Factor Tree

This project involves implementing a binary tree that represents the factorization of a positive integer value.

A prime number is a positive integer \( p \) such that \( p > 1 \) and the only divisors of \( p \) are 1 and \( p \) itself. The Fundamental Theorem of Arithmetic states that, given any integer \( N > 1 \) there exist unique prime numbers \( p_1, \ldots, p_k \) and unique positive integers \( e_1, \ldots, e_k \) such that:

\[
N = p_1^{e_1} \times p_2^{e_2} \cdots \times p_k^{e_k}
\]

This is called the prime factorization of \( N \). Determining the prime factorization for a given \( N \) is a fundamental problem in number theory. This assignment is not so much about how to do this efficiently as about how to do it simply and represent the process using a simple binary tree structure.

Factorization can be implemented as a recursive transformation of a binary tree. For instance, the factorization of 7892 could be accomplished by the following transformation:

```
7892
  |  |
2  3946
```

In each tree, the root value equals the product of the values in the leaves. The transformation is accomplished by recursively applying an algorithm, to search for factors and add them as children, to each tree node. The process is somewhat interesting since the tree changes as the process progresses.

It is possible to compute the tree in other ways, depending primarily on how we go about discovering the factors, and how we decide to add them to the tree. In the example above, factors are discovered by considering consecutive integers starting with 2, and are added as children of the factored number in ascending order from left to right. If we discover factors by starting with the square root of the original number and counting down we might construct the following tree:

```
7892
  |  |
4  1973
```

In each case, leaf nodes are guaranteed to store only prime numbers, otherwise the recursive application of the factor-seeking algorithm would have added children. Another alternative would be for each leaf node to store both a prime number and the corresponding exponent it bears in the unique prime factorization. If we consider only prime factors, and determine their exponents as we find the factors, we might obtain the tree:

```
7892
  |  |
2^2  1973^1
```
In order to obtain consistent results for grading, we will specify precisely how you must build the factor tree. When building the factor tree for a positive integer $N$, you must consider the primes between 2 and $N$ in ascending order. When you find a prime divisor of $N$, you must then determine its exponent in the prime factorization of $N$, and add to the current tree node a left child representing that term in the prime factorization, and a right child containing the factor obtained by dividing the value in the current tree node by the prime power that went into the left child, provided that factor is greater than 1.

This leaves one important question: how can you efficiently determine the prime values between 2 and $N$?

One good answer is to use the *Sieve of Eratosthenes*. The method consists of writing down all the integers from 2 up to $N$ and then sieving out all the composite numbers in the list. Composite numbers are identified by a very simple observation: given an integer $m$, while $m$ itself may or may not be prime, none of the numbers of the form $km$, where $k$ is an integer greater than 1, are prime. So, we may first eliminate all the multiples of 2, then all multiples of 3. At this point, the next number that has not been eliminated would be 5, so we know 5 must be prime and we can eliminate all its multiples. Continuing in this manner, we eventually eliminate all the composite numbers in the range.

Question to ponder: at what point can we stop the process? That is, what's the largest value of $m$, in relation to $N$, that we must consider before we are guaranteed that we have found all the composite numbers?

**Implementation**

Your factor tree implementation must conform to the declaration shown below. You may safely add members, but do not rename or omit any of the ones that are shown. You will, of course, need a node type. The interface of the node is of no interest to the test harness that will drive your implementation.

```cpp
// FactorTree.h
// ... includes omitted

class FactorTree {
public:
    FactorTree();                                  // create empty tree
    FactorTree(unsigned int Value);                // create factor tree for Value
    FactorTree(const FactorTree& Source);          // deep copy support
    FactorTree& operator=(const FactorTree& RHS);  // restore tree to empty state
    ~FactorTree();                                 // deallocate tree structure
    void Clear();                                  // ... entirely up to you
    void Display(std::ostream& Out) const;         // display contents (see below)
private:
};
```

You must place the declaration of your `FactorTree` class in a header file named `FactorTree.h` because the test harness files will explicitly `#include` that file. No `friend` declarations are necessary.

**Design and implementation requirements**

There are some explicit requirements, in addition to those on the *Programming Standards* page of the course website:

- You must implement a C++ class for the `FactorTree` itself and also for the nodes it uses.
- The `FactorTree` class must conform to the given interface.
- You must properly allocate and deallocate memory, as needed.
- Under no circumstances should any of the class functions, other than `Display()` and its helper, write output.
The `Display()` function must write the contents of the `FactorTree` in the following format:

```
-----------:2^3
6978568
-----------:17^1
-----------:872321
---------------------:23^2
---------------------:51313
---------------------:97^1
```

The hyphens identify the level in the tree at which a value is stored. Note that you must produce exactly ten hyphens, followed by a colon, for each level down a value is in the tree. You also must not embed any blank lines within the tree display. Leaves should be displayed showing the prime and corresponding exponent, as shown. Internal nodes should be displayed showing only the factors they store.

If you don't follow these instructions, the automated evaluation will conclude that your tree is not structured correctly.

**Testing:**

I will be testing your implementation with my own test driver. I may (or may not) release information about that driver before the assignment is due. In any case, it is your responsibility to design and carry out a sensible test of your implementation before submitting it. For that purpose, you may share test code *(but absolutely no tree code!!)* via the class Forum.

Be warned that your copy logic will be tested thoroughly, as will the operation of the destructor. Be sure you test all of the interface elements thoroughly, both in isolation and in interleaved fashion.

Also be sure you read and understand the test build instructions that will be posted on the website. The default build you would get if you're developing under .NET will often mask even serious errors in an implementation.

**Evaluation:**

You should document your implementation in accordance with the *Programming Standards* page on the course website. It is possible that your implementation will be evaluated for documentation and design, as well as for correctness of results. If so, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

**Note well:** if you make two or more submissions that are tied for the highest score, the earliest of those will be graded. There will be absolutely no exceptions to this policy!

**Moral:** code and document to meet requirements from the beginning, rather than planning to retrofit documentation into a finished program.

Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

**What to turn in and how:**

This assignment will be auto-graded using a test harness on the Curator system. The testing will be done under Mandrake Linux 9.1 using g++ 3.3.1. Note that the Curator link and password you used for the previous assignment will change for this one. The correct submission link can be found at the URL below.

Submit a single gzip'd tar file containing the C++ header files for your `FactorTree` declarations and implementations, including the node type, to the Curator System. Submit only those declarations and implementations. Submit nothing else. Be sure that your header file only contains `include` directives for Standard C++ header files; any other `include`
directives will certainly result in compilation errors. Be sure you must use the specified names for your header files. The file you submit will be unpacked using the following command:

```
tar -zxf <name of your submitted file>
```

The unpacked files will then be compiled with the test harness using the following command:

```
g++ -o <name we give the executable> *.cpp
```

Instructions, and the appropriate link, for submitting to the Curator are given in the Student Guide at the Curator website:


You will be allowed to submit your solution up to ten times.

**Pledge:**

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Programming Standards page in one of your submitted files.