Lab Sheet 1

"Bad programmers worry about the code.
Good programmers worry about data
structures and their relationships."
– Linus Torvalds

1. A *wrapper class* is a class that "wraps" around something else, just like its name. Wrapper classes are classes that are used to make *primitive variables* into objects, and to make wrapped objects into primitives. *int*, *boolean*, *double* are all primitive data types and their respective wrapper classes are *Integer*, *Boolean*, and *Double*.

   Wrapper classes are useful in storing primitive data types in higher level data structures such as Stack<Object>, List<Object>, Queue<Object>, since (in some programming languages) primitives cannot be directly placed in these data structures they must be boxed in the wrapper classes.

   Implement a wrapper class `Integer` to represent an integer quantity. The private data member can be a simple `int` type with the name `data`. It should have the following *constructors*:

   i) A *default* constructor that initializes `data` to 0. e.g., `Integer i1;`
   ii) A *copy* constructor that allows the copying of Integer objects. e.g., `Integer i2(i1);`
   iii) A constructor that takes a single *int* value. e.g., `Integer i1(10);`
   iv) A constructor that takes a *string* representation of an integer. e.g., `Integer i2(“123”).`

   The constructor that takes the string argument should throw an `invalid_argument` *exception* (of the `stdexcept` class) if the string does not represent an integer. For example, `Integer i1(“12a”).` You can have the following function signature for the constructor that takes the string argument:

   ```
   Integer(const char* s) throw (invalid_argument);
   ```

   A *mutator* method is a method used to control changes to a variable. The *mutator* method, sometimes called a "*setter*", is most often used in object-oriented programming, in keeping with the principle of *encapsulation*. According to this principle, member variables of a class are made private to hide and protect them from other code, and can only be modified by a public member function (the mutator method), which takes the desired new value as a parameter, optionally validates it, and modifies the private member variable. Often a "*setter*" is accompanied by a "*getter*" (also known as an *accessor*), which returns the value of the private member variable.

   The *Integer* class should have *getter* and *setter* methods with the following function signatures:

   i) `int get_data();`
   ii) `void set_data(int);`

   Overload the intput (>>) and output (<<) operators for the *Integer* class. Make use of friend functions to accomplish this. For example:

   ```
   Integer i1(10), i2(20);
   cout << i1 << endl;
   cout << i2 << endl;
   ```

   You can have the following prototypes for the above functions:

   ```
   friend ostream& operator<< (ostream &out, const Integer& i);
   friend istream& operator>> (istream &in, Integer& i);
   ```

   Overload the `+`, `-`, `*` and `/` operators to achieve the *addition*, *subtraction*, *multiplication*, and *division* of Integer objects. For example:

   ```
   Integer i1(10), i2(20), i3;
   i3 = i1 + i2;
   cout << i3 << endl;
   ```

The *addition* and *subtraction* operators should be *member functions* of the class and the *multiplication* and *division* operator should be *friend functions* of the class (should be defined outside the class). Division operator should accomplish integer division only. Division operator

Implement a function with the prototype `‘string Integer::to_string()’` which converts the Integer object to its *string* representation. For example:

```
Integer i1(123);
/* ‘i1.to_string() will return the STL string object “123”. */
cout << i1.to_string() << endl;
```

Implement a function `‘static int Integer::parse_int(string) throw (invalid_argumet)’` that converts the string representation of a number to its *integer* version. Make it a *static member function* of the class, so that it can be invoked using the class name rather than object name.

Overload the *assignment operator* ‘=’ to assign one Integer object to another (there is a language requirement that the assignment operator can only be defined as a member function of the class).

Overload the *relational operators* <, >, <=, >=, and == to compare two Integer objects (the return type of relational operators should be *bool*).

Overload the *increment* and *decrement* operators (++ and `--`), both the *prefix* and *postfix* versions, for the Integer class. For consistency with the built-in operators, the *prefix version* should return a *reference* to the incremented/decremented object and the *postfix version* should return the old value of the Integer object as a value and not as a reference. (There is no language requirement that the increment and decrement operators should be member functions of the class. However, because these operators *change the state of the object*, it is better to make them *member functions* of the class, and not friend functions).

The user of the `Integer` class should be able to do the following:

```
Integer i1(-4), i2(10);
cout << "i1: " << i1 << endl;
++i1;
cout << "After incrementing i1: " << i1 << endl;
cout << i1-- << endl;
cout << "After decrementing: " << i1 << endl;
(--i1)_.set_data(20);          /* What happens here?          */
cout << i1 << endl;            /* What is the output here?    */
(i1++).set_data(30);           /* Is it error?                */
Cout << i1 << endl;            /* Will it print ‘30’?         */
```

You may have the following function prototypes for the *increment* and *decrement* operator functions:

```
Integer& operator++();              /* Prefix increment.    */
Integer operator++(int);            /* Postfix increment.   */
Integer& operator--();              /* Prefix decrement.    */
Integer operator--(int);            /* Postfix decrement.   */
```

Overload the *logical operators* &&, `||` and `!` for the Integer class. Make them member functions of the class. The user of the class should be able to use Integer objects as follows:

```
Integer i1(-4), i2(10);
cout << "i1: " << i1 << endl;
cout << "OR  : It is " << ((i1 || i2) ? "true" : "false") << endl;
cout << "AND : It is " << ((i1 && i2) ? "true" : "false") << endl;
cout << "NOT : It is " << ((! i1)     ? "true" : "false") << endl;
```

You may have the following function prototypes for the logical operator functions:

```
bool operator||(const Integer&);
bool operator&&(const Integer&);
bool operator! ();
```

Make a *driver* function (the *main* function) to use the functionalities of the `Integer` class that you implemented.

Split up the source code between *header* files and *cpp* files. The `Integer` class definition should be in a header file with the name `Integer.h`. The definition of the member functions of the `Integer` class (and other helper functions and friend functions) should be in the file `Integer.cpp`. The main function can be in the file `main.cpp`.

2. Implement a class `time24` to represent the time of a day in 24 hour format. The `time24` *ADT* is given below:

```
/* A class to represent time in 24 hour format. */
class time24{
private:
    int hour;                   /* To represent hour.         */
    int minute;                 /* To represent minute.       */
    string separator;           /* ':', '/', '-', etc., to    */

    /* Utility function that sets the hour value in the ragen 0 to 23 and
     * the minute value in the range 0 to 59.
     */
    void normalize_time( );

public:
    /* Constructors. */
    time24(               );
    time24(int h, int m   );
    time24(const time24& t);

    /* Get the time and minutes of the day. */
    int get_hour( )const;
    int get_minute( ) const

    /* Input from the keyboard time in the form hh:mm.
     * Make sure that the time is normalized after it is accepted from the user.
     */
    void read_time( );

    /* Write the time to the output stream in the format hh:mm. */
    void write_time( );

    /* Function to get the length of the current time to some later time 't' as a
     * time24 value.
     *
     * Precondition: time24 object passed to the function should not be earlier
     * than the current time. Throw 'range_error' if the precondition is not
     * satisfied.
     */
    time24 duration(const time24& t);

    /* Function to update the time by adding minutes to the current time. */
    void add_time(int minutes);
};
```

Create a program that uses the `time24` objects to compute the *cost of parking a car* in a public garage. Assume that the rate is Rs. 10 per hour. The program prompts the user to input two times when a customer enters and later exits the garage. The output is a *receipt* that includes the *arrival time*, the *departure time* and the *length of time* the car is parked, along with the *total charges*.

Sample run 1:

```
Enter the entry and exit times:
11:40
14:15

----------------------------
Receipt
----------------------------
Car enters at: 11:40
```

```
Car exits at : 14:15
Parking time : 2 hrs 35 mins
Rate         : Rs. 10/hour
----------------------------
Total Cost   : Rs. 25.83
----------------------------
```

Sample run 2:

```
Enter the entry and exit times:
11:20
10:40

Error: Exit time cannot be less than entry time.
```

Sample run 3:

```
Enter the entry and exit times:
9:20
24:40

Error: Exit time is wrong.
```

Sample run 4:

```
Enter the entry and exit times:
0:30
9:20

----------------------------
Receipt
----------------------------
Car enters at: 0:30
Car exits at : 9:20
Parking time : 8 hrs 50 mins
Rate         : Rs. 10/hour
----------------------------
Total Cost   : Rs. 88.33
----------------------------
```