HW #4 (Cube & Time Class)

Part I:

- Consider the following definitions for class **Cube**:

```
class Cube {
private :
    double x, y, z;
public:
    …..
};
```

- Complete the class **Cube** by adding the following functions. Include for the functions in part **Ia)** and **Ib)** both the prototype and the functions code:

HW #4 (2)

Ia) Define a non-member function that overloads the operator **/** which divides the sum of the members x, y, and z in the first object by the sum of the corresponding members in the second object and return the result as double.

Ib) Define a member function that overload the operator **!=** which test if the volume of the first object is not equal the volume of the second object. The volume of a cube is x*y*z.

HW #4 (3)

Part II:

- Your task will be to create a class called **Time**, which will involve a variety of operator overloads.
- A Time object will store a quantity of time, in terms of **days**, **hours**, **minutes**, and **seconds**. You will overload some basic operators for use with these objects, including arithmetic, comparison, and insertion/extraction (I/O) operations.

- The program must be written in C++.

HW #4 (4)

- The **Time** class must allow for storage of a non-negative quantity of time, in term of days, hours, minutes, and seconds. The hours, minutes, and seconds portion will be expressed in the normal way that it would appear on a digital timer. Example: **18:30:58** means 18 hours, 30 minutes, 58 seconds. All values should be non-negative. The minutes and seconds values should never be more over 59, and the hours should never be over 23 (i.e., if you have 61 seconds, that is really 1 minute and 1 second). The Time should always be kept in this simplified format. There is no limit on the number of days. You should create appropriate data members in your class. All data members must be private.

HW #4 (5)

Constructors:

- The class should have a default constructor (**no** parameters), which should initialize the object so that it represents the quantity 0.
- The class should also have a constructor with a single integer parameter, which represents a quantity of seconds – which should be translated into the appropriate notation for a **Time** object. Note that this constructor with a single parameter will be a “conversion constructor” that allows automatic type conversions from “**int**” to “**Time**”. If the parameter is negative, default the **Time** object to represent 0.

HW #4 (6)

- The class should also have a constructor that takes 4 parameters, representing the days, hours, minutes, and seconds to use for initializing the object. If any of the provided values are negative, default the **Time** object to represent 0. If any of the provided values are too high (but all non-negative), simplify the object to the appropriate representation. Examples:

```
Time t; // this creates an object which is 0 days, 0 hours,
        // 0 minutes, 0 seconds
Time s(1234); // this creates an object representing 0 days,
              // 0 hours, 20 minutes, 34 seconds
```

HW #4 (7)

```
Time r(-123); // creates an object representing 0 days,
              // 0 hours, 0 minutes, 0 seconds
t = 4321; // conversion constructor allows this assignment.
          // t now stores 0 days, 1 hour, 12 minutes,
          // 1 seconds

Time x(1, 3, 5, 7); // 1 days, 3 hours, 5 minutes, 7 seconds
Time y(2, -4, 6, 8); // creates object representing 0, since
                     // -4 hours not legal
Time z(2, 25, 4, 62); // 3 days, 1 hours, 5 minutes, 2 seconds
```

HW #4 (8)

- Create an overload of the insertion operator << for output of **Time** objects. A time object should be printed in the format: **days~hh:mm:ss** – where days is the number of days, and the hours, minutes and seconds values are all expressed in two digits, as they would be on a digital stopwatch or timer. Examples:

10~12:23:48 means 10 days, 12 hours, 23 minutes, 48 seconds

123~04:02:33 means 123 days, 4 hours, 2 minutes, 33 seconds

HW #4 (9)

- Create an overload of the extraction operator >> for reading **Time** objects from an input stream. The format for the input of a **Time** object is the same as the output format listed above. This operator will need to do some error checking, as well. If any of the input values are negative, this is an illegal **Time** quantity, and the entire object should default to the value 0 (0 days, hours, minutes, seconds). If any of the values are over the allowable limit (i.e., not in simplified form), then this function should adjust the **Time** object so that it is in simplified form. Example: If the input object is "10~26:05:61", then it should be simplified to "11~02:06:01" since 61 seconds is really "1 minutes, 1 seconds", and 26 hours is really "1 days, 2 hours".

HW #4 **(10)**

- Create overloads for the + operator and the – operator to allow addition and subtraction of two quantities of time. Results should always be returned in the simplified form. For subtraction, if the first quantity of time is less than the second (which would normally result in a negative quantity), return the **Time** object 0~00:00:00 instead. Examples:

2~04:07:28 + 5~08:19:02 = 7~12:26:30

2~04:07:28 – 5~08:19:02 = 0~00:00:00

5~12:15:57 + 2~16:51:05 = 8~05:07:02

5~12:15:57 – 2~16:51:05 = 2~19:24:52

HW #4 (11)

- Create overloads for all 6 of the comparison operators (< , > , <= , >= , == , !=). Each of these operations should test two objects of type **Time** and return **true** or **false**. You are testing the **Time** objects for order and/or equality based on whether one quantity of time is more than (less than, equal to, etc) another.
- Create overloads for the increment and decrement operators (++ and --). You need to handled both the pre- and post- forms (pre-increment, post-increment, pre-decrement, post-increment). These operators should have their usual meaning – increment will add 1 second to the **Time** value,

HW #4 (12)

decrement will subtract 1 second. If the **Time** object is already at 0, then decrement does not change it (i.e., you still never will have a negative time value). Examples:

```
// suppose t1 is 2~04:07:58
// suppose t2 is 5~08:19:01
cout << t1++; // prints 2~04:07:58, t1 is now 2~04:07:59
cout << ++t1; // prints 2~04:08:00, t1 is now 2~04:08:00
cout << t2--; // prints 5~08:19:01, t2 is now 5~08:19:00
cout << --t2; // prints 5~08:18:59, t2 is now 5~08:18:59
```

- Sample main program to test the **Time** class:

```
#include <iostream>
#include "time.h"

using namespace std;

int main()
{
  Time t1, t2, t3(123456), t4(987654321);
  cout << "t1 = " << t1 << '\n';
  cout << "t2 = " << t2 << '\n';
  cout << "t3 = " << t3 << '\n';
  cout << "t4 = " << t4 << '\n';

  cout << "Enter first Time object (DAYS~HH:MM:SS): ";
  cin >> t1;

  cout << "Enter second Time object (DAYS~HH:MM:SS): ";
  cin >> t2;
  cout << "\n\n";
  cout << t1 << " + " << t2 << " = " << t1 + t2 << '\n';
  cout << t1 << " - " << t2 << " = " << t1 - t2 << "\n\n";

  if (t1 < t2) cout << t1 << " < " << t2 << " is TRUE\n";
  if (t1 > t2) cout << t1 << " > " << t2 << " is TRUE\n";
  if (t1 <= t2) cout << t1 << " <= " << t2 << " is TRUE\n";
  if (t1 >= t2) cout << t1 << " >= " << t2 << " is TRUE\n";
  if (t1 == t2) cout << t1 << " == " << t2 << " is TRUE\n";
  if (t1 != t2) cout << t1 << " != " << t2 << " is TRUE\n\n";

  cout << t1 << " + 654321 = " << t1 + 654321 << '\n';
  cout << t2 << " + 15263748 = " << t2 + 15263748 << '\n';
}
```