# Assignment 3 : Polymorphism + Operator Overloading

| | |
|---|---|
| Deadline: | Friday March 18 at 23:59 |
| Type: | Individual Assignment 7% |
| Weight: | |

**Marking Scheme:**
- Program correctness (75%)
- Program output format, clarity, completeness, and accuracy (10%)
- Program indentation and readability (5%)
- Choice of significant names for identifiers (5%)
- Comments - description of variables and constants (5%)

## Problem statement

A Graph is formally define as *G=(N,E)* consisting of the set *N* of vertices (or nodes) and the set *E* of edges, which are ordered pairs of the starting vertex and the ending vertex. Each vertex has ID and value as its basic attributes. Each edge has weight, starting vertex and ending vertex.

A Graph can be a directed graph where vertices are connected by edges, and all the edges are directed from one vertex to another.

A Directed Acyclic Graph (DAG) is a finite directed graph with directed cycles. This means from any vertex v, there is no way to follow a sequence of edge that eventually loops back to v again.

An undirected graph is a graph where the edges are bidirectional.

The definition of the abstract class **Graph** is provided below. **Note that all its member functions are pure virtual.** This is because the implementation of these functions is different from one graph to another. You are allowed to slightly modify this class by adding new member functions.

```
class Graph{
public:
        Graph();
        virtual ~Graph();
        //add in one vertex; bool returns if it is added successfully.
        virtual bool addVertex(Vertex& v)=0;
        virtual bool addVertices(Vertex* vArray) = 0;
        //the edges that has connection with this vertex need to be removed;
        virtual bool removeVertex (Vertex &v)= 0;
        //remove a edge;  as a result, some node may remain as orphan.
        virtual bool addEdge(Edge& e) = 0;

        // remove the edge
        virtual bool remove(Edge& e)=0;
        // return bool if a vertex exists in a graph;
        virtual bool searchVertex(const Vertex& v) = 0;
        // return bool if a Edge exists in a graph;
        virtual bool searchEdge(const Edge& e) =0;

        // display the whole graph with your own defined format
        virtual void display() const = 0;
        // convert the whole graph to a string such as 1-2-4-5; 1-3-5; each path is separated by ';'
        or just print the edges of the graph as a list
         // define your own format of a string representation of the graph.
        virtual string toString () const = 0;
        //remove all the vertices and edges;
        virtual bool clean() = 0;
// you may consider the following in your project:
// add/remove a set of edge; as a result, some node may remain as orphan.
        //virtual bool addEdges(Edge* eArray) = 0;
        //virtual bool removeEdges(Edge* eArray) = 0;
        // display the path that contains the vertex;
        virtual void display(Vertex& v) const = 0;
        // display the path that contains the edge;
        virtual void display(Edge& e) const = 0;
};
```

**Problem 1: Abstract Class and Polymorphism (60 Marks)**

1. Create a class Vertex and Edge to represent the vertices and edges of a graph. Provide programming code of your classes.

2. Create a concrete derived class of Graph. It can be directed, undirected or DAG. Provide full code of the derived class.

**3. Create a driver class with the main function to test the creation of a graph, and invoking each member function.**

**Problem 2: Operator Overloading (40 Marks)**

Improve the class created in Problem 1 by overloading the operators **=, ==, ++, <<, >, +.** These operators are defined as follows

1. **G1 == G2** returns true if G1 and G2 have the exact same vertices and edges
2. G1 **=** G2, assign Graph G2 to Graph G1;
3. **G++ and ++G, increases the weights of all edges by one;**
4. G3 = G1 **+** G2, returns a graph that contains all the nodes of G1 and G2, all the edges of G1 and G2;
5. G1 **>** G2, returns boolean if the sum of weights of G1' edges are larger than the sum of weights of G2's edges.
6. **<<**G outputs the graph G.
7. **Create a driver class with the main function to test the creation of graphs, and invoking each operator.**