Assignment 1

Due. Sept. 22 (Thursday), 17:30.

1. Chapter 1, Programming 6, p. 41.
   Using the `DigitSum` function from the section entitled "The `while` statement" as a model, write a program that reads in an integer and then displays the number that has the same digits in the reverse order, as illustrated by this sample run:

   ```
   This program reverses the digits in an integer.
   Enter a positive integer: 123456789
   The reversed integer is 987654321
   ```

   To make sure your program can handle integers as large as the one shown in the example, use the type `long` instead of `int` in your program.

2. Chapter 1, Programming 7, p. 41.
   Greek mathematicians took a special interest in numbers that are equal to the sum of their proper divisors (a proper divisor of N is any divisor less than N itself.) They called such numbers *perfect numbers*. For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide evenly into 6. Similarly, 28 is a perfect number because it is the sum of 1, 2, 4, 7, and 14.

   Write a predicate function `IsPerfect` that takes an integer `n` and returns `true` if `n` is perfect, and `false` otherwise. Test your implementation by writing a main program that uses the `IsPerfect` function to check for perfect numbers in the range 1 to 9999 by testing each number in turn. When a perfect number is found, you program should display it on the screen. The first two lines of output should be 6 and 28. You program should find two other perfect numbers in the range.

3. When a floating-point number is converted to an integer in C++, the value is truncated by throwing away any fraction. Thus, when 4.99999 is converted to an integer, the result is 4. In many cases, it would be useful to have the option of *rounding* a floating-point value to the nearest integer. For a positive floating-point number `x`, the rounding operation can be achieved by adding 0.5 to `x` and then truncating the result to an integer. If the decimal fraction of `x` is less than .5, the truncated value will be the integer less than `x`; if the fraction is .5 or more, the truncated value will be the next larger integer. Because truncation always moves toward zero, negative numbers must be rounded by subtracting 0.5 and truncating, instead of adding 0.5.

   Write a function `Round(x)` that rounds a floating-point number `x` to the nearest integer. Show that your function works by writing a suitable main program to test it.