CSCI 2170 Abstract Data Type, class

- In many cases, the solution to a problem requires operation on data, for example add, sort, print, and remove operations on data. In these cases, it is better to put the focus on data, and think in terms of what we can do with a collection of data, INDEPENDENTLY of HOW we do it. This is called **data abstraction**. Data abstraction is the fundamental idea of Object Oriented Programming.

- **Abstract Data Type (ADT)** – a collection of data together with a set of operations on that data
  ADT is different from Data Structure – data structure is a construct within a programming language that stores a collection of data. It defines how data is stored in the memory for a program.

- When constructing an ADT, there are two steps involved:
  - Specification – indicate precisely what each operation of an ADT does.
  - Implementation –specify how each operation is implemented. This includes selecting a data structure for data involved in ADT using programming language → data structure is part of an ADT's implementation

As a client, using an ADT is like using a vending machine:
we do not grab the snacks directly, we do not care how the snacks are stored..,
we press a button, and expect the corresponding snack to come out.
→
we request for an operation of an ADT, the operation outputs the appropriate result.
We do not need to know how the result is generated, or how data is stored in the ADT.

- **ADT implemented as class:**

class, object
- class is the ADT implemented in C++
- object is an instance of a class

public, private, const
- Data and methods declared in the public section can be accessed by the client program of the class
- Data and methods declared in the private section can not be accessed by the client program of the class
- A const method may not modify the data of a class

constructor (default constructor, other constructors)
- activated when an object of the class is created
- used to initialize data of the object
- a class can have more than one constructor

destructor
- activated when an object of the class exits its scope
- a class can only have one destructor
- destructor is ONLY necessary in a class when data of the class has acquired dynamically allocated memory. In this case, destructor is

method -- member function of a class

accessor – get method, retrieve the data value of an object

mutator – set methods, change the data value of an object

- Use **#ifndef / #define / #endif** to prevent multiple inclusion of a class definition

The Specification file: time.h

```
#ifndef TimeClass_H
#define TimeClass_H

class TimeClass
{
 public:

   TimeClass();  // default constructor
   TimeClass(int h, int m, int s);  // parameterized constructor

   // set the time to hour(h), minute(m) and second(s)
   // Pre-condition: the new hour, minute and second values are provided as the parameters of the function
   // Post-condition: the objects time is changed to the values provided as the parameters of the function
   void SetTime(int h, int m, int s);

   // return the hour value of the current time
   // Pre-condition: none
   // Post-condition: the hour value is returned
   int GetHour() const;

   // Change the hour value of the current time
   // Pre-condition:
   // Post-condition: hour is set to h
   void SetHour(int h);

   // the current time is displayed in the format hh:mm:ss
   // Pre-condition: none
   // Post-condition: the current time is displayed in the specified format
   void Display() const;

   // Increment the current time by one second
   // Pre-condition: none
   // Post-condition: the time is increased by one second
   void Increment();

   // Compare whether two time are the same
   // Pre-condition: a second time is provided as parameter
   // Post-condition: returns true if the time provided is the same as the current time, returns false otherwise
   bool IsEqualTo(TimeClass t) const;

   // Compare whether the current time is less than the time provided in parameter
   // Pre-condition: a second time is provided as parameter
   // Post-condition: returns true if the current time is less than the time provided, returns false otherwise
   bool LessThan(TimeClass t) const;

 private:
   int hour;
   int minute;
   int second;
};

#endif
```

The Implementation file: time.cpp

```
#include "time.h"
#include <iostream>
using namespace std;

TimeClass::TimeClass()
{
        hour = 0;
        minute = 0;
        second = 0;
}

TimeClass::TimeClass(int h, int m, int s)
{
        hour = h;
        minute = m;
        second = s;
}

void TimeClass::SetTime(int h, int m, int s)
{
        hour = h;
        minute = m;
        second = s;
}

void TimeClass::Display() const
{
        if (hour < 10)
                cout << "0";
        cout << hour;

        cout << ":";

        if (minute < 10)
                cout << "0";
        cout << minute;

        cout << ":";

        if (second < 10)
                cout << "0";
        cout << second;

        cout << endl;
}

void TimeClass::Increment()
{
        if (second < 59){
                second++;
        }
```

```
	else{
		second = 0;
		if (minute < 59) {
			minute ++;
		}
		else {
			minute = 0;
			if (hour < 24){
				hour ++;
			}
			else {
				hour = 1;
			}
		}
	}
}

int TimeClass::GetHour() const
{
	return hour;
}

void TimeClass::SetHour(int h)
{
	hour = h;
}

bool TimeClass::IsEqualTo(TimeClass t) const
{
	if ((hour == t.hour) && (minute == t.minute) && (second == t.second))
		return true;
	else
		return false;
}

bool TimeClass::LessThan(TimeClass t) const
{
	if (hour < t.hour)
		return true;
	else if (hour == t.hour) {
		if (minute < t.minute)
			return true;
		else if (minute == t.minute)  {
			if (second < t.second)
				return true;
			}
	}

	return false;
}
```

The Client program: main.cpp

```
#include <iostream>
#include "time.h"
using namespace std;

int main()
{
	TimeClass time1;     // create the object using the default constructor
	TimeClass time2(3, 40, 5);   // create the object using the value constructor

	time2.Display();
	if (time1.IsEqualTo(time2))     // time1 object invokes the IsEqualTo method
		cout << "Equal time " << endl;
	else
		cout << "Not Equal" << endl;

	time1 = time2;
	time1.SetHour(time1.GetHour()+1);
	time1.Display();

	for (int i=0; i<15; i++)
	{
		time2.Increment();
		time2.Display();
	}

	return 0;
}
```