Technical HW 2 Solutions

Problem 1:

Create a program in a file called "Program1.cpp" that inputs a number from the command line, interprets it as a long integer with the "atol" function (defined in <stdlib.h>), and calculates its factorial, if the integer is less than 20. If the integer is greater than or equal to 20, the program should print a warning and return zero. This should use a function called "factorial" that computes the factorial with signature:

unsigned int factorial(unsigned int x);

Your code should also handle the case where no arguments are provided and remind the user of the syntax..

This problem was mainly testing the ability to run on the command line and how to call a function. The reason you must require this to be less than 20 is that above that, you will run into an overflow error. Otherwise, the workhorse of the algorithm is, given the input as "x":

```
unsigned int ret = 1;
for ( unsigned int i = 1; i <= x; ++i ){
  ret *= i;
}
return ret;
```

Problem 2:

(15 points) Create a class template file called "LorentzVector.h" that implements a class template of a Lorentz vector using natural units c = 1 called "template<class F> LorentzVector". Create a method called "mass" that computes the invariant mass of a Lorentz vector ($m = \sqrt{t^2 - x^2 - y^2 - z^2}$). Make a specific class of type "LorentzVector<double>". Create two Lorentz vectors "v1 (1,0,0,1)" and "v2 (1,0,0,-1)". Write a program "Problem2.cpp" that will instantiate these two vectors and print them.
(10 points) Create an addition operator called "operator+" that will sum two four vectors and return the sum. Sum the vectors v1 and v2 from part a, using the same file "Problem2.cpp". Print the sum and its invariant mass.

This problem was mainly testing the syntax for the creation of classes and class templates. The workhorse of the algorithm to compute the mass is

```
F mass() const { return sqrt(t_*t_ - x_*x_ - y_*y_ - z_*z_); }
```

Problem 3:

Write a C++ program starting from "ReviewCpp/ClassExample/read_points_example.cc" renamed to "Program3.cpp". Create a class to represent a line (called "Line") that is declared in a file called "Line.h" and defined in a file called "Line.cc", with protected data members "double m_;" and "double b_;" that are the slope and y-intercept.
Be sure to copy the "Point" class (both "Point.cc" and "Point.h") into your Assignment2 directory from ReviewCpp/ClassExample.
The class should have two construction operators and two overloaded "set" methods that can EITHER input two Points (where you compute m and b), OR can input the slope and intercept directly.
The slope may be infinity, in which case set it to "std::numeric_limits<double>::infinity()".
The y-intercept may be undefined, in which case set it to "nan" (or "not a number") by setting it equal to "sqrt(-2)".
You are welcome to edit any code that you copy.

(15 points) Your main function should be defined in a file called "Problem3.cpp" that reads a file from the command line, which contains the following inputs (one per line). Compute lines from all pairs of inputs and print the equation of the line like "y = m x + b" where you substitute the values for m and b. The inputs are:
-1,-1
1,1
1,1
1,1.00000000000000005
1,2
3,4
1,8
(10 points) Sort the list of lines from part a in increasing order of slope, and print them all out with the same formatting as in a. Use the same function "Problem3.cpp".

This problem was mainly testing creation of classes, as well as careful handling of corner cases and numerical floating point comparisons. Remember that the "==" operator is extremely unsafe for floating point comparison, so you must use "std::abs(x1-x2)". In the solutions, the key part is in the "set" method. In case the line is vertical, the slope is infinity and the y-intercept is undefined, so we use "std::numeric_limits<double>::infinity()" and sqrt(-2) to set these values.

```
void Line::set( Point const & p1, Point const & p2)
{
  // First check if the line is vertical, so the slope is infinite and
  // the y-intercept is undefined, so we set it to "nan" (sqrt(-2))
  if ( std::abs(p1.x() - p2.x()) <
std::numeric_limits<double>::epsilon() ) {
    m_ = std::numeric_limits<double>::infinity();
    b_ = sqrt(-2);
  }
  // Otherwise we have normal execution
  else {
    m_ = (p2.y() - p1.y()) / (p2.x() - p1.x());
    b_ = p2.y() - m_ * p2.x();
  }
}
```