The Computer and Electrical Engineering Course Sequence Generator

Ardalan Jamshidi

*Department of Computer Engineering*

*Concordia University*

Montreal, Canada

[ardalan.jamshidi@hotmail.com](mailto:ardalan.jamshidi@hotmail.com) ID#: 27079265

Rihazul Islam

*Department of (program) Engineering*

*Concordia University*

Montreal, Canada

[rihazulislam@gmail.com](mailto:rihazulislam@gmail.com) ID#: 40212505

Younes Meliani

*Department of (program) Engineering*

*Concordia University*

Montreal, Canada

[younesmeliani@hotmail.com](mailto:younesmeliani@hotmail.com) ID#: 40211009

*Abstract*—This paper describes an object-oriented application of a graph consisting of the first and second semester course preferences a student in the Computer or Electrical Engineering undergraduate degree must follow. In addition, the program allows the user to request the possible course sequences available, as well as the query of a graph by inputting a specific course name, prerequisite link between two courses and show the course sequence followed by an inputted introductory course name.

Keywords—graph, node, edge, directed, undirected, string, inheritance, polymorphism, exception handling,

# Introduction

This report showcases and describes the structure, design and integrity of a graph implementation program in C++. It does so by providing a brief description of required classes and relationships amongst them with the support of a class diagram following UML conventions. Additionally, there are various non-trivial methods utilized in the project, the employment of inheritance, polymorphism and exception handling techniques will be described for a deeper analysis of the program. Ultimately, a final evaluation of the program with three different test cases will be applied to demonstrate the integrity and functionality of the latter, all while applying the black-box testing methodology.

# Classes and employed structure

The first step towards the creation of a graph implementation program would be to create two standalone classes “Node” and “Edge” (Fig. 1.1: the two down-most classes in orange inside the class diagram) which would take an object declared in the driver file and perform the necessary getter and setter functions. Table I and II represent the various functions of classes “Node” and “Edge” respectively.

Next, it is necessary to create two more classes “DirectedGraph” and “UndirectedGraph” (Fig. 1.1: the left-most and right-most classes in vivid green inside the class diagram) to create the distinction between directed and undirected graph representation. Since both previously discussed classes have different implementations, then consequently, the need of a base abstract class “Graph” (Fig. 1.1: the top class in light green inside the class diagram) is optimal in order for the two classes mentioned previously to inherit from pure virtual functions. Tables III through V represent the various functions of classes “Graph” , “DirectedGraph” and “UndirectedGraph” respectively.

1. Class “Node” Attributes

|  |  |
| --- | --- |
| **Class Attribute** | **Attribute Description** |
| Node() | Default constructor |
| Node(std::string) | Regular constructor |
| Node(const Node&) | Copy constructor |
| ~Node() | Destructor |
| void setcoursename(std::string) | Sets the string value “coursename” |
| std::string getcoursename() | Gets the string value “coursename” |

1. Class “Edge” Attributes

|  |  |
| --- | --- |
| **Class Attribute** | **Attribute Description** |
| Edge() | Default constructor |
| Edge(Node&,Node&) | Regular constructor |
| Edge(const Edge&, const Edge&) | Copy constructor |
| ~Edge() | Destructor |
| Node getStartingNode() | Gets the string value of the starting vertex |
| void setStartingNode(Node) | Sets the string value of the starting vertex |
| Node getEndingNode() | Gets the string value of the ending vertex |
| void setEndingNode(Node) | Sets the string value of the ending vertex |

1. Class “Graph” Attributes

|  |  |
| --- | --- |
| **Class Attribute** | **Attribute Description** |
| virtual bool addNode(Node&) = 0 | Adds the node object to graph |
| virtual bool removeNode(Node&) = 0 | Removes the node object from graph |
| virtual bool addEdge(Edge&) = 0 | Adds the edge object to graph |
| virtual bool removeEdge(Edge&) = 0 | Removes the edge object from graph |
| virtual void display1() const = 0 | display vertices and edges in graph |
| virtual bool doesEdgeexist(Edge&)=0 | function to check if edge exists in a graph (#7 function) |
| virtual bool doesvertexexist(std::string)=0 | function to check if vertex/node is contained in graph (#8) |

1. Class “DirectedGraph” Attributes

|  |  |
| --- | --- |
| **Class Attribute** | **Attribute Description** |
| DirectedGraph() | Default constructor |
| DirectedGraph(int,int) | Regular constructor |
| DirectedGraph(const DirectedGraph&,const DirectedGraph&) | Copy constructor |
| ~DirectedGraph() | Destructor |
| int getnumnodes() | Gets the int value of the number of nodes |
| void setnumnodes(int) | Sets the int value of the number of nodes |
| int getnumedges() | Gets the int value of the number of edges |
| void setnumedge(int) | Sets the int value of the number of edges |
| virtual bool addNode(Node&) | Adds the node object to graph |
| virtual bool removeNode(Node&) | Removes the node object from graph |
| virtual bool addEdge(Edge&) | Adds the directed edge object to graph |
| virtual bool remove(Edge&) | Removes the directed edge object from graph |
| virtual void display1() const | Lists all entered nodes and edges |
| void showpaths() | Lists all possible paths of a graph |
| void showPathWithStartingNode(std::string) | Lists possible path of a requested starting vertex that exists in a graph |
| virtual bool doesEdgeexist(Edge&) | Shows graph query by requesting an edge that exists in a graph |
| virtual bool doesvertexexist(std::string) | Shows graph query by requesting a vertex that exists in a graph |
| bool vertexSearch(Node&) | Checks to see if a vertex already exists in a graph |

1. Class “UndirectedGraph” Attributes

|  |  |
| --- | --- |
| **Class Attribute** | **Attribute Description** |
| DirectedGraph() | Default constructor |
| DirectedGraph(int,int) | Regular constructor |
| DirectedGraph(const DirectedGraph&,const DirectedGraph&) | Copy constructor |
| ~DirectedGraph() | Destructor |
| virtual bool addNode(Node&) | Adds the node object to graph |
| virtual bool removeNode(Node&) | Removes the node object from graph |
| virtual bool addEdge(Edge&) | Adds the undirected edge object to graph |
| virtual bool remove(Edge&) | Removes the undirected edge object from graph |
| virtual void display1() const | Lists all entered nodes and edges |
| virtual bool doesEdgeexist(Edge&) | Shows graph query by requesting an edge that exists in a graph |
| virtual bool doesvertexexist(std::string) | Shows graph query by requesting a vertex that exists in a graph |

An UML class diagram “Fig. 1.1” is created to further demonstrate the relationships between the discussed classes of the graph implementation program. In the diagram, there exists three types of relationships between the required classes; a straight arrow to indicate an inheritance of a derived class to a base class, a solid black diamond with an “X” pointing to an arrow and a dotted arrow to indicate an association and dependency, respectively, of a function in one class calling another function in another class without the method of inheritance.

An example of inheritance would be the function “display1()” in the class “DirectedGraph” inheriting its attributes and methods from the base class “Graph”. An example of association (or dependency) would be the implementation of the vector of type string called “nodee” in the class “DirectedGraph” using the functions from the class “Node”.



Figure 1.1 A class diagram in UML of the program with all classes and their functions, as well as the relationships between them.

# Applied Non-trivial Methods

There are four main non-trivial methods of member functions, amongst others, utilized in the program. The goal of the latter is to develop the required capabilities of a graph implementation program that cannot be achieved by merely employing a trivial set of functions.

## The Method of Displaying All Paths of a Graph

The member function “showpaths()” (Fig. 2.1) outputs all possible paths of each linking vertices.

First, the virtual function creates two vectors of type string, namely “paths”, to store each path, and “checked”, to store the sequence of each path in order to avoid repeating a path twice. Next, using a for loop with iterations, the vector “edges” is processed using member functions “getStartingNode()”, “getEndingNode()” and “getcoursename()” from classes “Node” and “Edge” to acquire the paths into the “paths” vector, as well as in the “checked” vector. Another for loop within the first loop is also implemented to then write the non-zero paths the two previously discussed vectors. As the primary first loop iterates, every node is processed to observe their linking edges and write a corresponding path.

Once the entire process is finished, a third and independent for loop outputs each paths stored in the vector “paths” to finish the function’s goal.

## The Method of Displaying the Path of a Graph with a Starting Node

The member function “showPathWithStartingNode()” (Fig. 2.2) outputs the possible path of each linking vertices from an initial specified vertex.

At first, the non-virtual function, exclusive to the “DirectedGraph” class, employs the same protocol as “showpaths()” with the vectors “paths” and “checked”, as well as the for loops to check, write and output the desired path. The main difference here is the passed value of a specific string “a”, which verifies if such vertex exists within the graph, by using the “compare” function. Once the vertex is located, then the function’s processing will continue, otherwise nothing will be outputted and “showPathWithStartingNode(std::string a)” will terminate.

## The Methods of Querying a Graph by Either a Specified Vertex or Edge.

The member functions “doesEdgeexist()” and “doesvertexexist()” (Fig. 2.3) output the Boolean result of their query inside a graph.

To begin with, in “doesvertexexist()”, we pass a string value of a vertex, while for “doesEdgeexist()”, we pass an object of an edge. Once the required values are passed, a for loop with iteration verifies the vector of type string “edgess” for the appearance of the passed value using the very same functions as mentioned in “showpaths()” to output whether or not the requested vertex or edge exists in the graph.

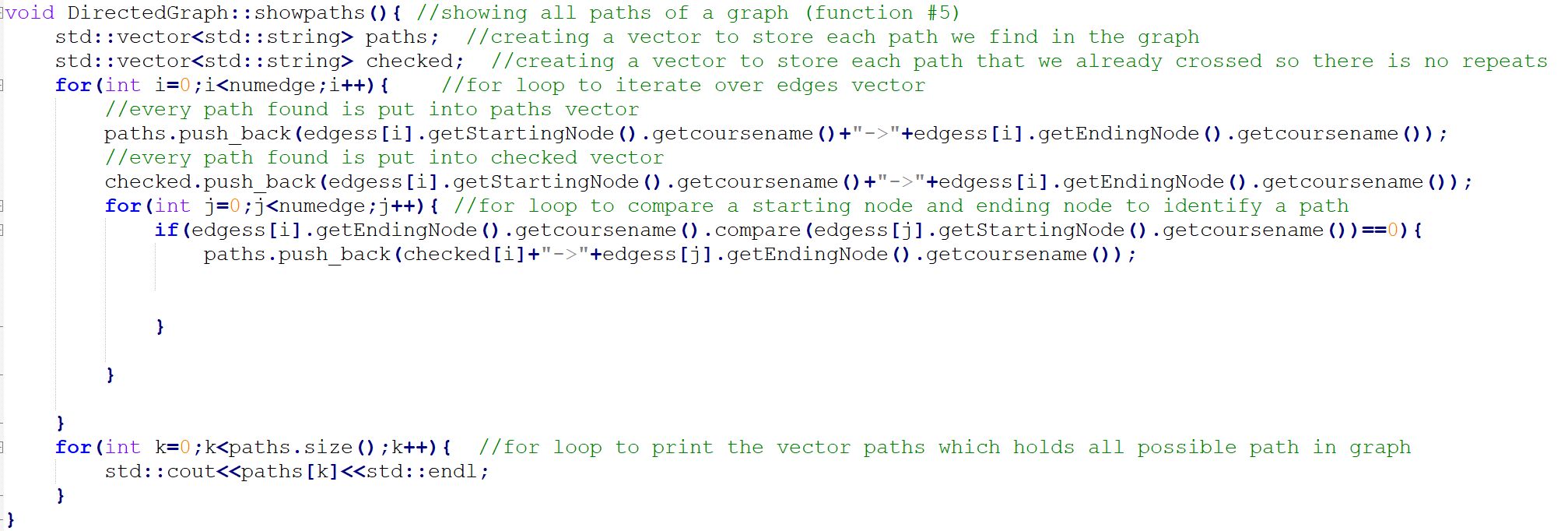


Fig. 2.1 The function implementation of “showpaths()”

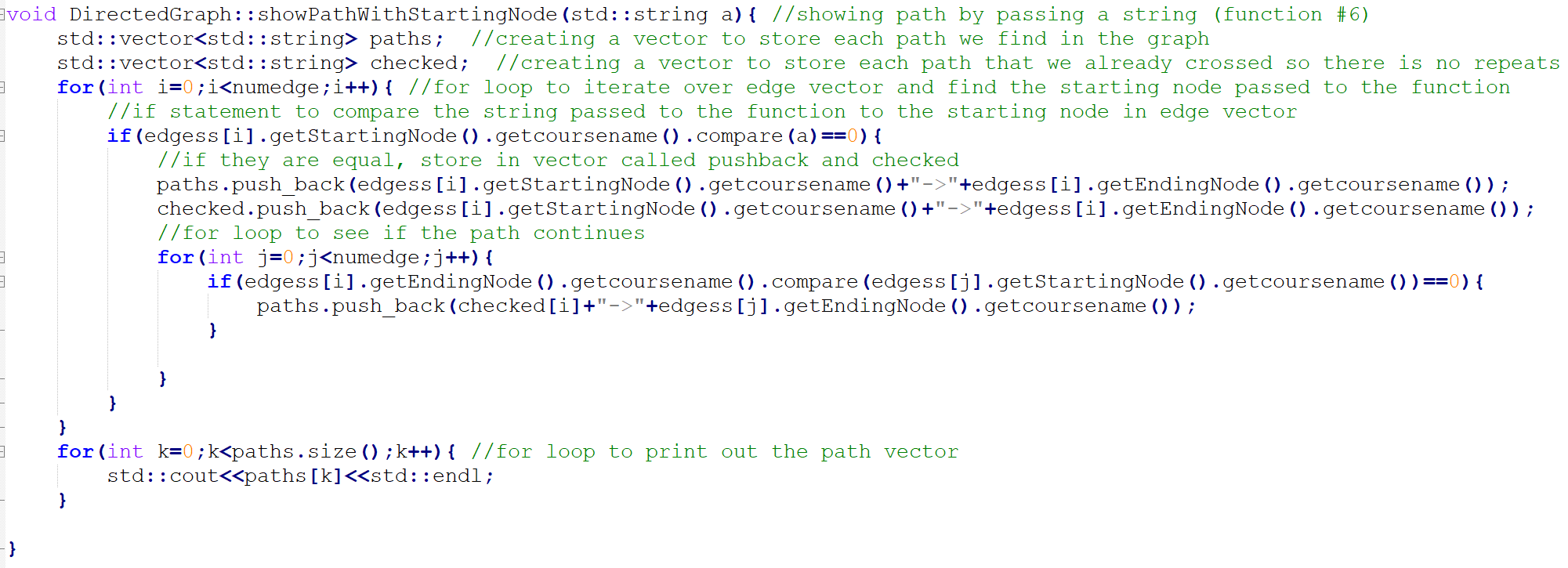


Fig. 2.2 The function implementation of “showPathWithStartingNode()”



Fig. 2.3 The function implementations of “doesEdgeexist()” and “doesvertexexist()”

# Required Techniques for Implementation

In the graph implementation program, there are three techniques employed and deemed vital for the functionality of the latter: inheritance, polymorphism, and exception handling.

## The Technique of Inheritance

The use of inheritance in this program is advantageous to promote code reusability, facilitate class library creation and to avoid code redundancy [1]. Inheritance is applied by simply stating the following command for the class headers of “DirectedGraph” and “UndirectedGraph”:

* Class DirectedGraph: public Graph
* Class UndirectedGraph: public Graph

This would allow the two derived classes to access the values of the base class “Graph”.

## The Technique of Polymorphism

This use of inheritance in addition to polymorphism is highly beneficial in incorporating function overloading to implement a function in a base abstract class with additional arguments in the derived classes, specific to their derived class’ purpose. The latter would also allow multiple forms of initializations with different constructors [2]. Polymorphism is applied by first creating a pure abstract function in the base class “Graph” without implementation:

* virtual void display1() const = 0;

, followed by its virtual implementation in the derived classes “DirectedGraph” and “UndirectedGraph”:

* virtual void display1() const;

From thereon, different arguments are coded according to the classes’ unique purposes.

## The Technique of Exception Handling

The use of exception handling in this program is advantageous in creating an error handling functionality to further guide the user in inputting the acceptable inputs and to prevent the exiting of a program. Another benefit would be the custom definition of the errors that should be handled [3]. Exception handling is applied in the program by utilizing the “throw”, “catch” and “try” commands to return a Boolean and prevent the application to crash or enter an infinite loop upon the entry of erroneous values (Fig. 3.3).

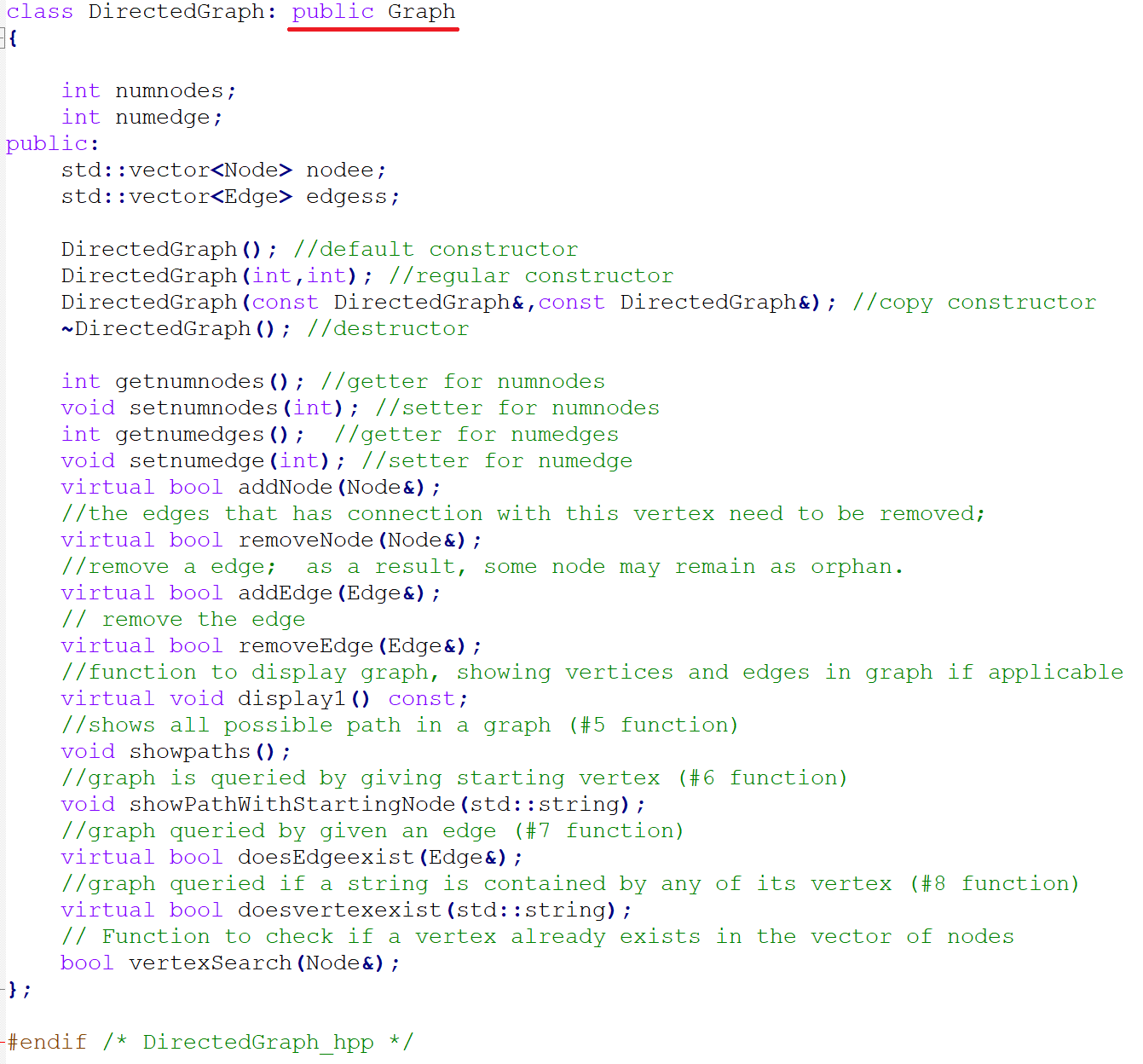


Fig. 3.1 The use of inheritance in classe “DirectedGraph”.

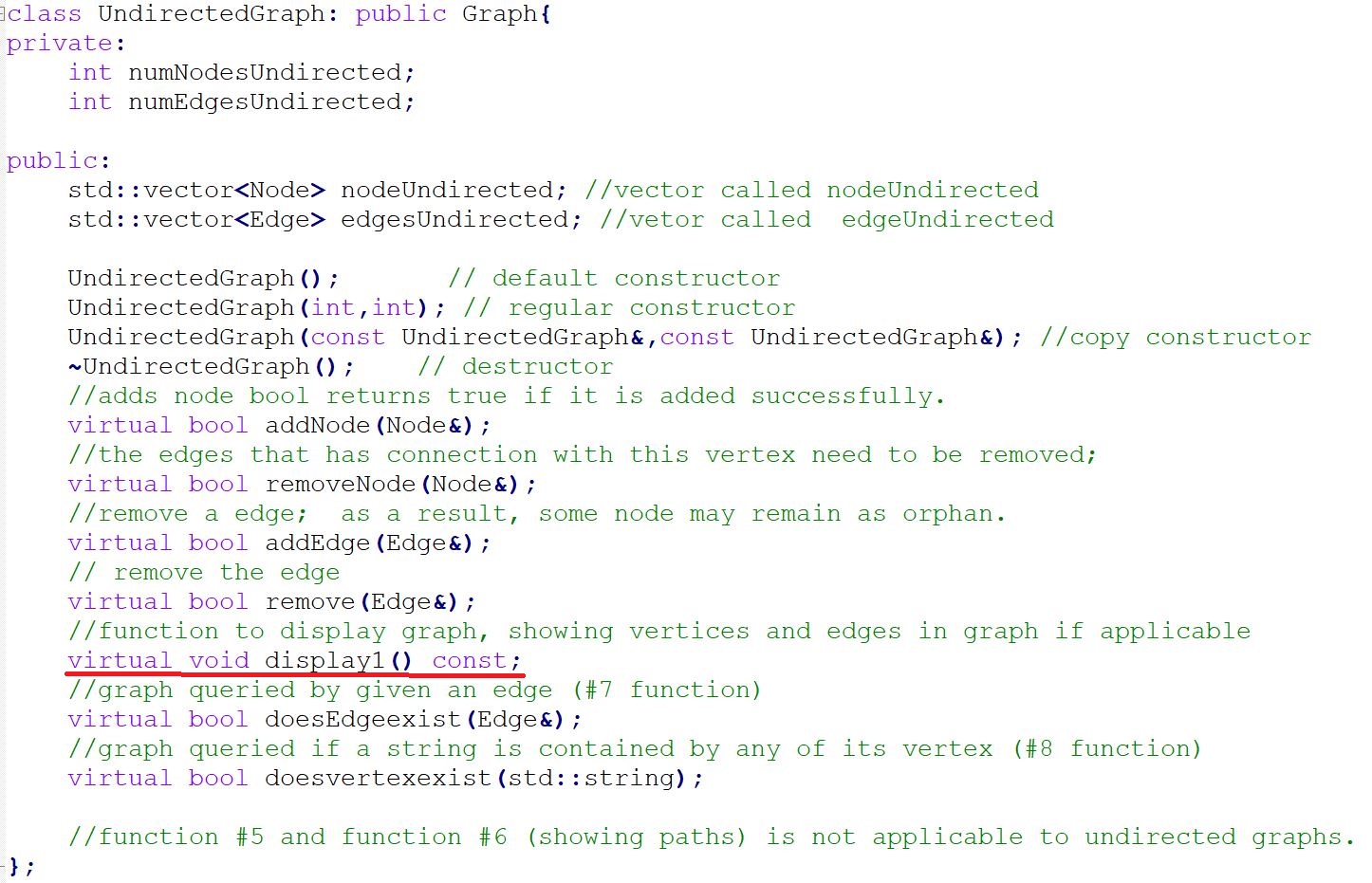
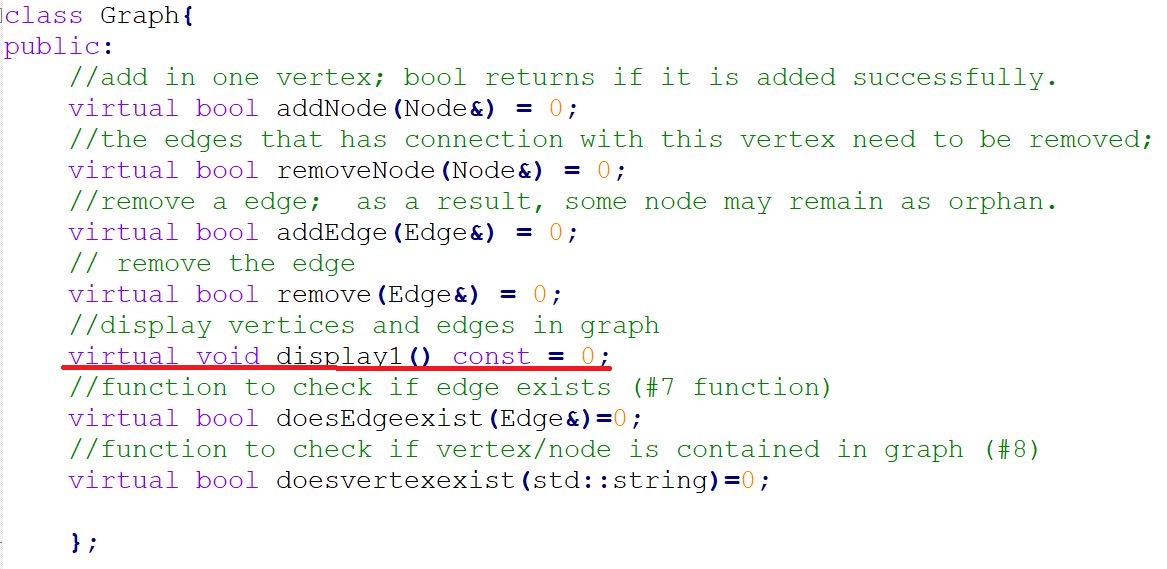


Fig. 3.2 The use of polymorphism in classes “Graph” and “UndirectedGraph”.

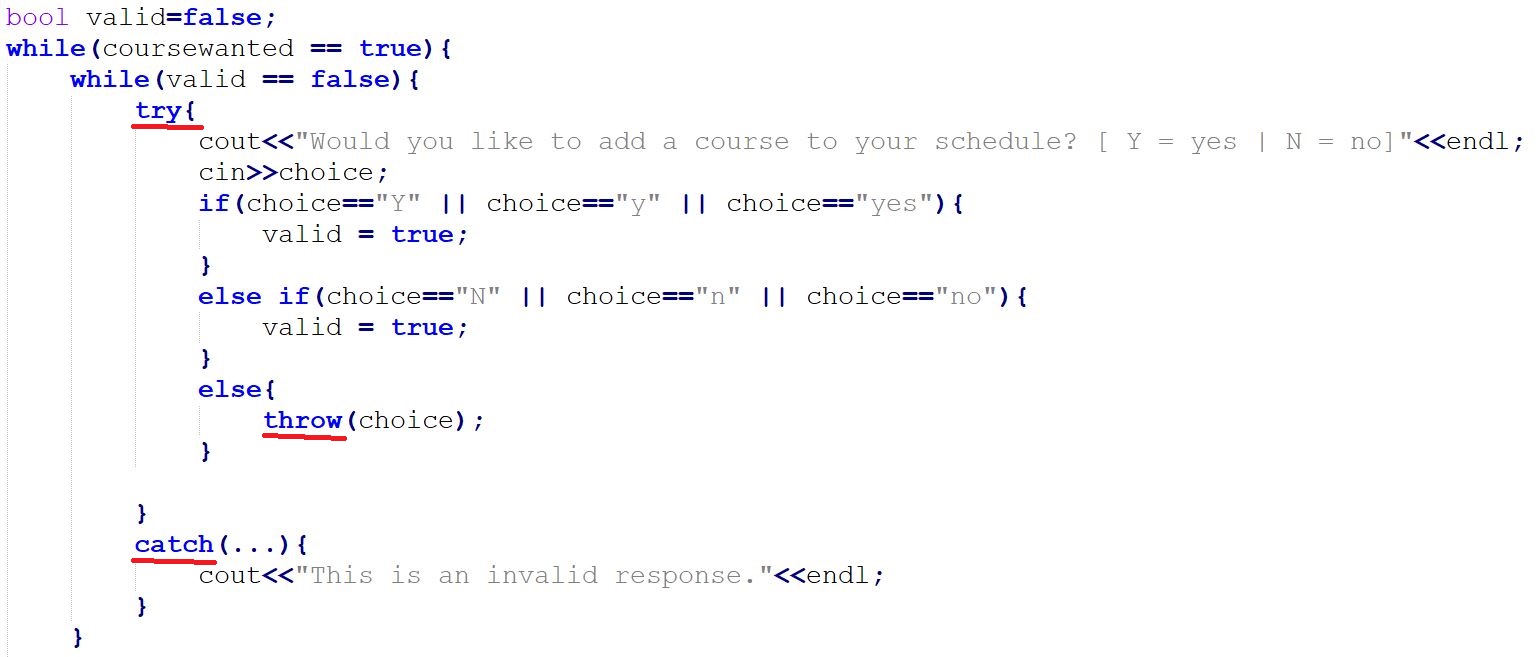
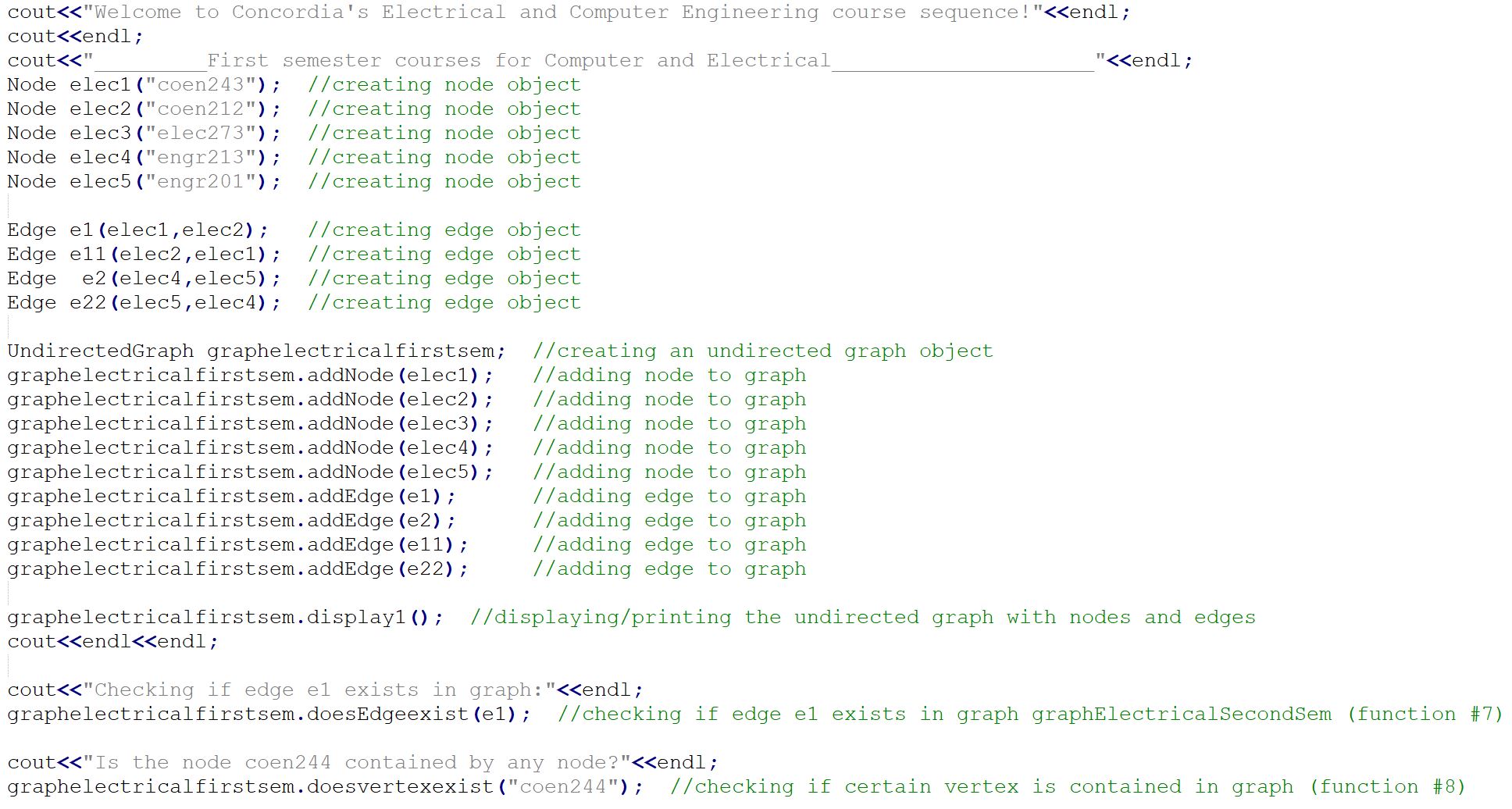


Fig. 3.3 The use of exception handling in the driver source file.

# Black-Box Testing of the Program

## The Actual Program

The ensure the full integrity and functionality of a program in any language, a testing protocol should take place following the black-box testing methodology. In this case, there are three different cases of entries tested in the driver source file to achieve the latter. Regularly, the program will run the specified set of entries in Fig 4.1 and output the results in Fig. 4.3 following the graph diagram in Fig. 4.2:



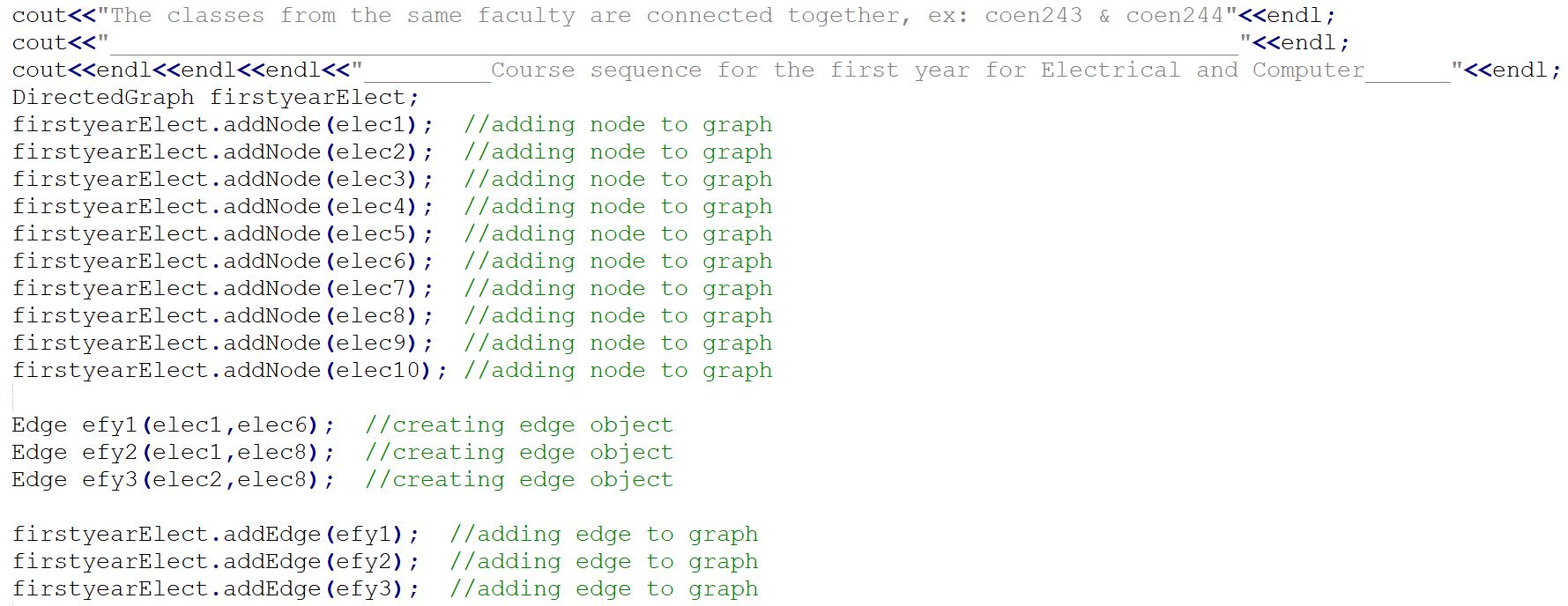
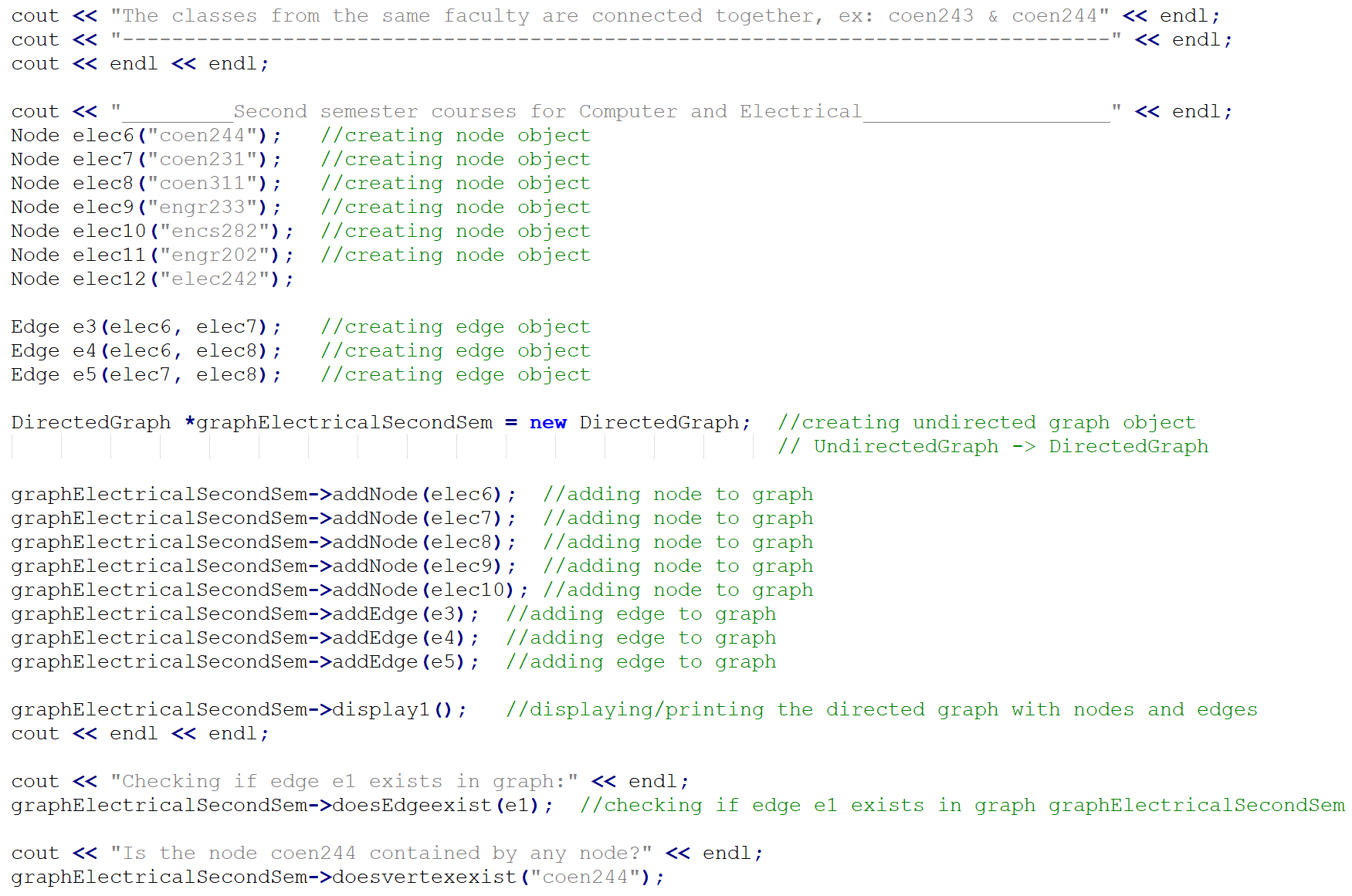


Fig. 4.1 The actual data in the driver source file.

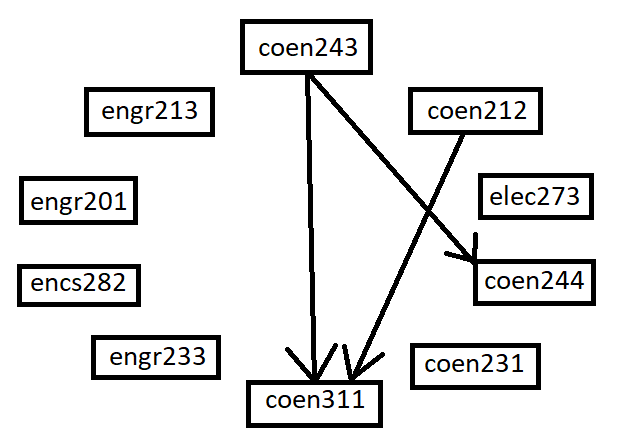


Fig. 4.2 The graph diagram of the code in Fig. 4.1

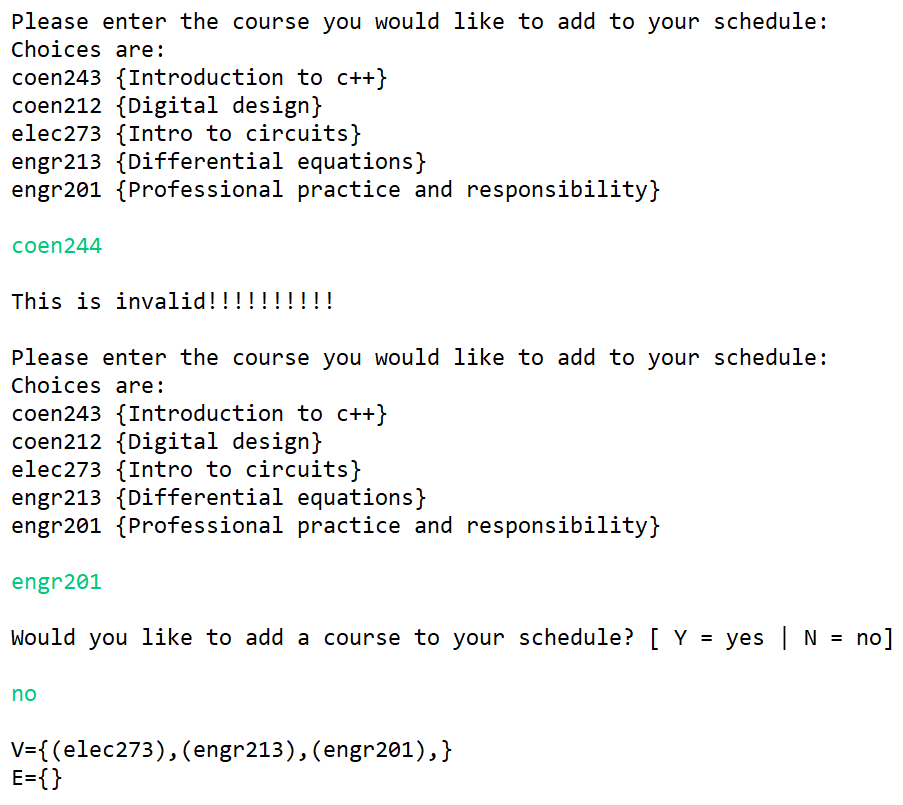
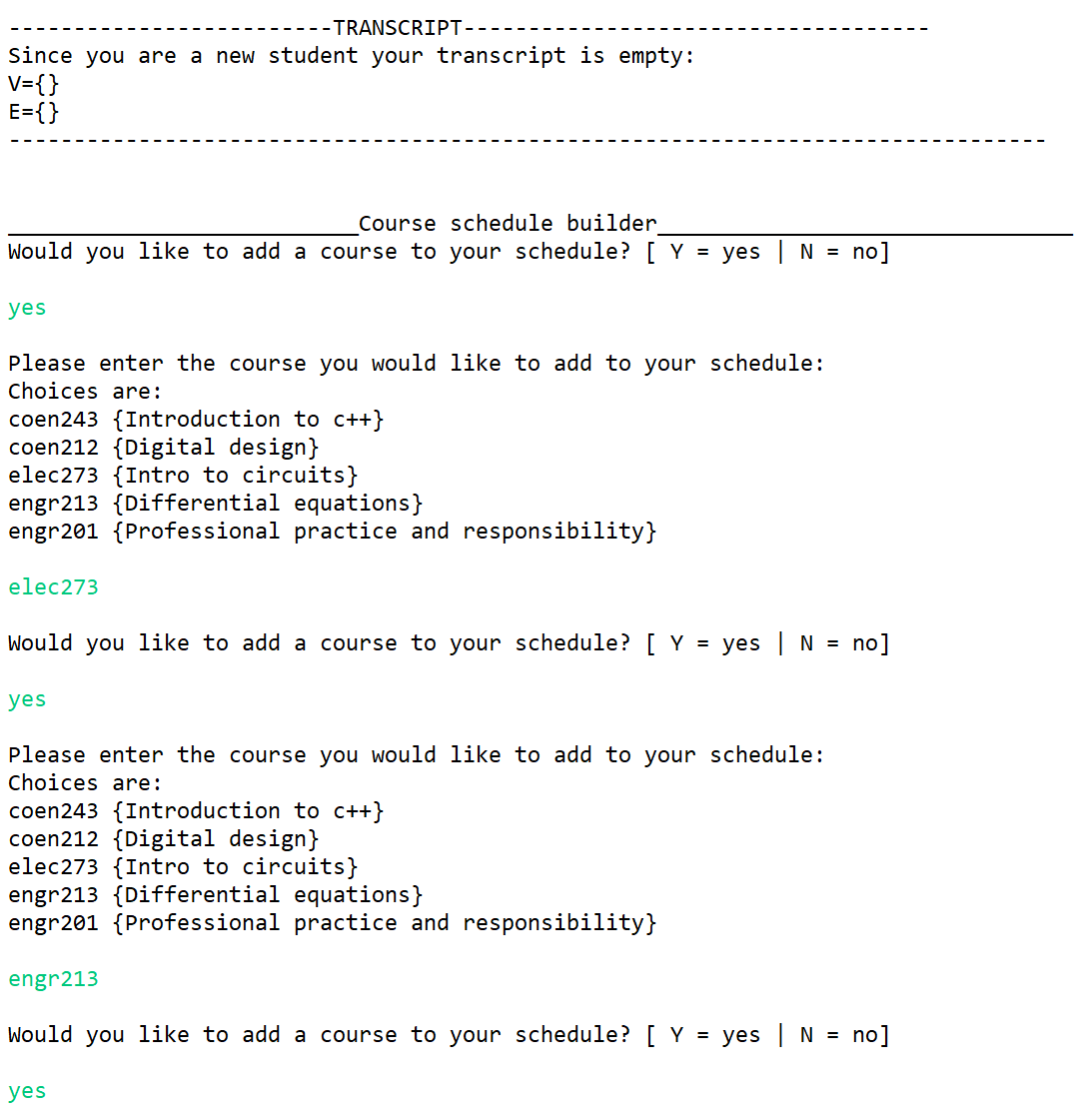
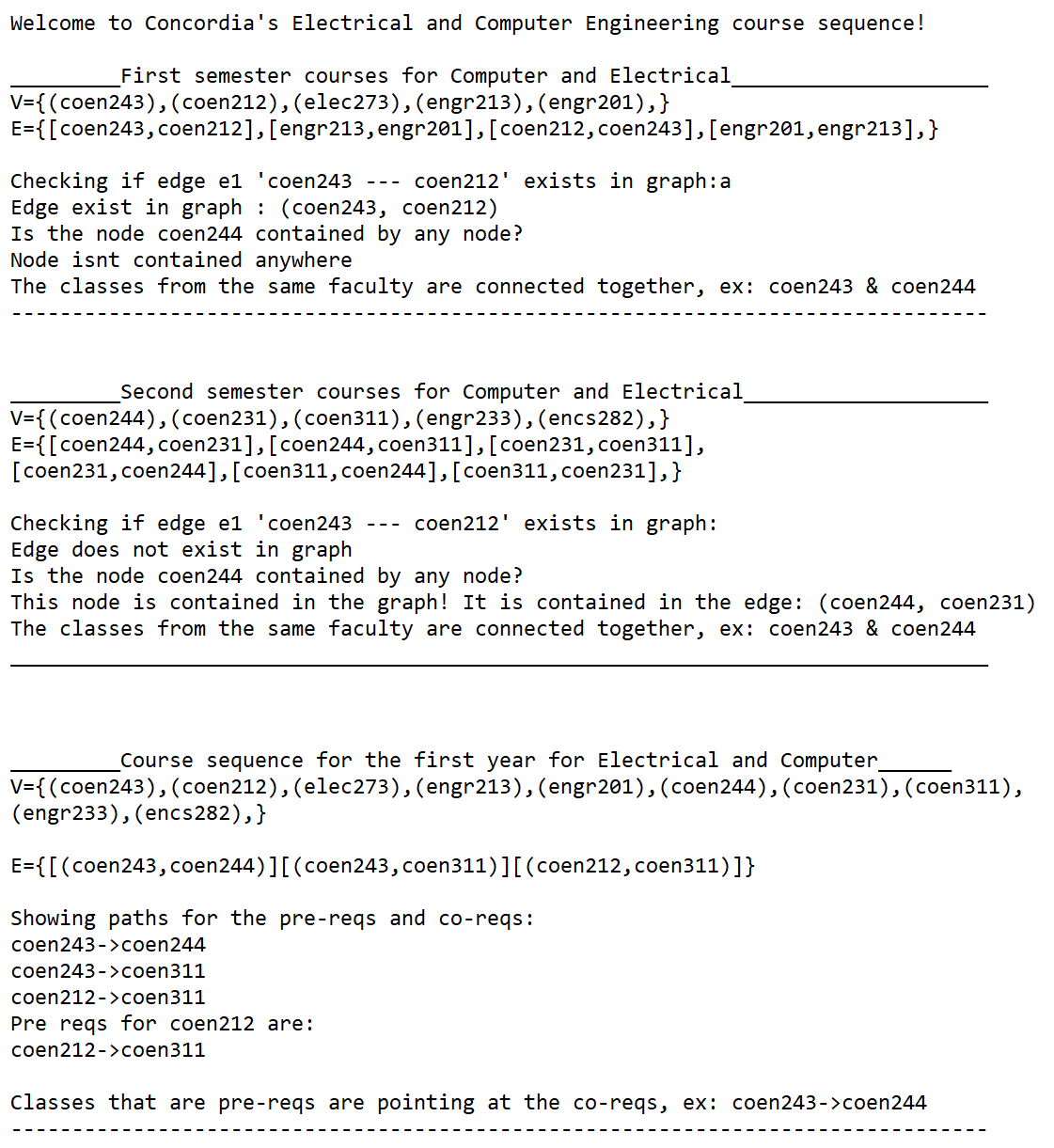


Fig. 4.3 The output of the original driver source file.

In this section, the application will be tested by using the black box techniques of Boundary Value Analysis , Error Guessing and Graph-Based Testing [4].

Normally, the program will demonstrate the paths involved in its graph representation without experiencing any runtime, linker or logical errors. The paths outputted follow the graph in Fig. 4.2 in an identical manner and invoking undefined entries won’t result in a crash.

## The Test Case Scenario

In the first test case, the value of the nodes, edges and the number of edges is altered to test the functionality of the various graph functions. Fig. 4.4.1 and 4.4.2 demonstrates the driver source code with red markings as indications of altered values and objects. Once the coding is complete, the output of the application in Fig. 4.6 must be in accordance with the new graph diagram in Fig. 4.5 to show the success of the test case scenario.

Text

Description automatically generated with medium confidence

Fig. 4.4.1 The test case data in the driver source file with the red marking indicating the changes made to the original source file.

Text

Description automatically generated

Fig. 4.4.2 The test case data in the driver source file with the red marking indicating the changes made to the original source file.

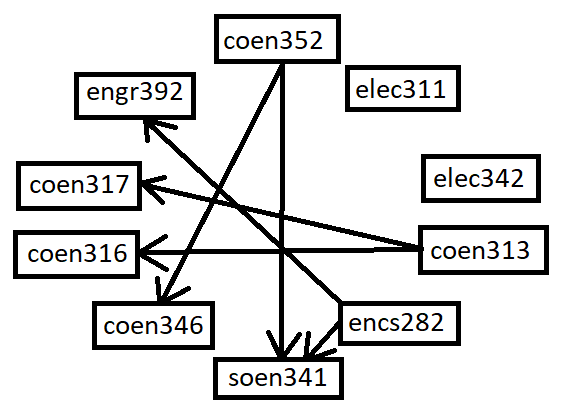


Fig. 4.5 The test case graph diagram of the codes in Fig. 4.4.1 and 4.4.2.



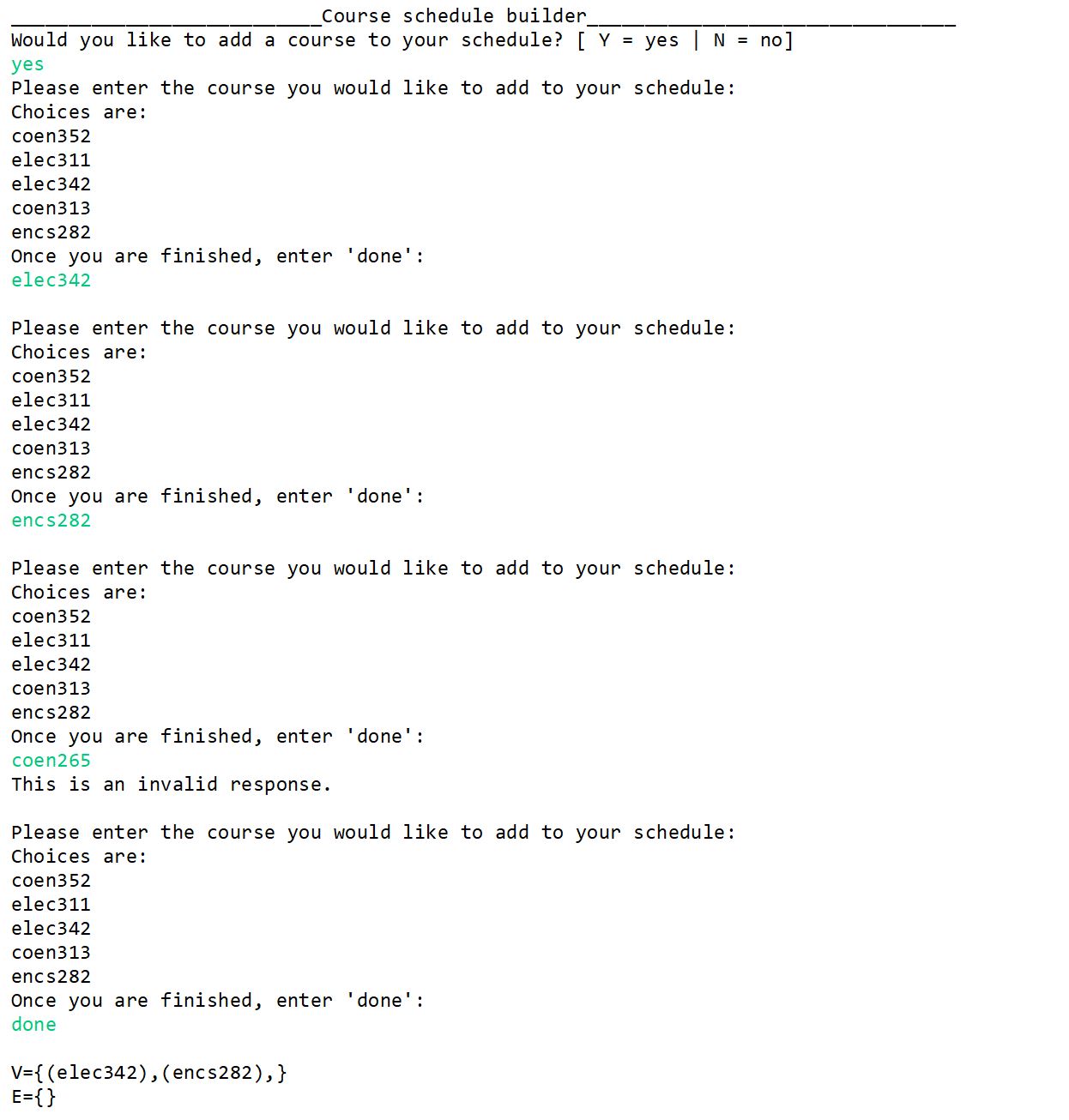
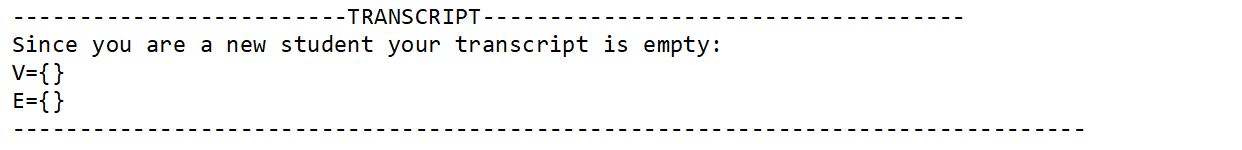


Fig. 4.6 The output of the test case scenario.

The first technique used for testing is the Graph-Based Testing, to show if a different set of nodes and edges with a different order will produces errors or not.

At first, the output generates the correct number of nodes and undirected edge for the first semester, as well as for the fourth semester. The function check of the existing edge ‘coen352 --- encs282’ is also successful, as it exists in the third semester, but not the fourth. The same is also performed for the query of the node ‘encs282’, which returns positive in the third semester and negative in the fourth semester.

Afterwards, the complete set of nodes and edges are completely displayed, following the Depth First Search method of outputting all the paths the graph could take. The paths are all in accordance with the diagram in Fig. 4.5 and a function check for showing all paths of the node ‘encs282’ results in a successful display of the two paths that are involved.

The latter successfully concludes the passing of the Graph-Based Testing in the test case scenario without the occurrence of any runtime, linker or logical errors.

The second and third techniques of Boundary Value Analysis and Error Guessing will be tested in the second portion of the output to observe if any erroneous input values out of the acceptable boundaries produce any kind of errors in the output.

At this point of the application, a Course Schedule Builder sub application is launched, where the user must enter the correct inputs for the program to exit successfully.

Once the user enters ‘yes’ as an input, the program allows the user to choose between a set of values to be added to the course sequence. Two existing strings ‘elec342’ and ‘encs282’ are entered and added with success, while a non-existing string ‘coen265’ out of bounds is rejected by the exception handling portion of the program with an error message. The application is thereafter ended by the entry of ‘done’ to display the added courses and possible undirected edge between them.

Therefore, the Boundary Value Analysis and Error Guessing techniques were passed with success, where no runtime, linker or logical errors were produced by the entry of out-of-bound or incorrect entries. In addition, no instances of any crashes of infinite loops were ever detected during the entire debugging of the Course Sequence Generator program.

Conclusions

In conclusion, the Course Sequence Generator program was successfully implemented, as well as fully tested by creating the appropriate classes and necessary non-trivial member functions. In addition to the latter, three implementation techniques of inheritance, polymorphism and exception handling were employed and demonstrated by an UML representation of the application’s class diagram to showcase the completeness of the coding. Once the application is complete, a black box test case was performed to ensure the functionality and integrity of the program.

Acknowledgment

Thanks to professor Malleswara Talla and tutorial instructor Shirin Panahi Moghadam for their dedicated guidance and support of the project. Their additional contribution of clarifying project terms and requirements has allowed us to properly prepare the work needed.

References

1. C. Singh, “Inheritance in C++,” beginnersbook.com, 11-Sep-2017. [Online]. Available: https://beginnersbook.com/2017/08/cpp-inheritance/. [Accessed: 05-Apr-2022].
2. “Polymorphism: Definition, Examples, Types &amp; Advantages,” Teach Computer Science, 09-Feb-2021. [Online]. Available: https://teachcomputerscience.com/polymorphism/. [Accessed: 10-Apr-2022].
3. “Exception Handling in C++,” GeeksforGeeks, 03-Feb-2021. [Online]. Available: https://www.geeksforgeeks.org/exception-handling-c/. [Accessed: 12-Apr-2022].
4. Vani, “Black Box Testing: An In-depth Tutorial with Examples and Techniques,” *Software Testing Help*, 27-Mar-2021. [Online]. Available: https://www.softwaretestinghelp.com/black-box-testing/. [Accessed: 12-Apr-2022].