CSc 300 – Assignment #5 – Gamradt
Due: 11-29-23 (Late: 12-06-23 @ 8:00 AM)

Create a user-defined Abstract Data Type (ADT) named **Graph**
Use an appropriate set of C++ header/implementation files as discussed in class
- **Graph** is implemented using **fixed sized arrays**
- **Graph** is implemented using an **adjacency matrix** consisting of 0 or more **Element** values
    - o **Element** is an exportable **unsigned short** data type

The **Graph** ADT must define and implement the following data types and operations.
- Do not add to or modify the public interface (exportable components – public components).
- Do not add to or modify any attributes or data types (storage components).

__**Exportable Operations: (declared .h file and defined .cpp file)**__

Graph	parameterized constructor – creates the described graph described graph file name passed in default graph file name **data.dat** **// standard text file** uses setGraph
~Graph	destructor – does nothing
dijkstra	performs Dijkstra's algorithm on the graph

__**Non-Exportable Operations: (declared .h file and defined .cpp file)**__

Graph	copy constructor – cannot be used to create the graph
setGraph	initializes the graph using the graph data contained within the graph file
setVisited	initializes the set of nodes to all unvisited – all false
setStart	prompts the user for the starting node validate the entered starting node – prompt until a valid one is entered
view	displays the contents of the distance array – see required output format below
restart	prompt the user to see if they wish to run dijkstra again using the same graph allow dijkstra to be run multiple times on the same graph before quitting

__**User-Defined Data Types:**__
Element

Graph Data File: **// standard text file – spaces NO tabs**

```
4                                            | V | – nodeCount – | V | x | V |
0         5         10        65535          [0][0]  [0][1]  [0][2]  [0][3]
65535     0         65535     3              [1][0]  [1][1]  [1][2]  [1][3]
65535     7         0         65535          [2][0]  [2][1]  [2][2]  [2][3]
65535     65535     4         0              [3][0]  [3][1]  [3][2]  [3][3]
```

USHRT_MAX == 65535 // maximum unsigned short – used to represent infinity
// predefined constants located in <climits>

Output Requirements: (view)

```
Distance[0] = 65535          // no path
Distance[1] = 0              // start node
Distance[2] = 7
Distance[3] = 3
```

Required header file (.h). **// only partially specified**

```
//  General description of the ADT and supported operations – exportable operations only
//  Do not include any implementation details

#pragma once

class Graph {
        public:                                                  //  exportable
//  General description of each of the ADT operations/functions – exportable operations only
                explicit Graph(const string = "data.dat");
                ~Graph();
                void dijkstra();
        private:                                                 //  non-exportable
//  No private member documentation – implementation details are hidden/abstracted away
                typedef unsigned short Element;
                enum {GRAPH_LIMIT = 15};
                Graph(const Graph &) = delete;
                void setGraph(const string);
                void setVisited();
                unsigned short setStart() const;
                void view() const;
                bool restart() const;
                Element cost[GRAPH_LIMIT][GRAPH_LIMIT];
                Element distance[GRAPH_LIMIT];
                bool visited[GRAPH_LIMIT];
                unsigned short nodeCount;
};
```

Graph ADT include sequence: // Never include .cpp files

$$\begin{tikzcd}\text{main.cpp} \arrow[r] & \text{Graph.h} & \text{Graph.cpp} \arrow[l]\end{tikzcd}$$

Graph ADT incremental building sequence: // Using make

1. Place all files in the project folder // I would use Gamradt5
2. make // Process Makefile
3. ./output // Run project – make generated executable

Make sure that you completely document the header/implementation files.

- The header (.h) file tells the user exactly how to use your ADT
    - General descriptions only – do not include implementation details
- The implementation file (.cpp) tells the implementer/programmer exactly how the ADT works
    - Detailed descriptions – include implementation details

I will write a test program that will include your ADT so all header/implementation files tested must use common names. You **MUST** use:

- the **EXACT** same names for each data type and function in the header/implementation files.
- the **EXACT** same function argument sequence in the header/implementation files.

Use **PITA** everywhere possible

- Prefer Initialization to Assignment

Apply function **Reuse** wherever possible

- E.g., constructors, destructor, …

Project Folder:	Lastname5	// I would use Gamradt5
Graph.h	**Graph** class header file	
Graph.cpp	**Graph** class implementation file	
main.cpp	driver program file	// I will use my own
Makefile	appropriate set of incremental build rules	// "1" module

Push your assignment solution to your GitHub account, then send me a shared link to the assignment repository

- E.g., CSc300 // CSc300
    - Remember that a 20% reduction is applied for not using GitHub
    - See **Assignment Requirements** – D2L Handouts Folder

List the class number, your lastname, and assignment number as the e-mail message subject:
SUBJECT: csc300 – Lastname – a5 // I would use "… Gamradt …"

Pseudocode

```
setGraph( graph file name )
        open graph file
        test open status                                // same idea as testing new status
        read in the graph size                          // N x N => | V | x | V |
        for each vertex in the graph                    // v
                for each adjacent vertex in the graph   // w
                        read one value from graph file
                        store one value in adjacency matrix     // [v][w]
        close graph file
```