Concordia University
Gina Cody School of Engineering and Computer Science
Winter 2021

COEN244 - Programming Methodology II
Final Project - Report

Asha Islam (40051511)
Pavithra Sivagnanasuntharam (40117356)

To be handed on : Friday, April 23rd, 2021
Teacher: Prof. Yan Liu

"We certify that this submission is our original work and meets the Gina Cody School's Expectation of Originality."

ABSTRACT

This project consists of creating a program where a real-life application is demonstrated. This project in particular utilizes the concept of hospital employees to represent their relationships in a hospital. This is done with an undirected graph, considering there is no employee being directed toward another, they instead work together, demonstrating a bidirectional relation. Among several techniques, inheritance, polymorphism, operator overloading, as well as exception handling was used. These techniques accompanied by the several functions in the **undirectedGraph** class have aided in displaying the information of the hospital employees including the relations among them. To conclude, the program has run through the functions and output the desired results successfully.

KEYWORDS

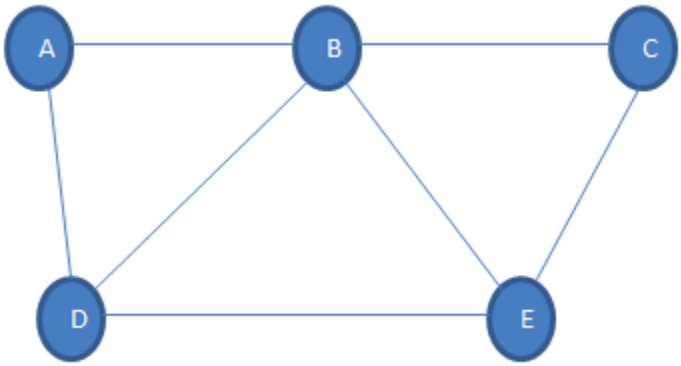
Keyword	**Class**
virtual	Graph.cpp Graph.h undirectedGraph.cpp undirectedGraph.h
dynamic_cast<...>	Main (driver)
try/catch	Main (driver)
HospitalEmployees	HospitalEmployees.cpp HospitalEmployees.h Main (driver)
vertex	vertex.h Vertex.cpp undirectedGraph.cpp
Edge	Edge.h Edge.cpp
print();	Vertex.cpp Edge.cpp

INTRODUCTION

A graph is a non-linear representation of multiple data in one diagram which allows to illustrate relationships between these data. Many forms of graphs exist, whether they are pie charts, bar graphs, and much more, they all serve the common purpose of demonstrating the relationship between data. In C++, when creating a graph, there is an association made between vertices, which are node points in a graph called vertices that contain the data. These vertices are connected with the help of edges, which links one vertex to another.

There are multiple types of graphs[1], such as a directed graph or an undirected graph. A directed graph is used when there needs to be a direct link made from one vertex to another, that is the edge begins at one specific vertex and ends at another specific vertex, making it necessary to show that for instance vertex A is linked to vertex B and not the other way around.

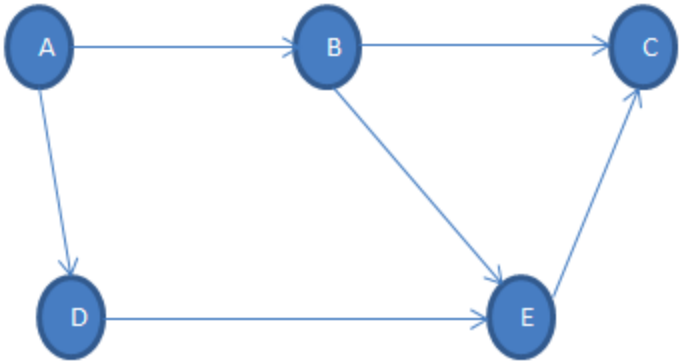
Figure 1. Undirected Graph.



A directed graph can also form a cycle, that is when one vertex is connected to another vertex, but not adjacent vertices are linked to one another. This creates a cycle that goes either clockwise or counterclockwise.

Figure 2. Directed Graph

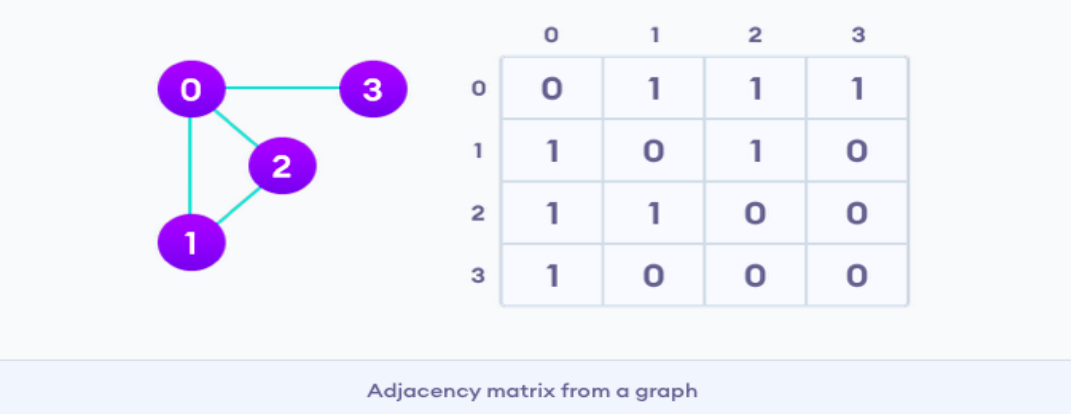
[1] https://www.softwaretestinghelp.com/graph-implementation-cpp/



In the picture above, we see that there are arrows that are pointing from one vertex to another, meaning that there is a sequential pattern. On the other hand, the undirected graph does not use arrows, meaning that there is only a relationshional pattern. In order to make a link between one vertex to another, we need to add an edge between the two nodes. To do so, we need to get an identification value from the initial node and terminal node.

We can represent the information about the relation between the vertices through an adjacency matrix[2]. An adjacency matrix, by its definition, is a matrix which has a dimension **m x m**. This m value derives from the fact that there are **m** vertices in the graph. The goal of an adjacency matrix is to show the edges that connect two vertices. When two vertices have a relation or a connection, there is a 1 that is displayed on the adjacency matrix. Otherwise, the value of the ith and jth position remains the same, which is of 0.

Figure 3. Adjacency Matrix with Relation to a Graph.



[2] https://www.programiz.com/dsa/graph-adjacency-matrix

Another way of displaying information about a graph is by using functions that are going to be displaying specific details on which vertex is connected to which.

The focus of our project is related to hospital employees. In fact, we are going to be working on a fixed amount of hospital employees and create different relations between them. In a real life setting, we know that healthcare professionals work with each other. For example, in the operating room, we see nurses, surgeons, physician assistants that are working on a common goal, which is to save lives. Our goal, in this project is to show the relations each hospital employee has with one another. We are going to be using an undirected graphing method, considering we are looking for relations, rather than sequential.

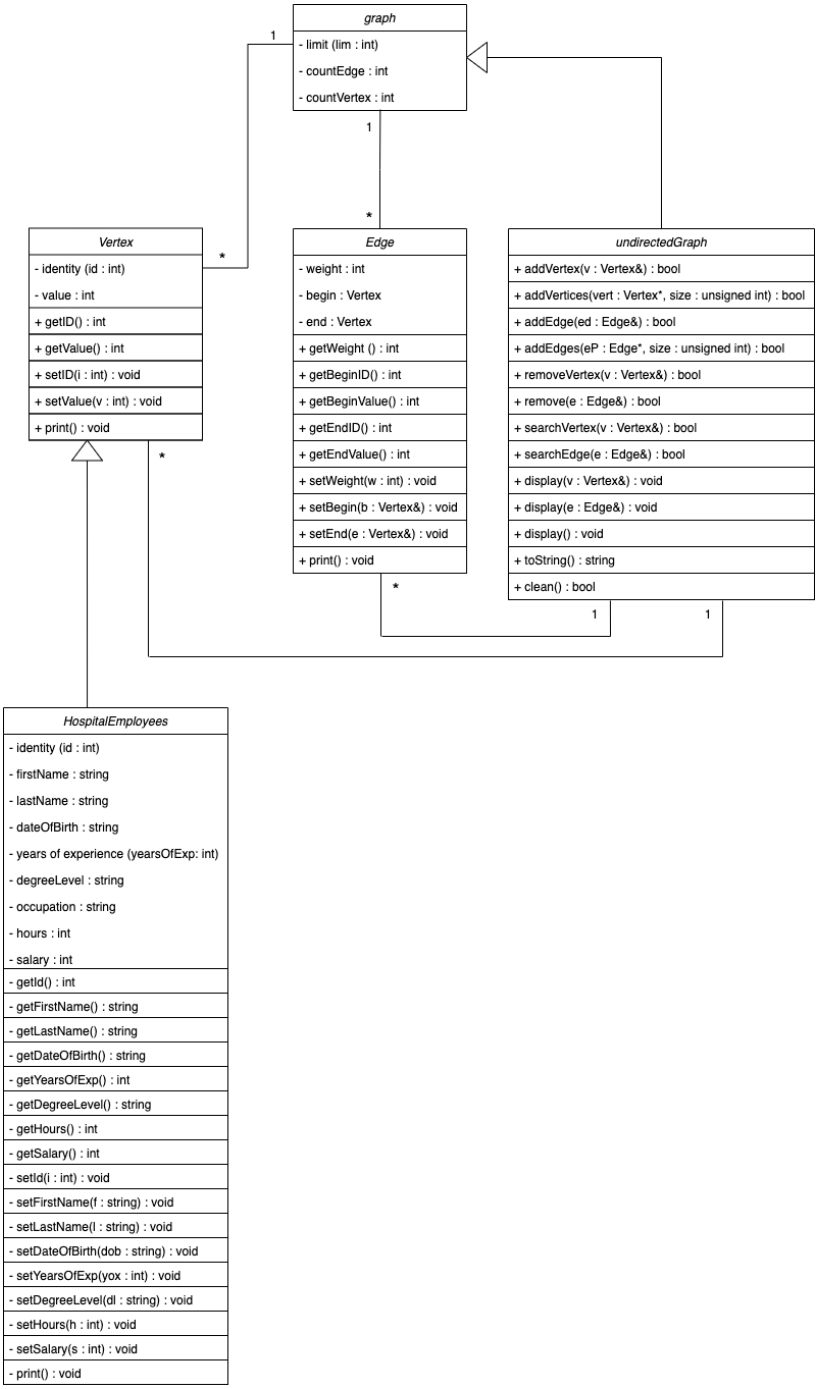
METHODOLOGY

We have used the Eclipse and XCode software in order to compile and run our project, since the code that we wrote was written in C++. For the UML design and modelling schema, we have used a tool on Google called diagram.net to create the graph. The same goes for the graph in which we show the relation between the vertices. For graphing the relation between the employees, we have decided to use the **Graph** class as an abstract class, since it is going to be containing virtual functions. In order to graph the relations between the employees, we must use undirected relations. In the main method, we are going to be referring to the **undirectedGraph** class when creating graphs.

DESIGN DESCRIPTION

a) Provide a Class Diagram that describes, in detail, the classes you employ, and the relationships between them (following UML conventions)

Figure 4. UML Design.



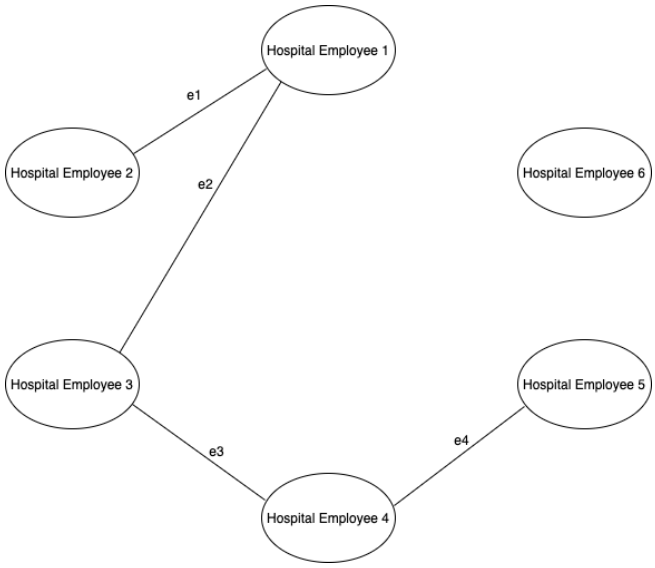
The class diagram above illustrates the various associations made among the classes created in the program. To begin, there is a class **graph** which inherits the class **undirectedGraph**. The class **graph** is therefore the base class and **undirectedGraph** is the derived class. This connection is represented with the **generalization** symbol.

There is another inheritance connection between the class **Vertex** and the class **HospitalEmployees**, where **Vertex** is the base class and **undirectedGraph** is the derived class. This connection is represented with the **generalization** symbol.

There is an association between the class **graph** and **Vertex**. This connection is shown as a **many-to-one**. In this case, there are many vertices in one graph, however there is only one graph in which a particular vertex is found.

Another association is between the class **graph** and the class **Edge**. This link is shown as a **many-to-one** association. Here, a graph can contain many edges, however a specific edge is only found in one graph.

Figure 5. HospitalEmployees and Edge Relations.



The diagram above represents various associations that can be made between vertices. These associations are therefore various ways to link one hospital employee to another, however it is an undirected graph. This signifies that the link is not directed from

one employee to another, there is instead a relationship between them which allows to connect them together. An example would be the relation between a doctor and a nurse who work in the same department. When an employee does not have any relation to another hospital employee, that employee is represented as an orphan vertex, which is the one with no edges connected to it, in this case hospital employee 6.

b) Describe at least 2 non-trivial methods (member function) in your program

addVertex:

One of the member functions is the **addVertex** function. This is a **bool** function which returns true or false depending on the proper addition of a vertex into the graph. This is done by first using the **if** statement in order to confirm that the number of vertices in the graph does not surpass the limit, that is the maximum number of vertices in the graph that was initialized before with the variable **lim**. If the condition is met, that the number of vertices represented by the variable **countVertex**, which is updated each time a vertex is added or removed from the graph, is indeed less than the limit, the program proceeds to the next line. If this condition is not met, the program returns false and ends reading the function's code. In the case where the condition was met, the following line is also an **if** statement, which in its turn verifies if the vertex that is being added is the first element in the array. This is done by comparing the **countVertex** to the integer 0. If the condition is met, that **countVertex** is equal to 0, that means there are currently no elements in the array and the new vertex is the first one. In this case the program continues to the next line, which assigns the array element **vertex[countVertex]** to the pointer **v**. By doing so, the pointer represents the last vertex in the array of vertices. The next line is to assign the number of vertices in the graph to the new number of vertices, which is **countVertex + 1** since there was an addition of one vertex. If these lines of codes were executed properly, the program returns **true** to show that the vertex was added correctly into the graph. If on the other hand, the **countVertex** was not equal to 0, the program skips to the line of code **else**, which means that this is not the first vertex in the graph. Here, the program proceeds to do a **for** loop, where an integer **i** increments until the number of vertices **countVertex**. In this loop,

for each position **i**, the program verifies if the vertex that is attempted to be added already exists in the graph. The following condition is read in order to verify this:

- Verify if the identity (ID) of an existing vertex in the array is the same as the identity (ID) of the vertex being added. For searching through the vertices already in the array in the graph, the code **vertex[i].getID()** is written, where **vertex[i]** is the vertex at the respective position **i** following the loop and **getID()** is the getter function for the identity in the class **Vertex**.

If this condition is met, the program returns **false**, meaning that the vertex already exists with this same identity and that the vertex failed to be added into the graph.

searchVertex:

A second member function is the **searchVertex** function. This is a **bool** function which consists in searching through the array of vertices in the graph for the vertex that was entered by a user. To be more specific, the program goes through the array of vertices with the help of iteration and verifies if the different attributes of the vertex are identical to the attributes given by the user. The total number of vertices in the graph has been initialized and each time a vertex is added or removed from the graph, the variable **countVertex** will update itself accordingly. This variable is used in a **for** loop where an integer variable **i** is incremented until the number of vertices found in **countVertex**. For each value **i**, the program uses an **if** statement to verify if certain conditions are met in order to proceed with the code. The conditions are as follows:

- Verify if the identity (ID) of the vertex at the respective position **i** in the array is equal, or is the same, as the identity (ID) of the vertex that is searched by the user. For the vertex at position **i** in the array, the array element is called with **vertex[i].getID()**, where getID() is the getter function of the identity in the class **Vertex**. As for the vertex identity entered by the user, it is called with the pointer **v** in the form **v.getID()**.

- Verify if the value of the vertex at the respective position **i** in the array is the same as the value of the vertex that is searched by the user. The vertex value at position **i** in the existing array is represented by **vertex[i].getValue()**, where getValue() is the getter function created in the class **Vertex**. The vertex value searched by the user that is compared is represented by **v.getValue()**.

Both of these conditions need to be satisfied in order to proceed to the next step in the code. In order to show this, the symbol **&&** is used. If the two conditions above were met, the program continues to the next line of code, which is **return true**. The program then outputs **true** if all conditions in the **if** statement were met, meaning that the specific vertex is found in the graph by having the same identity and value as asked by the user. However, if one or more of the conditions was not met, the loop will terminate, and the program goes to the line of code **return false**. This means that specific vertex was not found in the array of vertices contained in the graph because either the identity, the value, or both were not the same as the one entered by the user.

c) Describe your usages of 3 of techniques from inheritance, polymorphism, operator overloading, template and exception handling

Inheritance

There is usage of inheritance among classes in the following manners:

The class **undirectedGraph** is derived from the class **graph**, therefore the class **graph** is the base class and **undirectedGraph** is the derived class. This would signify that the attributes found in the class **graph** are passed down to the class **undirectedGraph**. To be more specific, these two classes are linked in a way that the class **undirectedGraph** can access the attributes in class **graph**, unlike other classes that are not linked through inheritance.

Figure 6. Inheritance from Graph..

```
#include "graph.h"
using namespace std;

class undirectedGraph: public Graph {

    public:
            undirectedGraph();
            virtual ~undirectedGraph();

            //adding the vertex and array of vertices
            virtual bool addVertex(Vertex&);
            virtual bool addVertices(Vertex*, unsigned int);

            //adding edge and array of edges
            virtual bool addEdge(Edge&);
```

The class **HospitalEmployees** is derived from the class **Vertex**, where **Vertex** is therefore the base class. This means that the class **HospitalEmployees** can access the attributes found in class **Vertex**. There is inheritance here because the vertices in the graph represent the data of each hospital employee entered into the graph. This means that the attributes found in class **Vertex** apply to the hospital employees, therefore in this case, the identity and the value given to a certain vertex is now part of the attributes of a hospital employee. The hospital employee would then be recognized through the identity of the vertex to which it has been assigned and when creating the graph, the identity of the vertex will be used to identify the hospital employee in question.

Figure 7. Inheritance from Vertex.

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

class HospitalEmployees:public Vertex{
private:
    int idNum;
    string firstName;
    string lastName;
    string dateOfBirth;
    int yearsOfExp;
```

Polymorphism

Another technique used while writing the code for our project is polymorphism. Polymorphism, by its definition, is a technique in which an object could take many forms[3]. In our

[3] https://www.cplusplus.com/doc/tutorial/polymorphism/

case, we have used polymorphism in various forms through our program. In fact, if we take a look in **figure x**, we have used a cast in order to convert from one type to another.

Figure 8. Polymorphism.

```
//Employee#1 - Doctor
Vertex v1(1, 111);

//get information for the Hospital Employee
int iden = 12345;
string fname = "John";
string lname = "Doe";
string dob = "01-01-2000";
int yearsofexp =2;
string degree = "BEng";
string occ = "Bioengineer";
int hours =37;
int salary = 45;

HospitalEmployees h1(v1.getID(), v1.getValue(), iden,fname, lname, dob, yearsofexp, degree, occ, hours, salary);

//dynamic casting for nursing to hospitalemployees
v1 = dynamic_cast<Vertex&>(h1);
```

For this type of casting to occur, we must have some sort of inheritance. In our case, we had the class **Vertex** and **HospitalEmployees**. In the code, the Vertex class is the base class, and the HospitalEmployees class is the derived class. In order to convert the derived class into a base class, we must use casting. We have chosen to use dynamic casting, since it could convert from one pointer or reference type of another pointer or reference type.

Base b1;

Derived d1;

b1 = dynamic_cast <Base&>(d1);

We have used dynamic casting to make the HospitalEmployees into a Vertex, since we are going to be using vertices and edges in order to create the relations between the hospital employees.

We have also used virtual functions, because we have functions with the same name in different classes that are redefined, but undergo the same function. For example, in the class **Graph** and **undirectedGraph**, we have a list of functions with the same name. However, in the base class **Graph**, the functions are pure virtual functions that do not need to be defined, because there is a redefinition in the derived class **undirectedGraph**.

Figure 9. Inheritance in connection with Polymorphism.

```
class undirectedGraph: public Graph {

    public:
            undirectedGraph();
            virtual ~undirectedGraph();

            //adding the vertex and array of vertices
            virtual bool addVertex(Vertex&);
            virtual bool addVertices(Vertex*, unsigned int);

            //adding edge and array of edges
            virtual bool addEdge(Edge& );
            virtual bool addEdges(Edge* , unsigned int);

            //removing edge and vertex
            virtual bool remove(Edge&);
            virtual bool removeVertex(Vertex& );

            //searching vertex and edge
            virtual bool searchVertex(const Vertex& );
            virtual bool searchEdge(const Edge& );

            //displaying various paths containing a vertex or edge
            //dont want to confuse with the print function, cuz already overloaded
            virtual void display(Vertex&) const;
            virtual void display(Edge& ) const;
            virtual void display() const;

            //conversion to string to display data
            virtual string toString () const;

            //remove all vertex and edge
            virtual bool clean();

            //compare and have same edge/vertex
            bool operator==(const undirectedGraph&) const;

};
```

Figure 10. Inheritance and Polymorphism.

```
class Graph{

    protected:
            Vertex *vertex;
            Edge* edge;
            int lim;
            int countEdge;
            int countVertex;

    public:
            Graph();
            virtual ~Graph();
            virtual bool addVertex(Vertex& )=0;
            virtual bool addVertices(Vertex* , unsigned int) = 0;
            virtual bool removeVertex(Vertex& ) = 0;
            virtual bool addEdge(Edge& ) = 0;
            virtual bool addEdges(Edge* , unsigned int) = 0;
            virtual bool remove(Edge& ) = 0;
            virtual bool searchVertex(const Vertex& ) = 0;
            virtual bool searchEdge(const Edge& ) = 0;
            virtual void display(Vertex& ) const = 0;
            virtual void display(Edge& ) const = 0;
            virtual void display() const = 0;
            virtual string toString () const = 0;
            virtual bool clean() = 0;

};
```

Operator overloading

We have used operator overloading for one instance in the main file. In fact, in order to compare one graph to another, we could use the overloaded equality operator to see if the edges and the vertices of one graph is equal to another. If they have the same elements and the same information, then it is going to be returning true.

Figure 11. Operator Overloading.

```
 //same vertex and edge in two compared graphs
bool undirectedGraph::operator==(const undirectedGraph & graph)const{

    if (countEdge == graph.countEdge && countVertex == graph.countVertex){
        bool *ed;
        bool *ve;
        ed = new bool[countEdge];
        ve = new bool[countVertex];

        //initializing the array pointers
        for(int i = 0; i< countEdge; i++){
            ed[i] = false;
        }

        for(int j = 0; j< countVertex; j++){
                ve[j] = false;
        }

        //compare and have the same id and value in the two graphs
        //for the edge
        for(int m = 0; m<countEdge; m++){
            for(int n = 0; n<countEdge; n++){
                if((edge[m].getBeginID()==graph.edge[n].getBeginID() && edge[m].getEndID()==graph.edge[n].getEndID()
                        && edge[m].getBeginValue()== graph.edge[n].getBeginValue()&& edge[m].getEndValue()==graph.edge[n].getEndValue()
                        && edge[m].getWeight()==graph.edge[n].getWeight()) || (edge[m].getEndID()==graph.edge[n].getBeginID()
                        && edge[m].getBeginID()==graph.edge[n].getEndID() && edge[m].getEndValue()== graph.edge[n].getBeginValue()
                        && edge[m].getBeginValue()==graph.edge[n].getEndValue()&& edge[m].getWeight()==graph.edge[n].getWeight())){
                    ed[m] = true;
                }
            }
        }
        //compare and have the same id and value in the two graphs
        //for the vertices
        for(int o = 0; o<countVertex; o++){
            for(int p = 0; p<countVertex; p++){
                if(vertex[o].getID() == graph.vertex[p].getID() && vertex[o].getValue()==graph.vertex[p].getValue()){
                    ve[o] = true;
                }
            }
        }



        //if there are any false value in the array pointer, then return false
        for(int q = 0; q<countEdge; q++){
            if(ed[q] == false){
                return false;
            }
        }

        //return false if there are any false value in the array pointer of vertices
        for(int r = 0; r<countVertex; r++){
            if(ve[r] == false){
                return false;
            }
        }

        //if they dont have the same number of vertex or edge
    } else if (countEdge != graph.countEdge || countVertex != graph.countVertex){
        return false;
    }

    return true;
}
```

Exception handling

Exception handling should be included in programming codes, because of the fact that programs are prone to errors. This method is used in order to catch errors and fix them. We have used exception handling with try-catch blocks.

In our code, specifically in the main program, we have added a try-catch block in the part where we ask the user if they want to add another employee. The answers should be ‘y’ or ‘n’. If it is ‘n’, then we are going to be getting out of the while loop and continue on with the program. If it is a yes, then we are going to be asking the user to enter the information about the new employee and store it in the object **newEmp** of type **HospitalEmployees**. If anything else than those two options are entered, we go into the catch block in which an error message is going to be printed. It is going to prompt the user to re-enter a valid answer, hence why there is a loop that is involved in this problematic code.

Figure 12. Try-Catch Block.

```
try{
    cin >> ans;
    if (ans=='n'){
        throw 1.00;
    } else if (ans !='y'){
        throw 1;
    } else {
        looping = false;

        int idenNew;
        cout << "Identity Number: ";
        cin >> idenNew;
        vNew.setValue(idenNew);
        cout <<"\n";
```

//---------code continues in the file----------

```
    } catch(double a){
        cout <<"You have chosen not to add any further employee.\n\n";
        break;

    } catch(int catching){
        cout << "Your answer is not valid, please try again.\n";

    }
}
```

When the exception is caught, it is going to be going to the catch block that is appropriate. In fact, if we look at our example above, we see that the first throw is of type double. Hence, this throw is going to be caught in the **double a** block. The same goes for the second exception, which is of type int. This is going to be caught in the block that is with the value **int catching**.

EQUATIONS

The skeletons of some of the techniques used are going to be illustrated in this section. Note that the format is going to be depending on the type of variables that are used.

Try/catch blocks

```
try {
        // code to be tried
        throw exception;
} catch (type exception) {
        // code to be executed in case of exception
}
```

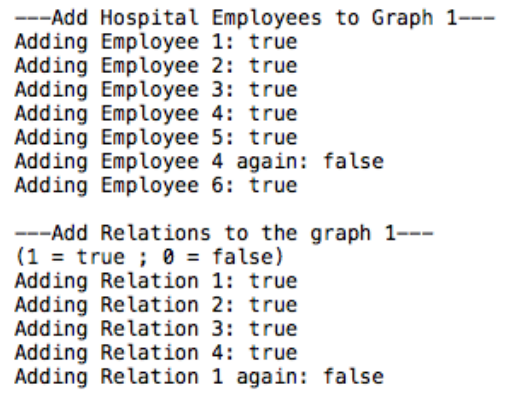
Dynamic casting

```
Base b1;
Derived d1;
b1 = dynamic_cast <Base&>(d1);
```

RESULTS

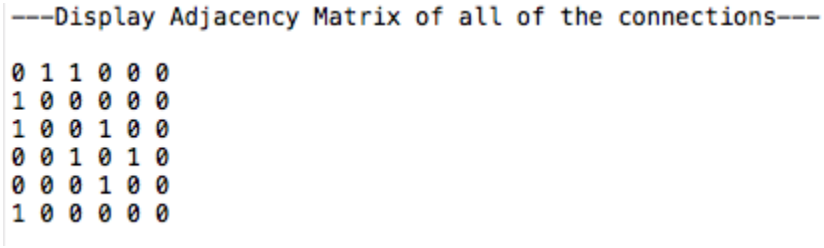
We have created the graphs, in which we have added six employees. With those six employees, we have created relations through the use of edges.

Figure 13. Results of Adding Edges and Vertices.



However, it is important to note that not all of the edges created have been added in the graph. For example, the edge **e5** has not been added to the graph despite being created. Therefore, in the adjacency matrix, we see that there are no relations in the coordinate [6-1][5-1] and [5-1][6-1].

Figure 14. Results of Adjacency Matrix.



Since the edge has not been added, the vertex v6, which is represented by the hospital employee **h6**, is considered orphan. Let's take a look at the detailed relation of the graph:

Figure 15. Displaying Details About the Graph.

```
---Displays details of relations in Graph 1---

Employee Relation Graph:

Employee ID:12345--Position#: 1 --> Employee ID:23456--Position#: 2
Employee ID:23456--Position#: 4 --> Employee ID:23456--Position#: 5


Orphan Employees:

Employee ID:23456--Position#: 6
```

Now, let's look at the effect of removing employees from the graph:

Figure 16. Removing Employees.

```
---Remove employee and relations---
Removes employee 1: true
Removes relation 1: false
Removes relation 2: true
```

As seen, once you remove an employee from the graph, you don't need to remove the relation, as the person has been removed. The employee that was related to this employee, if they don't have any other relation, will become an orphan employee. This means that they do not work with another employee.

Figure 17. Displaying Information After Node Removal.

```
Employee Relation Graph:

Employee ID:23456--Position#: 4 --> Employee ID:23456--Position#: 5


Orphan Employees:

Employee ID:23456--Position#: 2
Employee ID:23456--Position#: 6
```

We see the removal of the relations in the adjacency matrix as well:

Figure 18. Adjacency Matrix After Removal of Employee.

```
---Adjacency matrix---

0 0 0 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
1 0 0 0 1 0
0 0 0 1 0 1
0 0 0 0 1 0
```

Now, let's assume that we are in another department. In that other department, we have the same employees. The difference we see in department 2 (graph 2) , compared to the first department (graph 1) is that the hospital employees work with different people, because the tasks are different. We have added different connections, and have compared the two graphs to see if there is any difference in the networking. Here is the result:

Figure 19. Comparison of Two Graphs

```
---Compare Graph 1 & 2 Employee Network---
Network 1 == Network 2?: false
```

Since there is a difference in the relations, the result of the comparison is false.

We are then going to be asking the user if they want to add another employee into the network in graph 1.

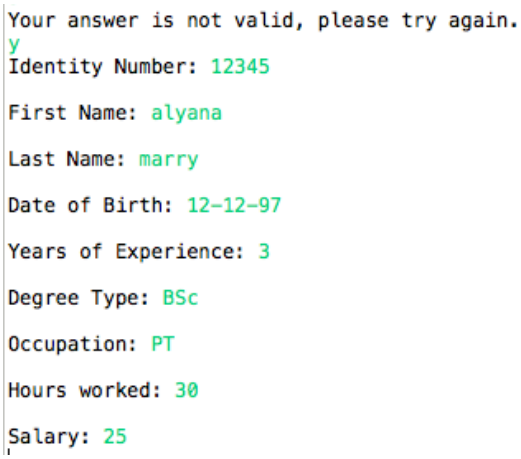
If the answer is in an invalid format, the following is going to be displayed:

Figure 20. Adding More Employee to System.

```
Do you want to add another Employee into the system? (y/n): r
Your answer is not valid, please try again.
2
Your answer is not valid, please try again.
5
```

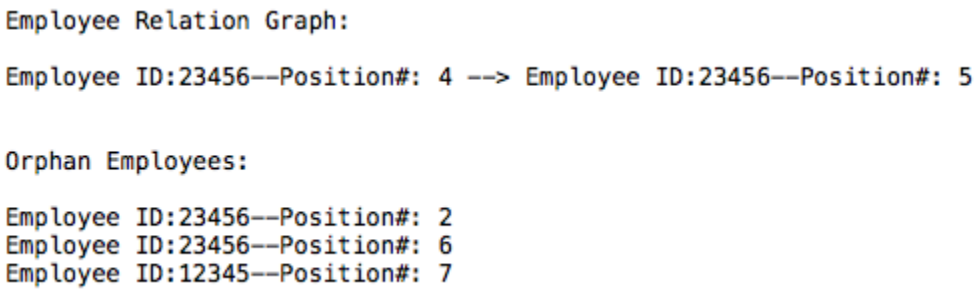
If the answer is yes, it is going to prompt the user to fill in the informations about the employees:

Figure 21. Information Entry of New Employee.



This employee is then going to be added to the network of hospital employee:

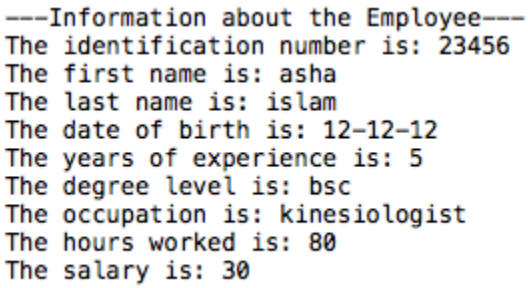
Figure 22. Displaying Graph After Adding Employee.



As seen, the new hospital employee is an orphan. This makes perfect sense, considering that the employee is new, and hasn't gotten the chance to create a relationship.

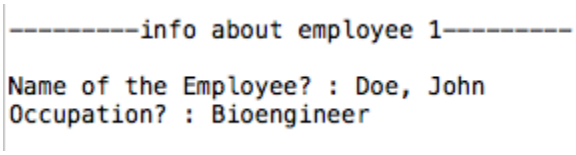
When displaying the information of the employees, we could use the **printInfo** function, which is going to be printing the information about the employee. This is a function that is found in the class HospitalEmployees.

Figure 23. Printing Information About Employee.



If some specific information is wanted from an employee, we could use the get functions. This is illustrated by the screenshot below:

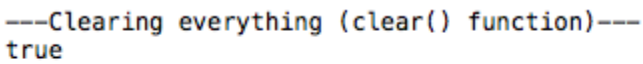
Figure 24. Printing Partial Information About Another Employee.



At the end of this whole process, we are going to be clearing the graph. This means that we are going to be deleting the employees from the relation graph. The result of that is illustrated by the adjacency matrix and the clear function.

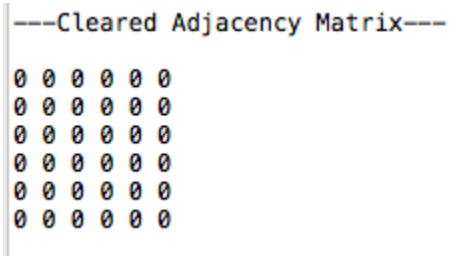
When the clear function is called, it is going to be returning true or false whether it has been done or not.

Figure 25. Clear Function.



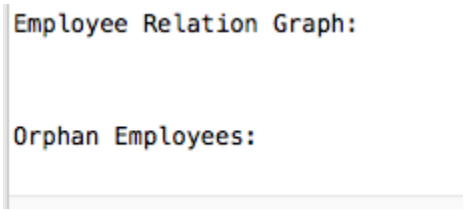
In the adjacency matrix, we expect to see a matrix with only zeros.

Figure 26. Adjacency Matrix with Clearing Function.



In the employee relation graph in the first graph, we could see no vertices (employees) and no orphan employees as well.

Figure 27. Emp



DISCUSSION

When adding a hospital employee into the graph as seen in, the program iterates through the list of employees to verify certain conditions in order to add the vertex. In this case, the program verifies that the number of employees in the hospital's database does not surpass the number of employees that actually work there. If this condition were not met, the employee will not be added into the database. The same applies for the second condition, which is to verify that the ID of the employee entered by the user is not the same as the ID of an employee that already exists in the database. According to the results in *Figure 13*, the hospital employees 1, 2, 3, 4, 5 and 6 have been added successfully into the graph, however when attempting to add employee 4 once again, it returns false, therefore the ID of the employee has been recognized by the system and rejected the addition, following the second condition mentioned. In the case of the hospital employee 6, when they are added to the database, they do not have any relationship with any other employees yet, therefore there is no edge relating them to another employee, thus representing them as an orphan vertex on the graph.

On the other hand, when removing a hospital employee from the database, the employee is removed the program by satisfying the following conditions: there are indeed employees in the list and the ID and the value of the hospital employee entered by the user exist in the database because if they are not found, that would mean that the employee in question is not entered or does not exist. It is seen in the results in *Figure 16* that the system removes the employee and returns true when this is done, therefore the conditions listed above were met. However, the edge, or the relation, that was between an employee and the person who was removed was not deleted successfully. This indicates that when the hospital employee is removed, its vertex is removed from the graph, but the edge relating it to another employee has not been removed since the remaining employee will build a relationship with the next hospital employee. Thus, the relation link will remain present, just not between the same two employees. If there are no other employees left to make a relation, then the remaining hospital employee will be considered as an orphan vertex in the graph as shown in *Figure 17*. This is also an indication that the position of the previously working employee in the array of employees at the hospital is now the position of the next employee that is in the list.

Toward the end of the program in *Figure 21*, the user is asked if they wish to add another employee into the database. If the answer were yes, the user is then asked to enter the new employee's information. Although, once this hospital employee is added, they are represented on the graph as an orphan vertex. This means that this vertex is not connected to any edge and therefore this employee does not have any relationships yet with any other employees (see *Figure 22*). This is relatable to the real-life situation where the new employee does not build connections with others as soon as they arrive. This therefore shows that any new employee added into the database has no relationships, or in other terms, any new vertices are not connected to an edge, therefore do not have a relation, in the graph right away. It would instead be with the addition of other vertices that they could build a relation.

At this same part where the user is asked if they wish to add a hospital employee into the system, they are first asked to enter a "y" for yes or a "n" for no. Seeing the results in *Figure 20*,

when the user enters an answer other than “y” or “n”, the program displays an error message, stating the following:

“Your answer is not valid, please try again.”

This indicates that there is a proper usage of exception handling since the program recognized a problem in the response and threw the exception, which is the error message in this case, whereas when the user enters the right response, the program continues with the code by asking the remaining questions.

The following results in *Figure 23* shows the functionality of the **printInfo()** function. There is a display of all the data of a hospital employee. This shows that the function is created to print the information entered by the user, or that is found in the database for that matter, for a specific employee.

At the very end of the program in *Figure 25*, the program goes through a function called **clean()**. As the results are showing, once this function is executed, there is no more data stored into the matrix and there is no more graph information being displayed, such as any present vertices or edges. This means that the **clean()** function has deleted all information from **vertex** and **edge**, therefore clearing the whole graph’s data.

CONCLUSION

As a conclusion, a program displaying a real-life application has been done, as we have created a graph which demonstrated the relations between hospital employees. It is important to do such mapping, in order to know which healthcare professionals should work in parallel. With the help of many techniques, which include inheritance, polymorphism, operator overloading, as well as exception handling, we were able to store information about hospital employees, create different relations, etc. The various functions that were created for that matter have also been tested by running the program and showed proper functionality.

REFERENCES

n.d (2021). Graph Implementation in C++ Using Adjacency List.
https://www.softwaretestinghelp.com/graph-implementation-cpp/

N.d. Programiz. **https://www.programiz.com/dsa/graph-adjacency-matrix**