg** is:

$$kg = 0.4536 * lb;$$

A formula to convert a height **ft** in feet to meters **m** is:

$$m = 0.3048 * ft;$$

ASSIGNMENT #4: Write Three Functions!

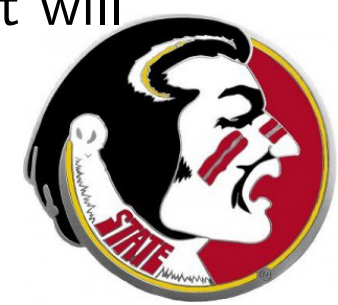
Write three functions:

- **lb_to_kg(*)** converts input pounds to output kilograms;
- **ft_to_m(*)** converts input feet to output meters;
- **bmi(*,*)** takes input weight in pounds and input height in feet, and returns the BMI;

Assume that all numbers are real numbers of type **double**.

Your **bmi(*,*)** function will call the other two functions for help, and so it will have to declare them.

The main function will call the **bmi(*,*)** function, and so it will need to declare the **bmi(*,*)** function.



ASSIGNMENT #4: Careful With Fractions!

Notice that the BMI is a fraction of the form $\frac{weight}{height^2}$.

Here are four ways to compute such a fraction, one of which is wrong.

```
value1 = weight / height * height;        <--wrong!
value2 = weight / height / height;
value3 = weight / ( height * height );
value4 = weight / pow ( height, 2.0 );
```

ASSIGNMENT #4: Details of Assignment

Write a main program which calls **bmi(*,*)** to determine the Body-Mass-Index for each of the following individuals:

Name	Weight in pounds	Height in Feet	BMI
Pinocchio	5	1.25	?
Big John	245	6.50	?
Miss Liberty	312,000	305.00	?

You may find this program harder to write than the previous ones. Please look at it early enough so you can ask for help!

Email to Detelina:

- your program's three BMI results;
- a copy of your program.

The program and output are due by Thursday, June 16.