Objective:

- To explore/implement an application of Composition relationship.

Challenge-X: *Sample Run*

If TA asks then for every task given in this lab, you have to write a sample run through which you should be able to create object(s) of classes developed and execute their different methods. TA may instruct you to execute any behavior which should act/behave as required. Failing to do so will result in marks deduction in that particular challenge.

Challenge-A: *Bounded Integral Type*

(13)

In your programming career, you may not always get ready to be used data types which works as per your program logic/requirements. Sometimes, we have to design/mold the language define types so that we may get a clean/easy support for the problem/solution under consideration.
For example, if we were writing a program to implement a clock or a calendar, then we would need numbers to represent minutes or days. But these numbers aren't C/C++ integers, which range from (-2^{31} to $2^{31}-1$); they range from 0 to 59 and 1 to 31 (or less!), respectively. C/C++ primitive types are useful building blocks, but at the level of atoms, not molecules.

For this task, you need to develop an ADT named as 'BoundedInteger' which store values within a given range.
The most important feature of our '*BoundedInteger*' class is to "wrap around" when we try to increment or decrement some given value in it.
For example, suppose that we have a *minutes* bounded integer (0~59) whose value is 52. Adding 10 to this object gives it a value of 2. See the sample run for further understanding.
The detail below gives explanation of how the '*BoundedInteger*' ADT members will act to achieve the desired objective.

Data Members:

The first two members i.e. *lowerBound* and *upperBound* represents the range of data that can be stored in bounded integer object.
The *value* represents the integral-value stored in bounded integer which should be in the range *(lowerBound <= value <= upperBound)*

- `lowerBound;`
- `upperBound;`
- `value;`

Member Functions:

- `bool isValidBound(int lb, int ub);`
  A utility function which helps you find whether the bound is valid or not. i.e. it should be: (lb <= ub).
- `BoundedInteger();` (.5)
  Constructor, which initializes the lower Bound with INT_MIN and upper bound with INT_MAX, and set the value to lower bound. INT_MIN and INT_MAX are predefined constants defined in C++, where INT_MIN is the minimum value in int and INT_MAX is the maximum value in int.
- `BoundedInteger(int lb, int ub);` (1)
  It initializes the lower bound and upper bound as received in parameters. It should validate the bounds and also initializes the value with lower bound. If the bound is not valid then it initializes as per default constructor.
- `BoundedInteger(int lb, int ub, int val);` (1)
  Same as the previous constructor except it also receive the val to set with data member value.
- `void setValue(int val);` (.5)
  setter for value.
- `int getValue();`
  getter for value.
- `int getLowerBound();`
  getter for lowerBound
- `int getUpperBound();`

21 % 7 ③ × 21 % 12 ⑨ bounded + 3 11 + 3 14 % 12 + 1 ③ 12 − 5 ⑦ 21 % 7 + 1 11 + 10 ④ 21 % 7 ③ + 5 ⓪ 21 − 7 $\boxed{7-4=3}$ 3 11 + 0 21 % 12 − 9 ③ + 5 11 % 7 $\boxed{4+5}$

getter for upperBound

- `void increment(int inc=1);` (4)
  increments/adds the received inc in the value keeping in view the bounds and wraps around accordingly.
- `void decrement(int dec=1);` (6)
  decrements/subtracts the received dec from the value keeping in view the bounds and wraps around accordingly.
- `bool isValidValue(int val);`
  validates, whether the received val is valid according to bounds or not.

```
//BoundedInteger.h
class BoundedInteger
{
     int lowerBound;
     int upperBound;
     int value;
     bool isValidBound(int lb, int ub) const;
public:
     BoundedInteger();
     BoundedInteger(int lb, int ub);
     BoundedInteger(int lb, int ub, int val);
     void setValue(int val);
     int getValue() const;
     int getLowerBound() const;
     int getUpperBound() const;
     void increment(int inc=1);
     void decrement(int dec=1);
     bool isValidValue(int val) const;
};
```

```
//BoundedInteger.cpp
/*
Definitions for BoundedInteger.h
*/

//Driver.cpp
int main()
{
     BoundedInteger bi(5,12,7);
     bi.decrement(4);
     cout<<bi.getValue()<<'\n';//display 11
     bi.increment(10);
     cout<<bi.getValue()<<'\n';//display 5
     return 0;
}
```

Note: While developing logic for increment/decrement function, you may go for applying looping mechanism but that will *not* give you full credit rather very less marks. So, think smart.

Challenge-B: *Time* (10)

Reimplement your Time class as given below. The change is the data representation of class Time is done through BoundedInteger class instead of primitive ints. Hopefully, it will help our data to be more resistant to wrong values because even the Date class own member functions will not be able to place wrong values in them.

Note: Virtues of Abstraction: As long as we don't change the public interface (public members (data + functions prototypes)) of our classes, client will not be affected at all.

	10
`class Time`	
`{`	
`BoundedInteger hour;`	
`BoundedInteger minute;`	
`BoundedInteger second;`	
`public:`	
`Time ( int = 0, int = 0, int = 0);`	1
`void setMinute ( int m );`	
`void setSecond ( int s );`	0.5
`void setHour(int h);`	
`void setTime ( int h, int m, int s );`	
`int getHour ( ) const;`	
`int getMinute ( ) const;`	
`int getSecond ( ) const;`	
`void printTwentyFourHourFormat ( ) const;`	1
`void printTwelveHourFormat ( ) const;`	
`void incSec ( int inc = 1 );`	1
`void incMin ( int inc = 1 );`	0.5
`void incHour ( int inc = 1 );`	
`void deccSec ( int inc = 1 );`	3
`void decMin ( int inc = 1 );`	2
`void decHour ( int inc = 1 );`	1
`};`	

20 10 less simple
20 30 large simple
40 70 less simple
40 90 less complex

Challenge-C: *Date*

(12.5)

Reimplement your Date class as given below. The change is the data representation of class Date is done through BoundedInteger class instead of primitive ints. Hopefully, it will help our data to be more resistant to wrong values because even the Date class own member functions will not be able to place wrong values in them.

Note: Virtues of Abstraction: As long as we don't change the public interface (public members (data + functions prototypes)) of our classes, client will not be affected at all.

	12.5
`class Date`	
`{`	
`    BoundedInteger day;`	
`    BoundedInteger month;`	
`    BoundedInteger year;`	
`    static const int daysInMonth[ 13 ];`	
`    bool isLeapYear () const;`	✓ 0.5
`public:`	
`    Date ( int d, int m, int y );`	
`    void setDate ( int, int, int );`	✗ 1
`    void setDay ( int );`	
`    void setMonth ( int );`	0.5
`    void setYear ( int );`	0.25
`    int getDay ( ) const;`	
`    int getMonth ( ) const;`	
`    int getYear ( ) const;`	
`    void incDay ( int = 1 );`	1.5 1
`    void incMonth ( int = 1 );`	.5 ✓
`    void incYear ( int = 1 );`	
`    void decDay ( int = 1 );`	3
`    void decMonth ( int = 1 );`	2 5
`    void decYear ( int = 1 );`	1
`    void printFormat1 ( ) const;`	
`    void printFormat2 ( ) const;`	
`    void printFormat3 ( ) const;`	
`    CString getDateInFormat1 ( ) const;`	1 ✓
`    CString getDateInFormat2 ( ) const;`	
`    CString getDateInFormat3 ( ) const;`	
`    int getTotalDaysInMonth ( ) const;`	0.5 ✗
`    long long int getDaysBetweenDates ( const Date & end ) const;`	1 ✓
`};`	

Challenge-D: *Watch*

(15)

Implement class 'Watch' which keeps record of both date and time information in it. See the class below which is self-explanatory.

	15
`class Watch`	
`{`	
`    Date watchDate;`	
`    Time watchTime;`	
`public:`	
`    Watch ( );`	
`    Watch ( Date d, Time t );`	✓ 2
`    Watch (const Watch & ); // Although not necessary at all`	✓ 1
`                           // but would like to see, how you implement it.`	✓ 2
`    void setTime ( Time t );`	1 0.5
`    void setDate ( Date d );`	
`    void setMinute ( int m );`	✓
`    void setSecond ( int s );`	1
`    void setHour ( int h );`	
`    void setTime ( int h, int m, int s );`	
`    Date getDate ( ) const;`	1 ✓

`Time getTime ( ) const;`	
`int getHour ( ) const;`	✓ 1
`int getMinute ( ) const;`	
`int getSecond ( ) const;`	
`void printTwentyFourHourFormat ( ) const;`	✓ 1
`void printTwelveHourFormat ( ) const;`	
`void setDay ( int );`	✓ 1
`void setMonth (int );`	
`void setYear ( int );`	
`int getDay ( ) const;`	
`int getMonth ( ) const;`	
`int getYear ( ) const;`	
`void incSec ( int inc = 1 );`	✗ 1
`void incMin ( int inc = 1 );`	
`void incHour ( int inc = 1 );`	
`void deccSec ( int dec = 1 );`	✓ 1
`void decMin ( int dec = 1 );`	
`void decHour ( int dec = 1 );`	
`void incDay ( int = 1 );`	✓ 1
`void incMonth ( int = 1 );`	
`void incYear ( int = 1 );`	
`void decDay ( int = 1 );`	✓ 1
`void decMonth ( int = 1 );`	
`void decYear ( int = 1 );`	
`};`	

Challenge-Y: *CString* (0)

Responsible for storing CString.		0
`class CString {};`	Your complete CString class as per practice 13	20

"It isn't the mountains ahead to climb that wear you out; it's the pebble in your shoe".

-- Muhammad Ali --

The champ is referring to the everyday distractions that take us off course and keep us from seeing the bigger picture of our lives. While the mountain, our life's purpose, is always ahead of us, the pebbles often trip us up. We may perpetuate unhealthy behaviors, patterns and thoughts in our lives by worrying about all the what ifs, could have beens and never-going-to-be's.

Keep your eyes fixed on that mountain! Eliminate the pebbles, one by one.