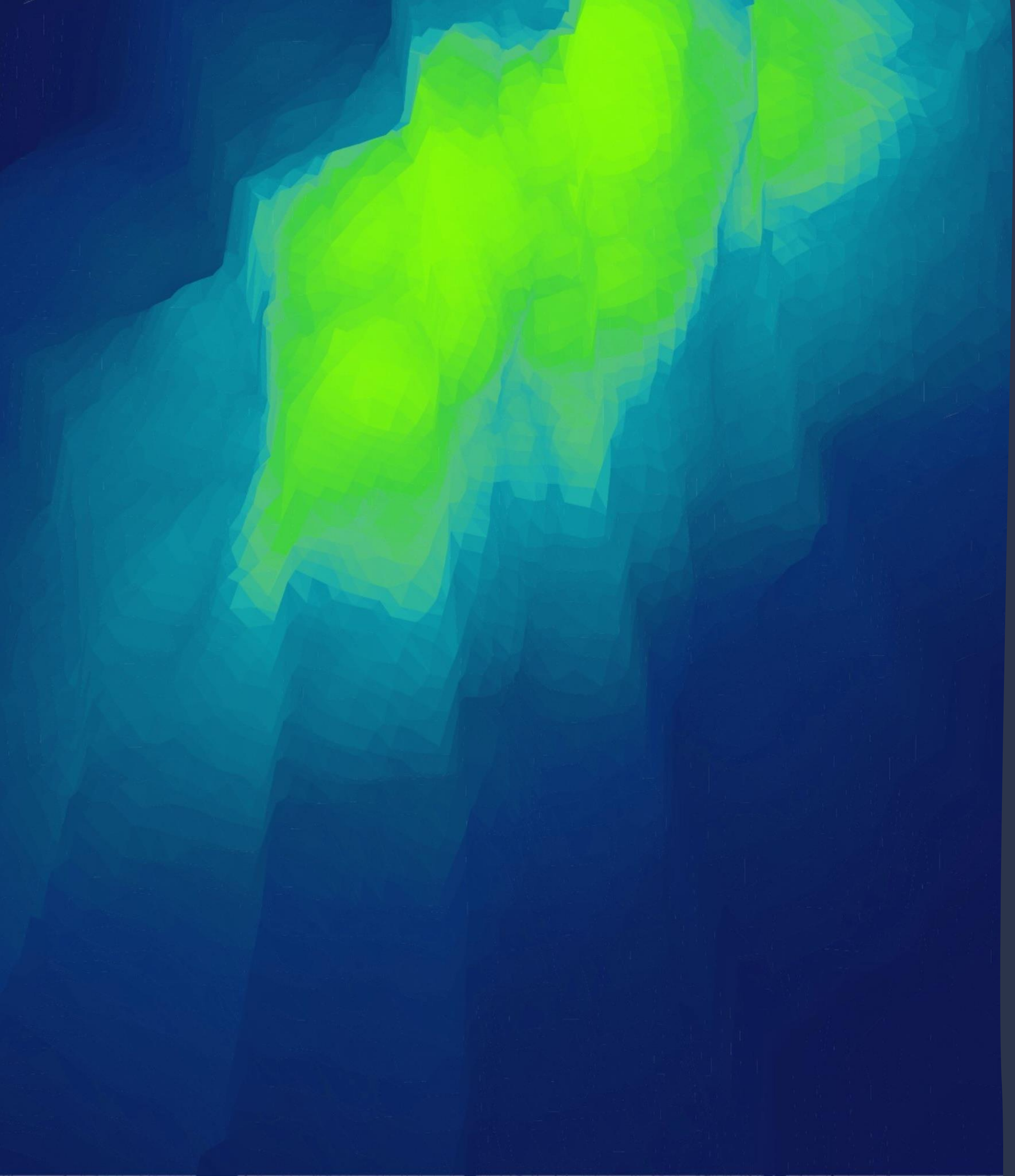
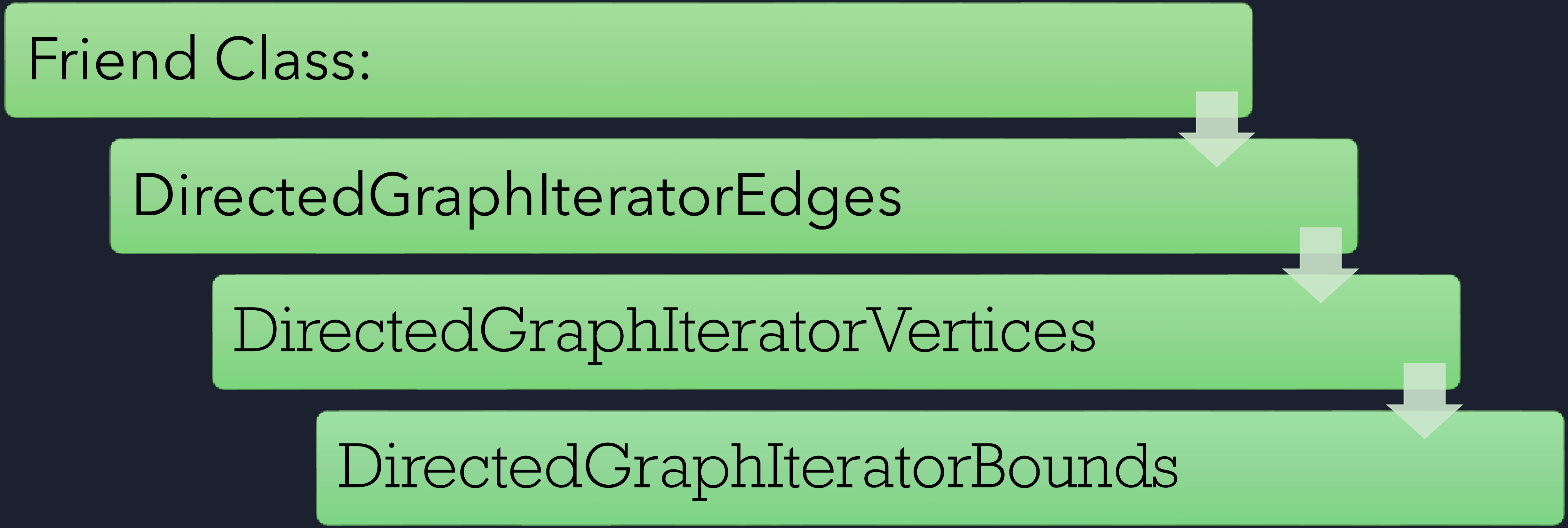


Graphs-2023-2024 Practical Work No.1

Assignment made by Cora Ioan Alexandru

Group 912

C++ Implementation of class DirectedGraph



Friend Class:

DirectedGraphIteratorEdges

DirectedGraphIteratorVertices

DirectedGraphIteratorBounds

Private Components

- Number_of_vertices (int)
- Number_of_edges (int)
- Vertices (vector<string>)
- Edges (map <string, vector <string>>)
- Cost (map <tuple<string, string>, int>)
- InDegree (map <string, int>)
- File (string)

Private Functions

- InitializeEdge(string vertex) ->void
- InitializeInDegree(string vertex) -> void
- FindInVector(vector<string> vector, string ElementToCheck) -> bool
- IterateEdgesHelper(void) ->vector <tuple<string, string>>

Public Functions 1/3

- DirectedGraph(string file_name = "graph1k.txt") -> Constructor
- DirectedGraph(DirectedGraph& other) -> Copy Constructor
- AddVertex(string vertex) -> void
- AddEdge(string start, string end) -> void
- DeleteVertex(string vertex) -> void
- DeleteEdge(string start, string end) -> void
- IterateInbounds(string vertex) -> DirectedGraphIteratorBounds

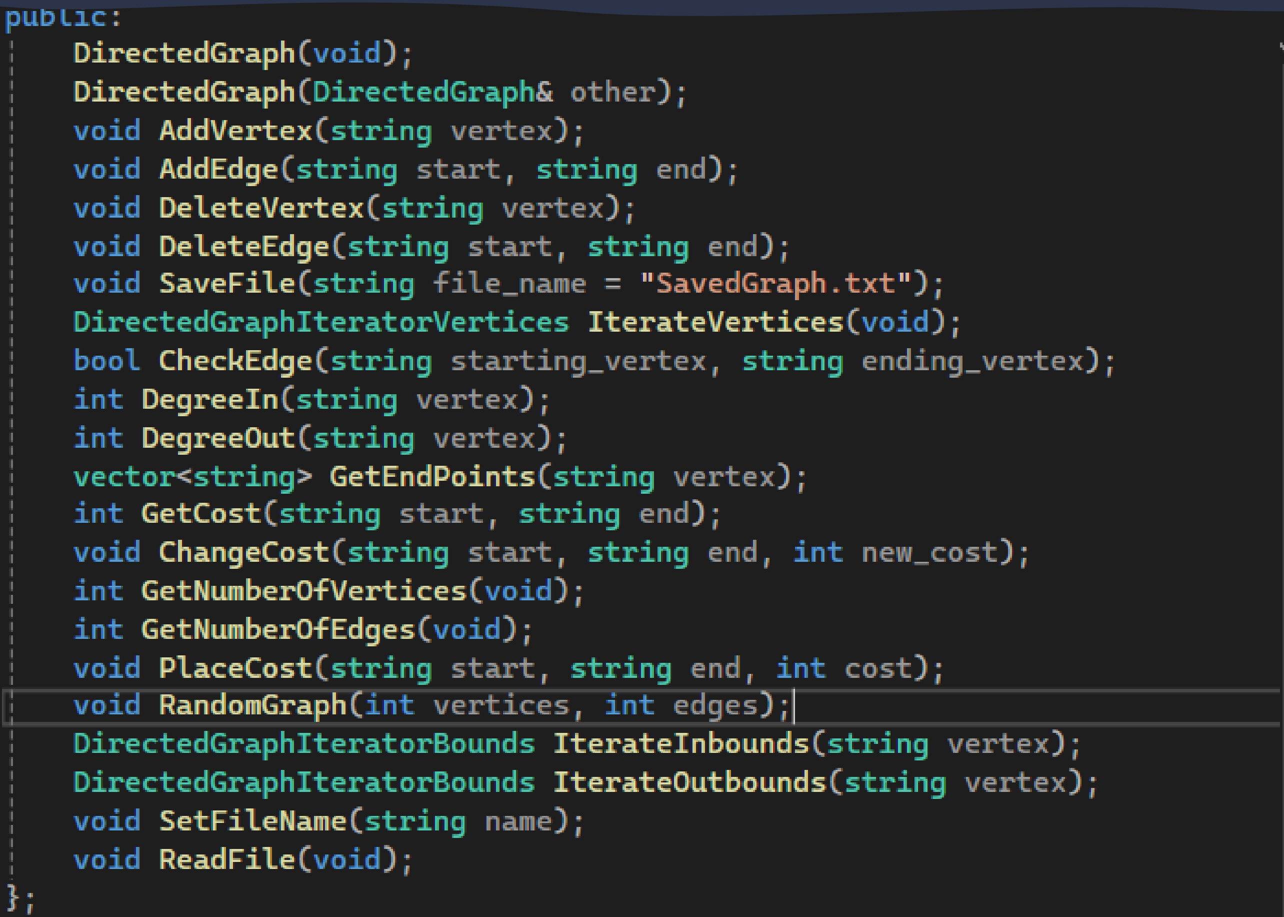
Public Functions 2/3

- SaveFile() -> void
- IterateVertices(void) -> DirectedGraphIteratorVertices
- CheckEdge(string starting_vertex, string ending_vertex) -> bool
- DegreeIn(string vertex) -> int
- DegreeOut(string vertex) -> int
- GetEndPoints(string vertex) -> vector<string>
- GetCost(string start, string end) -> int

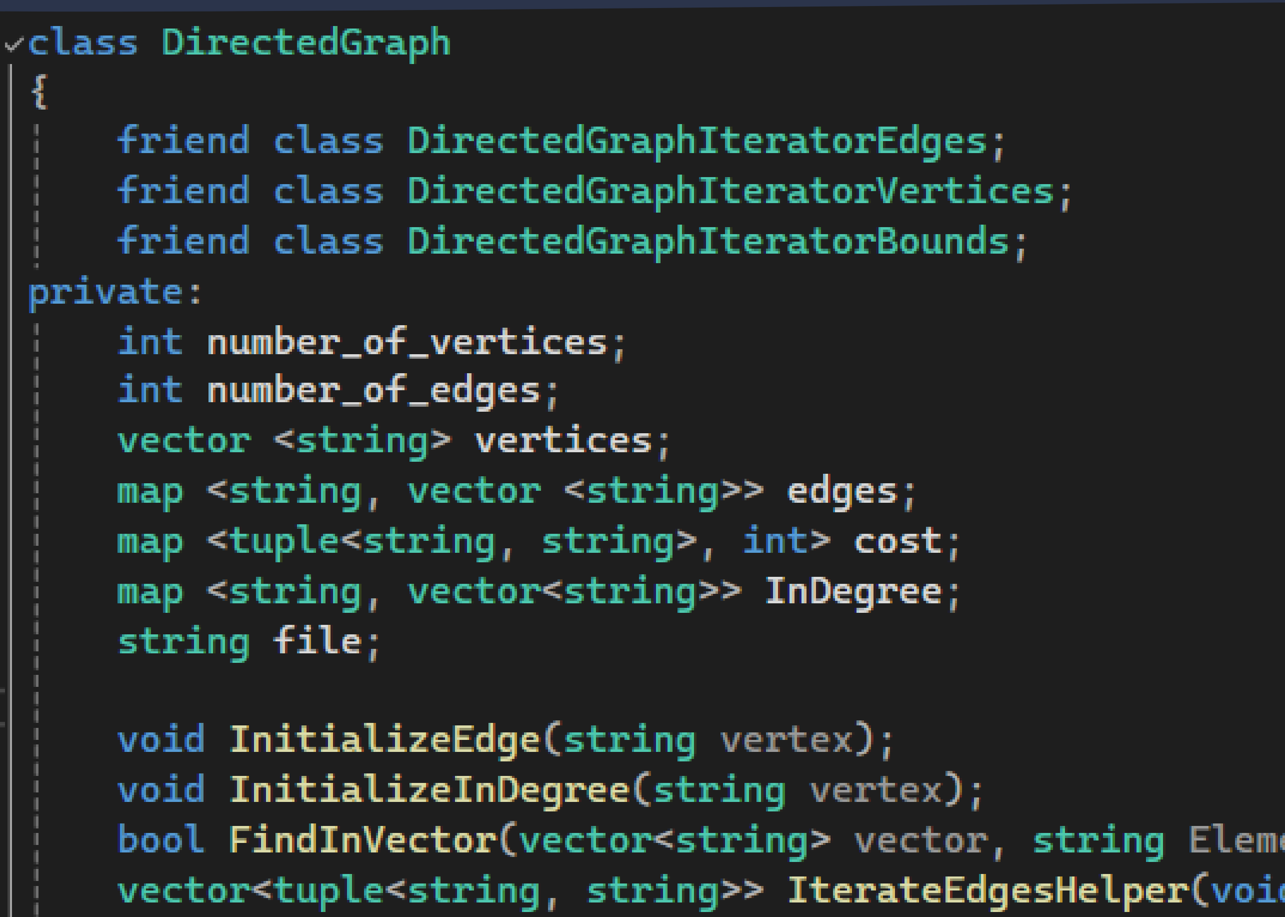
Public Functions 3/3

- ChangeCost(string start, string end, int new_cost) -> void
- GetNumberOfVertices(void) -> int
- GetNumberOfEdges(void) -> int
- PlaceCost(string start, string end, int cost) -> void
- IterateOutbounds(string vertex) -> DirectedGraphIteratorBounds
- RandomGraph(int vertices, int edges) -> void

C++ Header Preview

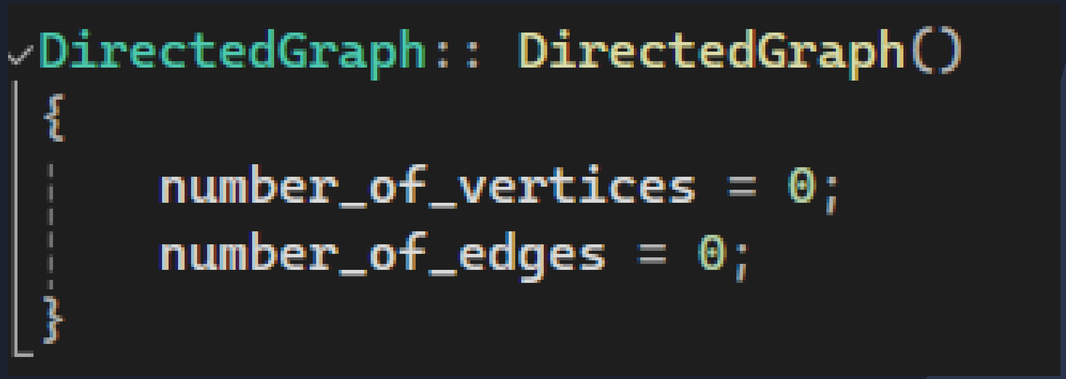


```cpp
public:
    DirectedGraph(void);
    DirectedGraph(DirectedGraph& other);
    void AddVertex(string vertex);
    void AddEdge(string start, string end);
    void DeleteVertex(string vertex);
    void DeleteEdge(string start, string end);
    void SaveFile(string file_name = "SavedGraph.txt");
    DirectedGraphIteratorVertices IterateVertices(void);
    bool CheckEdge(string starting_vertex, string ending_vertex);
    int DegreeIn(string vertex);
    int DegreeOut(string vertex);
    vector<string> GetEndPoints(string vertex);
    int GetCost(string start, string end);
    void ChangeCost(string start, string end, int new_cost);
    int GetNumberOfVertices(void);
    int GetNumberOfEdges(void);
    void PlaceCost(string start, string end, int cost);
    void RandomGraph(int vertices, int edges);
    DirectedGraphIteratorBounds IterateInbounds(string vertex);
    DirectedGraphIteratorBounds IterateOutbounds(string vertex);
    void SetFileName(string name);
    void ReadFile(void);
};
```



```cpp
class DirectedGraph
{
    friend class DirectedGraphIteratorEdges;
    friend class DirectedGraphIteratorVertices;
    friend class DirectedGraphIteratorBounds;
private:
    int number_of_vertices;
    int number_of_edges;
    vector <string> vertices;
    map <string, vector <string>> edges;
    map <tuple<string, string>, int> cost;
    map <string, vector<string>> InDegree;
    string file;

    void InitializeEdge(string vertex);
    void InitializeInDegree(string vertex);
    bool FindInVector(vector<string> vector, string Eleme
    vector<tuple<string, string>> IterateEdgesHelper(voi
```

Constructor

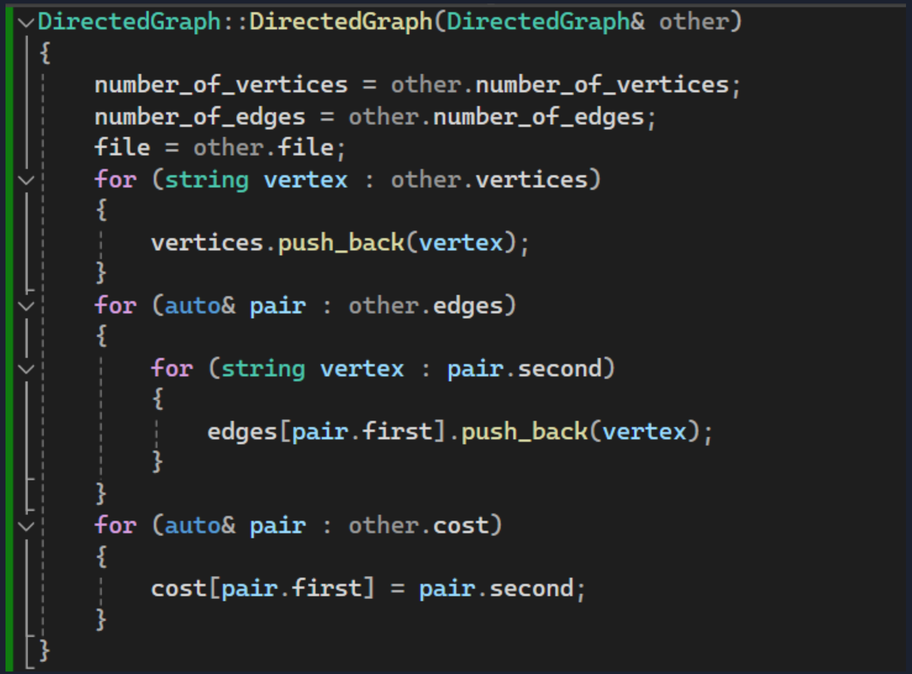


```
DirectedGraph:: DirectedGraph()
{
    number_of_vertices = 0;
    number_of_edges = 0;
}
```

- The components get initialized
- By default the name of the file is "graph1k.txt"

Copy Constructor

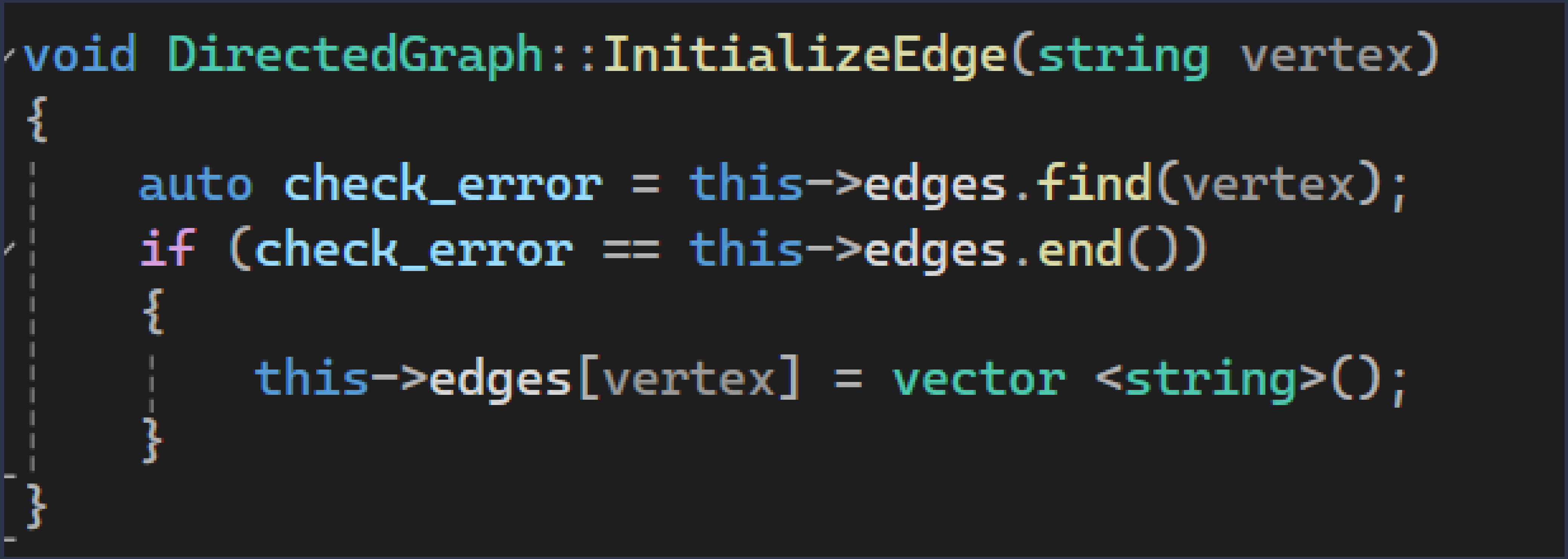
- Providing a Copy Constructor method
- To avoid assigning by reference, we go through each element of the vectors / maps to make copies of them.



```
DirectedGraph::DirectedGraph(DirectedGraph& other)
{
    number_of_vertices = other.number_of_vertices;
    number_of_edges = other.number_of_edges;
    file = other.file;
    for (string vertex : other.vertices)
    {
        vertices.push_back(vertex);
    }
    for (auto& pair : other.edges)
    {
        for (string vertex : pair.second)
        {
            edges[pair.first].push_back(vertex);
        }
    }
    for (auto& pair : other.cost)
    {
        cost[pair.first] = pair.second;
    }
}
```

InitializeEdge

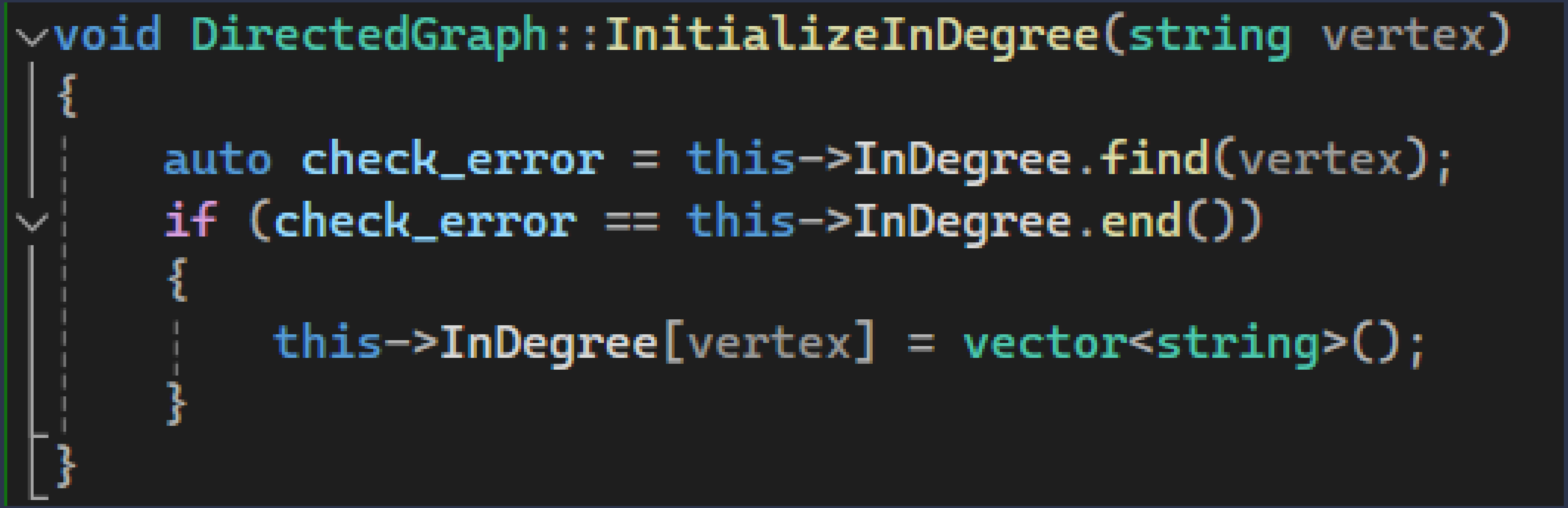
- Used as a helper function
- Initializes a vector for the ending point s of an edge
- Does nothing if the vector has already been initialized



```
void DirectedGraph::InitializeEdge(string vertex)
{
    auto check_error = this->edges.find(vertex);
    if (check_error == this->edges.end())
    {
        this->edges[vertex] = vector <string>();
    }
}
```

InitializeInDegree

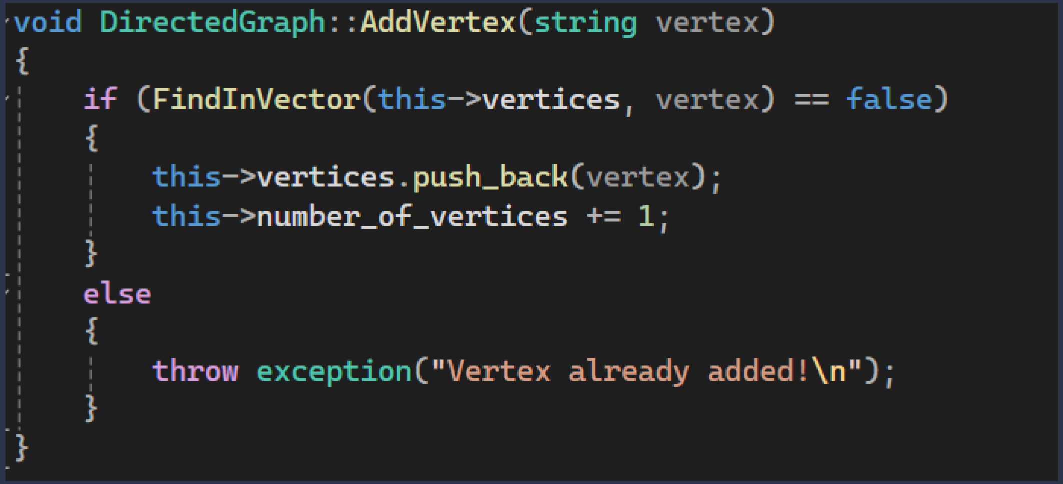
- Used as a helper function
- Implemented to achieve O(1) complexity in a Getter function
- Initializes a vector in the map value



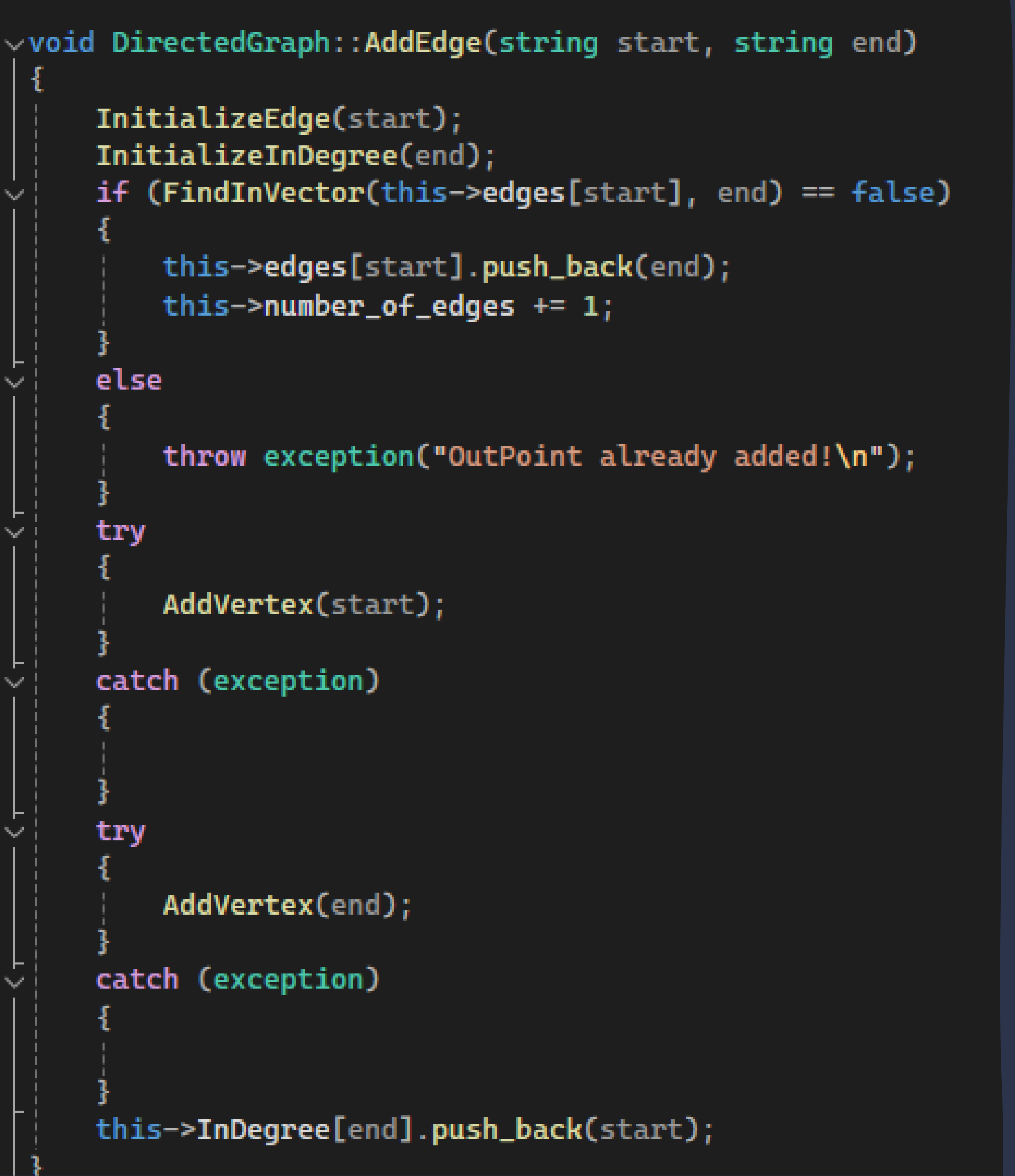
```
void DirectedGraph::InitializeInDegree(string vertex)
{
    auto check_error = this->InDegree.find(vertex);
    if (check_error == this->InDegree.end())
    {
        this->InDegree[vertex] = vector<string>();
    }
}
```

AddVertex

- Adds a vertex in the graph if it's not already added, otherwise throws exception
- Increments the number_of_vertices component



```
void DirectedGraph::AddVertex(string vertex)
{
    if (FindInVector(this->vertices, vertex) == false)
    {
        this->vertices.push_back(vertex);
        this->number_of_vertices += 1;
    }
    else
    {
        throw exception("Vertex already added!\n");
    }
}
```



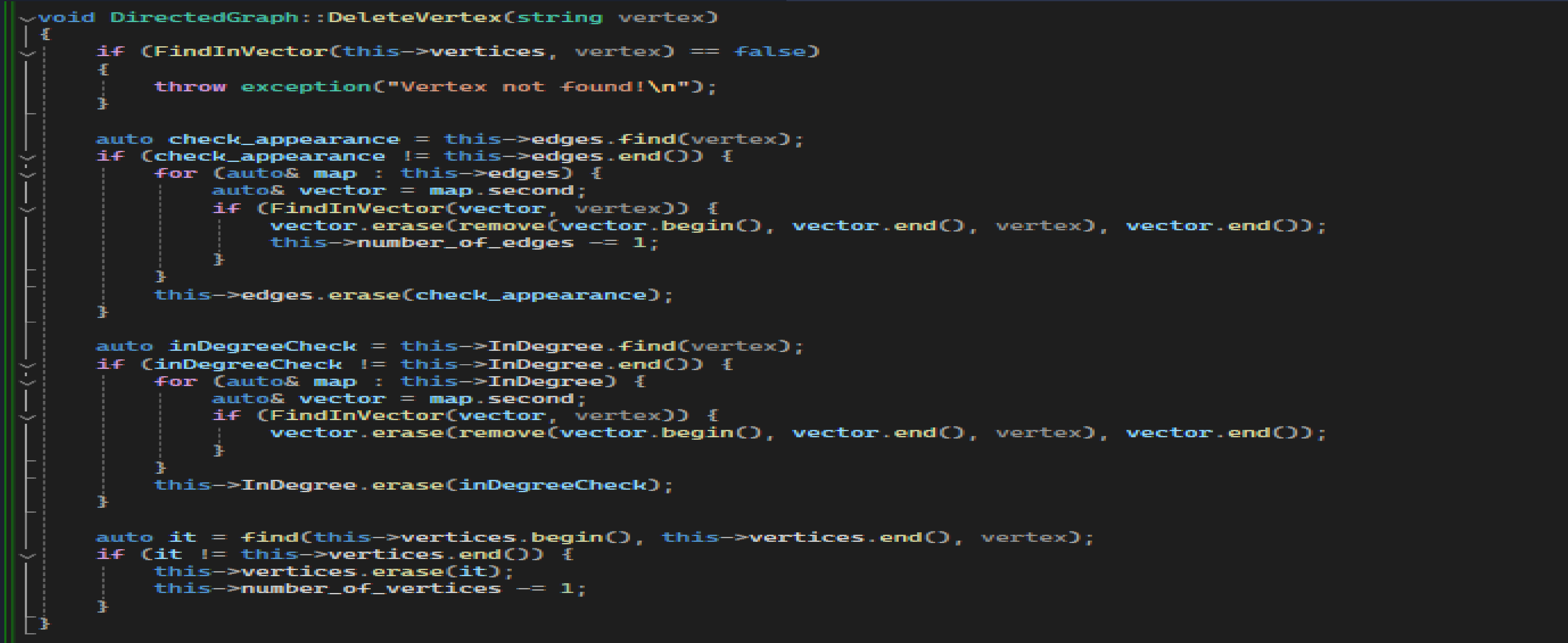
```
void DirectedGraph::AddEdge(string start, string end)
{
    InitializeEdge(start);
    InitializeInDegree(end);
    if (FindInVector(this->edges[start], end) == false)
    {
        this->edges[start].push_back(end);
        this->number_of_edges += 1;
    }
    else
    {
        throw exception("OutPoint already added!\n");
    }
    try
    {
        AddVertex(start);
    }
    catch (exception)
    {

    }
    try
    {
        AddVertex(end);
    }
    catch (exception)
    {

    }
    this->InDegree[end].push_back(start);
}
```

AddEdge

- Initializes a space for edges / InDegree's
- Adds an edge if not already added, otherwise throws exception
- Adds new vertices, catches exception if found

DeleteVertex

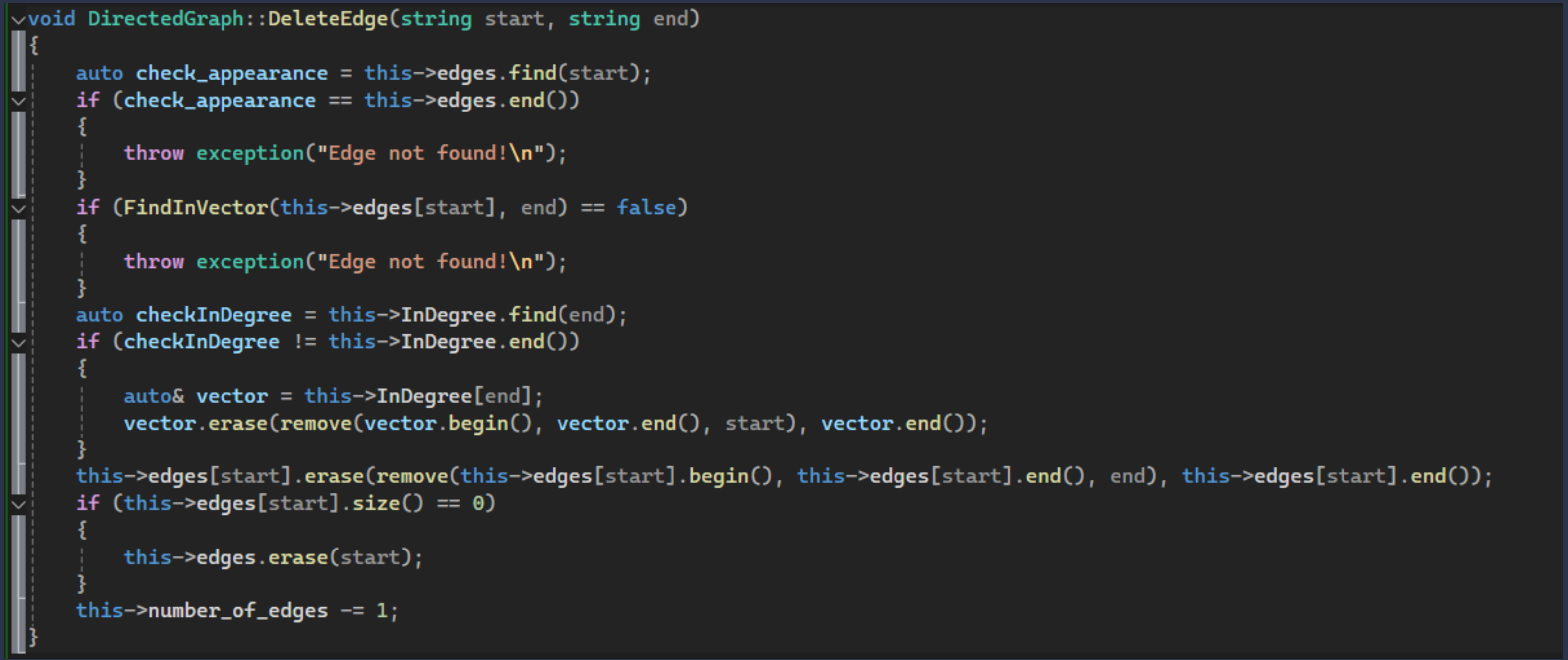
- Deletes the given vertex, the inDegrees and all it's edges if found, otherwise throws exception



```
void DirectedGraph::DeleteVertex(string vertex)
{
    if (FindInVector(this->vertices, vertex) == false)
    {
        throw exception("Vertex not found!\n");
    }

    auto check_appearance = this->edges.find(vertex);
    if (check_appearance != this->edges.end()) {
        for (auto& map : this->edges) {
            auto& vector = map.second;
            if (FindInVector(vector, vertex)) {
                vector.erase(remove(vector.begin(), vector.end(), vertex), vector.end());
                this->number_of_edges -= 1;
            }
        }
        this->edges.erase(check_appearance);
    }

    auto inDegreeCheck = this->InDegree.find(vertex);
    if (inDegreeCheck != this->InDegree.end()) {
        for (auto& map : this->InDegree) {
            auto& vector = map.second;
            if (FindInVector(vector, vertex)) {
                vector.erase(remove(vector.begin(), vector.end(), vertex), vector.end());
            }
        }
        this->InDegree.erase(inDegreeCheck);
    }

    auto it = find(this->vertices.begin(), this->vertices.end(), vertex);
    if (it != this->vertices.end()) {
        this->vertices.erase(it);
        this->number_of_vertices -= 1;
    }
}
```

DeleteEdge

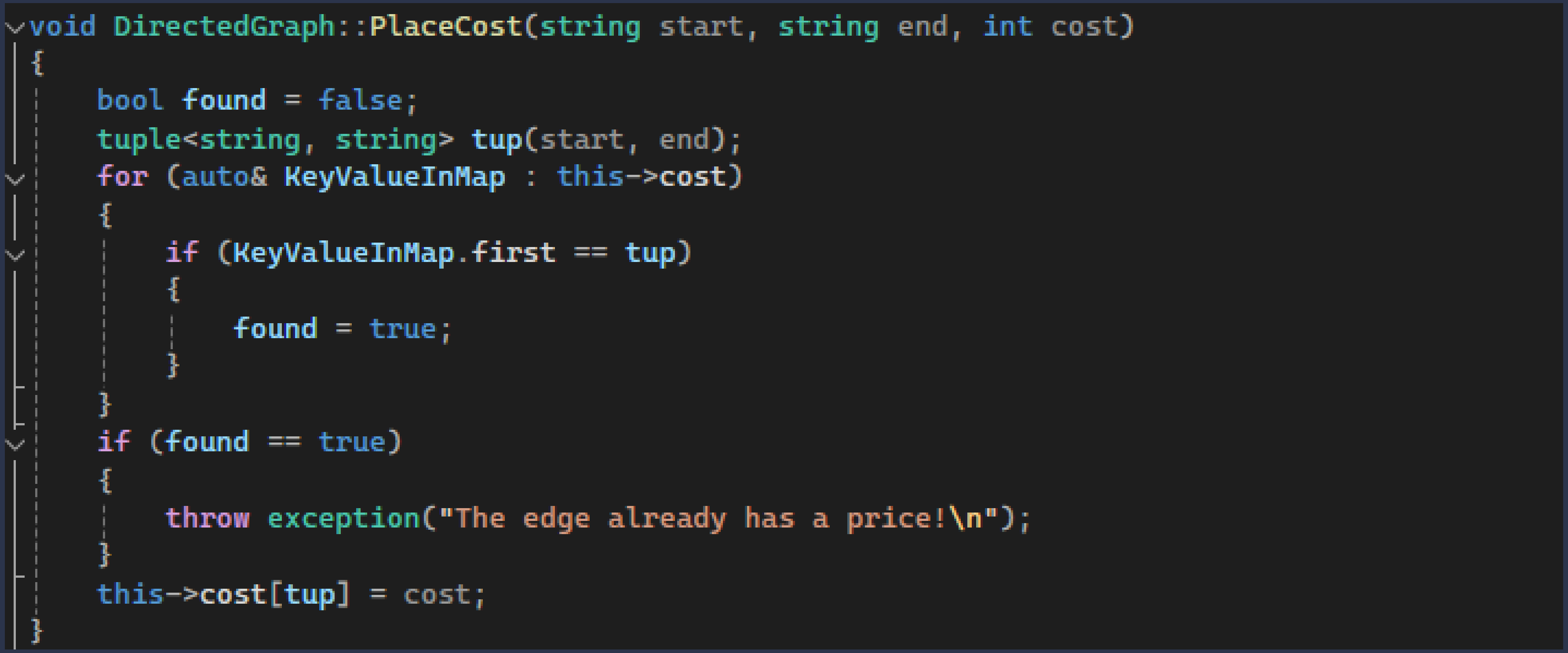
Deletes the edge if found, otherwise throws exception



```
void DirectedGraph::DeleteEdge(string start, string end)
{
    auto check_appearance = this->edges.find(start);
    if (check_appearance == this->edges.end())
    {
        throw exception("Edge not found!\n");
    }
    if (FindInVector(this->edges[start], end) == false)
    {
        throw exception("Edge not found!\n");
    }
    auto checkInDegree = this->InDegree.find(end);
    if (checkInDegree != this->InDegree.end())
    {
        auto& vector = this->InDegree[end];
        vector.erase(remove(vector.begin(), vector.end(), start), vector.end());
    }
    this->edges[start].erase(remove(this->edges[start].begin(), this->edges[start].end(), end), this->edges[start].end());
    if (this->edges[start].size() == 0)
    {
        this->edges.erase(start);
    }
    this->number_of_edges -= 1;
}
```

PlaceCost

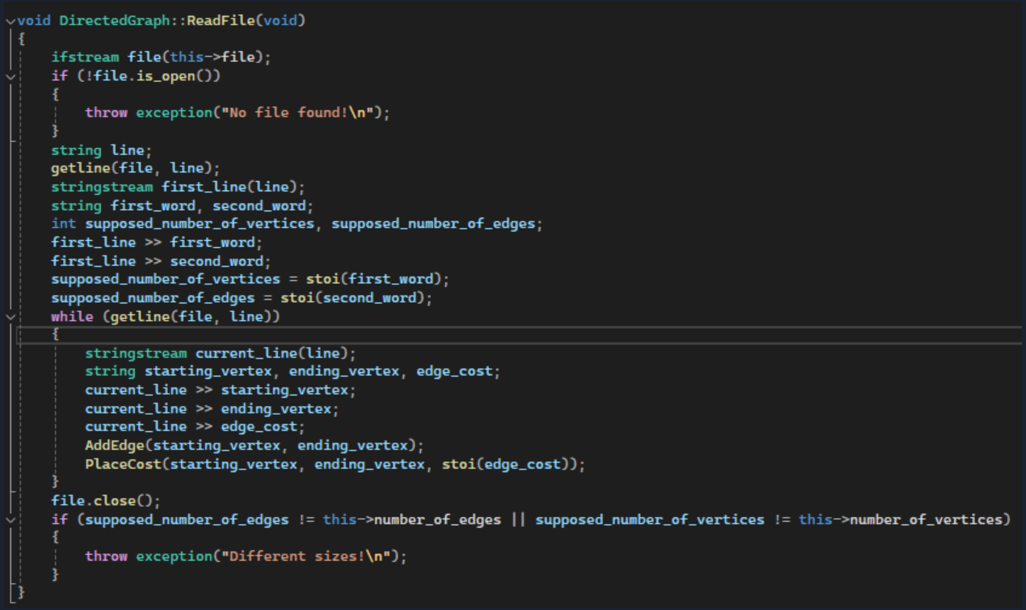
- Places a "Cost" value on an edge
- If edge doesn't exists, nothing happens
- If edge already has a cost, throws exception



```cpp
void DirectedGraph::PlaceCost(string start, string end, int cost)
{
    bool found = false;
    tuple<string, string> tup(start, end);
    for (auto& KeyValueInMap : this->cost)
    {
        if (KeyValueInMap.first == tup)
        {
            found = true;
        }
    }
    if (found == true)
    {
        throw exception("The edge already has a price!\n");
    }
    this->cost[tup] = cost;
}
```

ReadFile

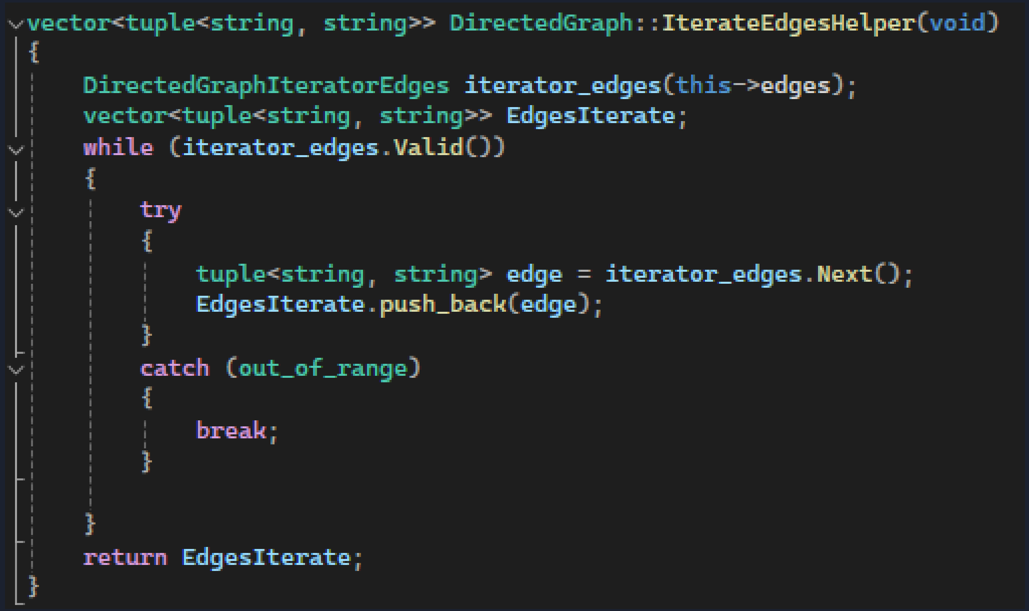
- Reads the desired number of vertices and edges
- Reads the file line by line and adds edges / costs
- If the file is not found, throws exception
- If the final number of vertices / edges is not the same one as the one read from the file, throws exception



```cpp
void DirectedGraph::ReadFile(void)
{
    ifstream file(this->file);
    if (!file.is_open())
    {
        throw exception("No file found!\n");
    }
    string line;
    getline(file, line);
    stringstream first_line(line);
    string first_word, second_word;
    int supposed_number_of_vertices, supposed_number_of_edges;
    first_line >> first_word;
    first_line >> second_word;
    supposed_number_of_vertices = stoi(first_word);
    supposed_number_of_edges = stoi(second_word);
    while (getline(file, line))
    {
        stringstream current_line(line);
        string starting_vertex, ending_vertex, edge_cost;
        current_line >> starting_vertex;
        current_line >> ending_vertex;
        current_line >> edge_cost;
        AddEdge(starting_vertex, ending_vertex);
        PlaceCost(starting_vertex, ending_vertex, stoi(edge_cost));
    }
    file.close();
    if (supposed_number_of_edges != this->number_of_edges || supposed_number_of_vertices != this->number_of_vertices)
    {
        throw exception("Different sizes!\n");
    }
}
```

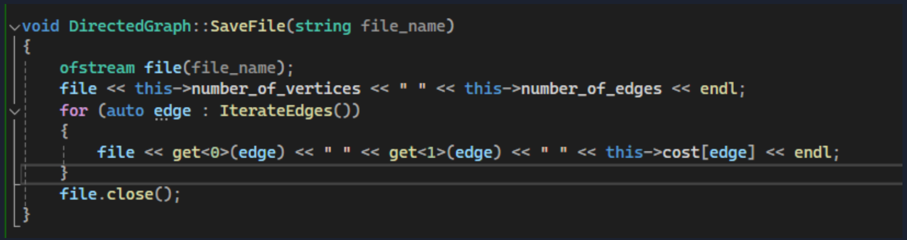
IterateEdgesHelper

- Uses the Iterator Class and it's function to get every pair of edges and returns it



```
vector<tuple<string, string>> DirectedGraph::IterateEdgesHelper(void)
{
    DirectedGraphIteratorEdges iterator_edges(this->edges);
    vector<tuple<string, string>> EdgesIterate;
    while (iterator_edges.Valid())
    {
        try
        {
            tuple<string, string> edge = iterator_edges.Next();
            EdgesIterate.push_back(edge);
        }
        catch (out_of_range)
        {
            break;
        }

    }
    return EdgesIterate;
}
```

SaveFile

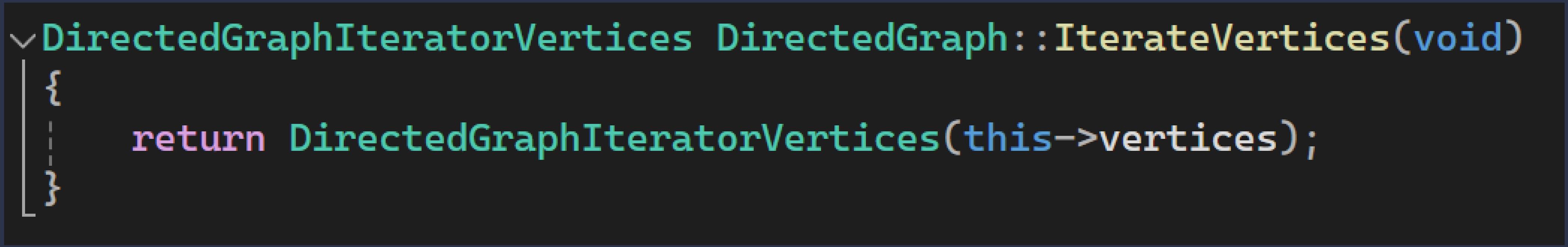


```
void DirectedGraph::SaveFile(string file_name)
{
    ofstream file(file_name);
    file << this->number_of_vertices << " " << this->number_of_edges << endl;
    for (auto edge : IterateEdges())
    {
        file << get<0>(edge) << " " << get<1>(edge) << " " << this->cost[edge] << endl;
    }
    file.close();
}
```

- Opens a file in writing mode
- Writes in the same style as an input file

IterateVertices

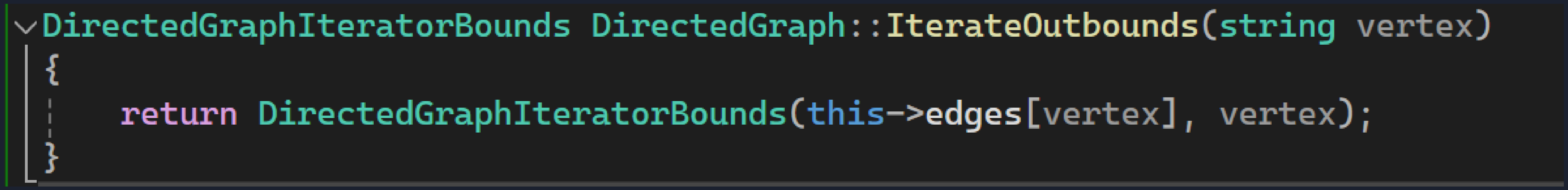
Returns an iterator for the graph vertices



```
DirectedGraphIteratorVertices DirectedGraph::IterateVertices(void)
{
    return DirectedGraphIteratorVertices(this->vertices);
}
```

IterateOutbounds

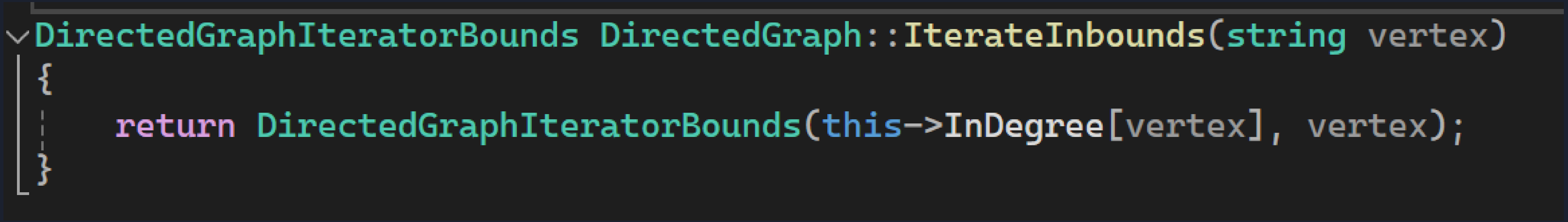
- Returns an iterator for the graphs outbounds



```
DirectedGraphIteratorBounds DirectedGraph::IterateOutbounds(string vertex)
{
    return DirectedGraphIteratorBounds(this->edges[vertex], vertex);
}
```

IterateInbounds

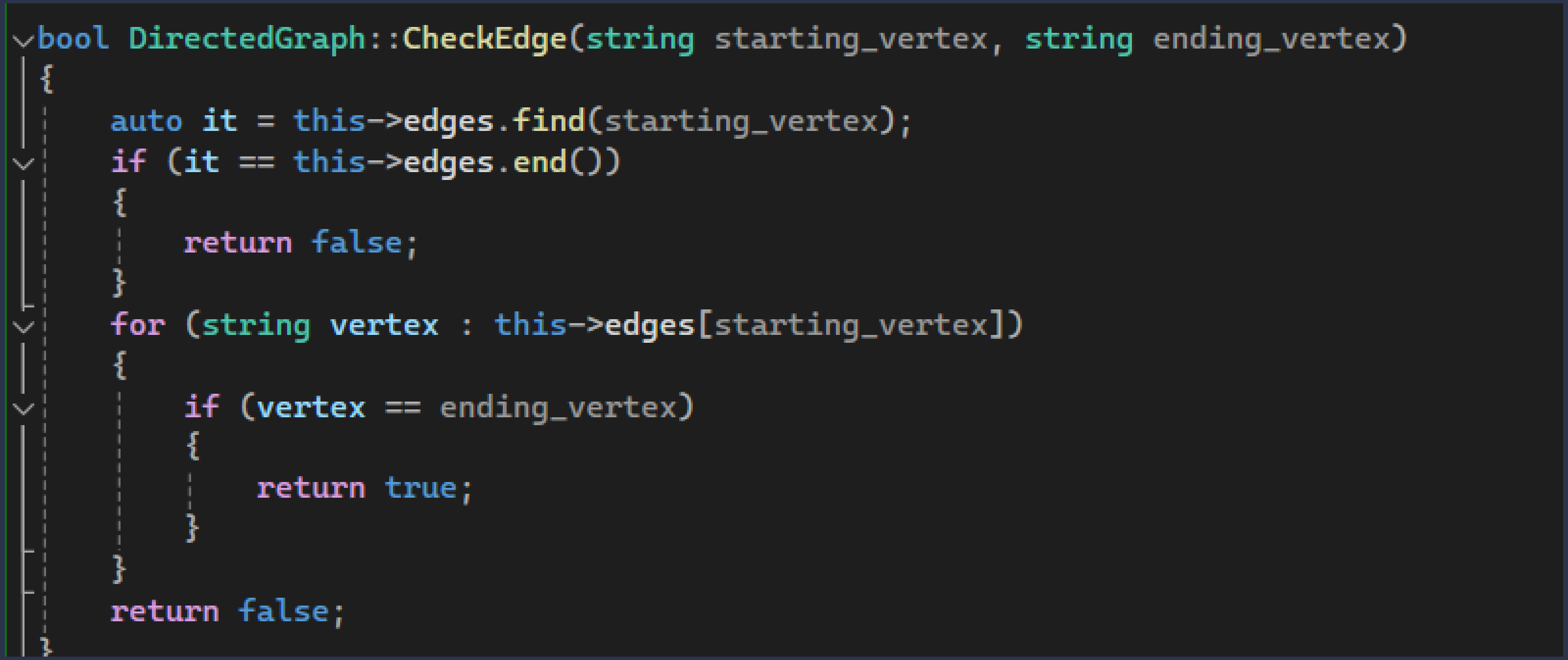
- Returns an iterator for the graphs inbounds



```
DirectedGraphIteratorBounds DirectedGraph::IterateInbounds(string vertex)
{
    return DirectedGraphIteratorBounds(this->InDegree[vertex], vertex);
}
```

CheckEdge

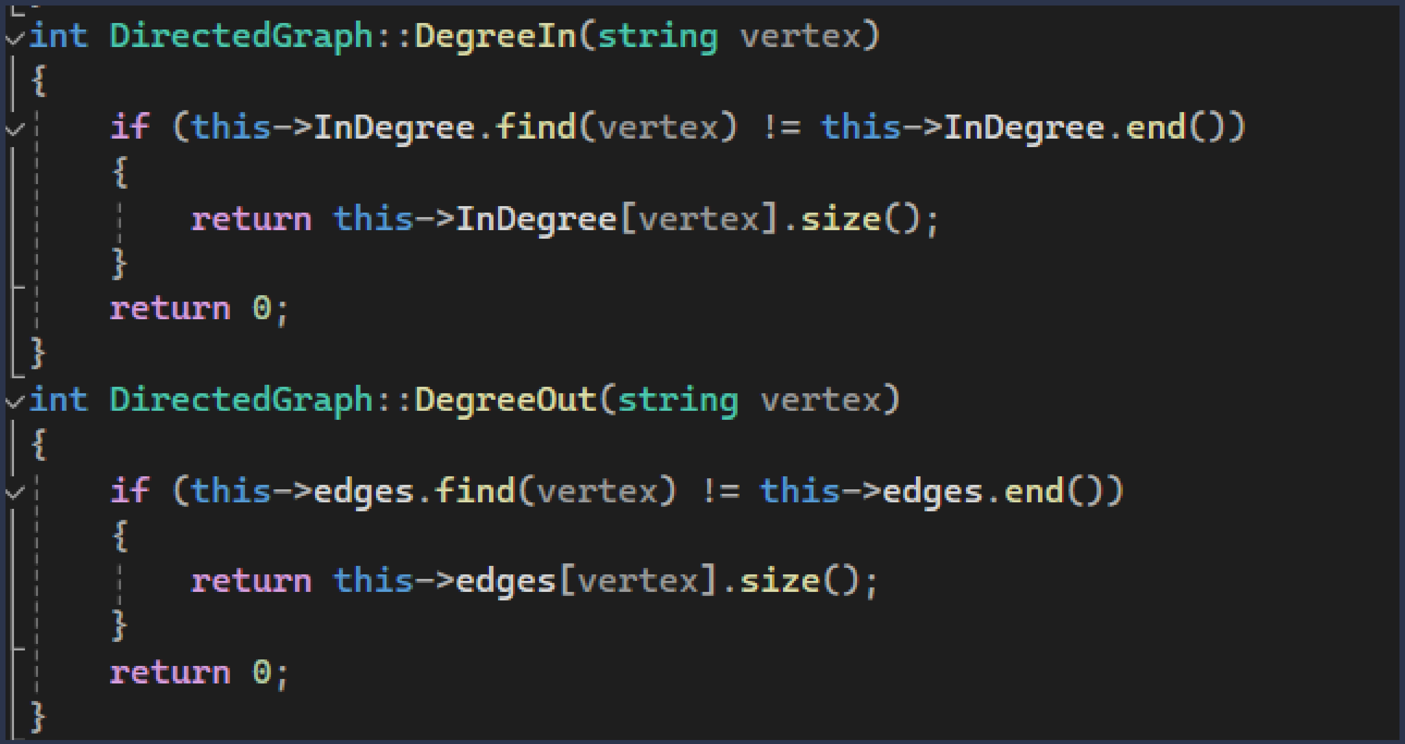
- Checks if a specified edge exists
- Returns true / false



```cpp
bool DirectedGraph::CheckEdge(string starting_vertex, string ending_vertex)
{
    auto it = this->edges.find(starting_vertex);
    if (it == this->edges.end())
    {
        return false;
    }
    for (string vertex : this->edges[starting_vertex])
    {
        if (vertex == ending_vertex)
        {
            return true;
        }
    }
    return false;
}
```

DegreeIn/DegreeOut

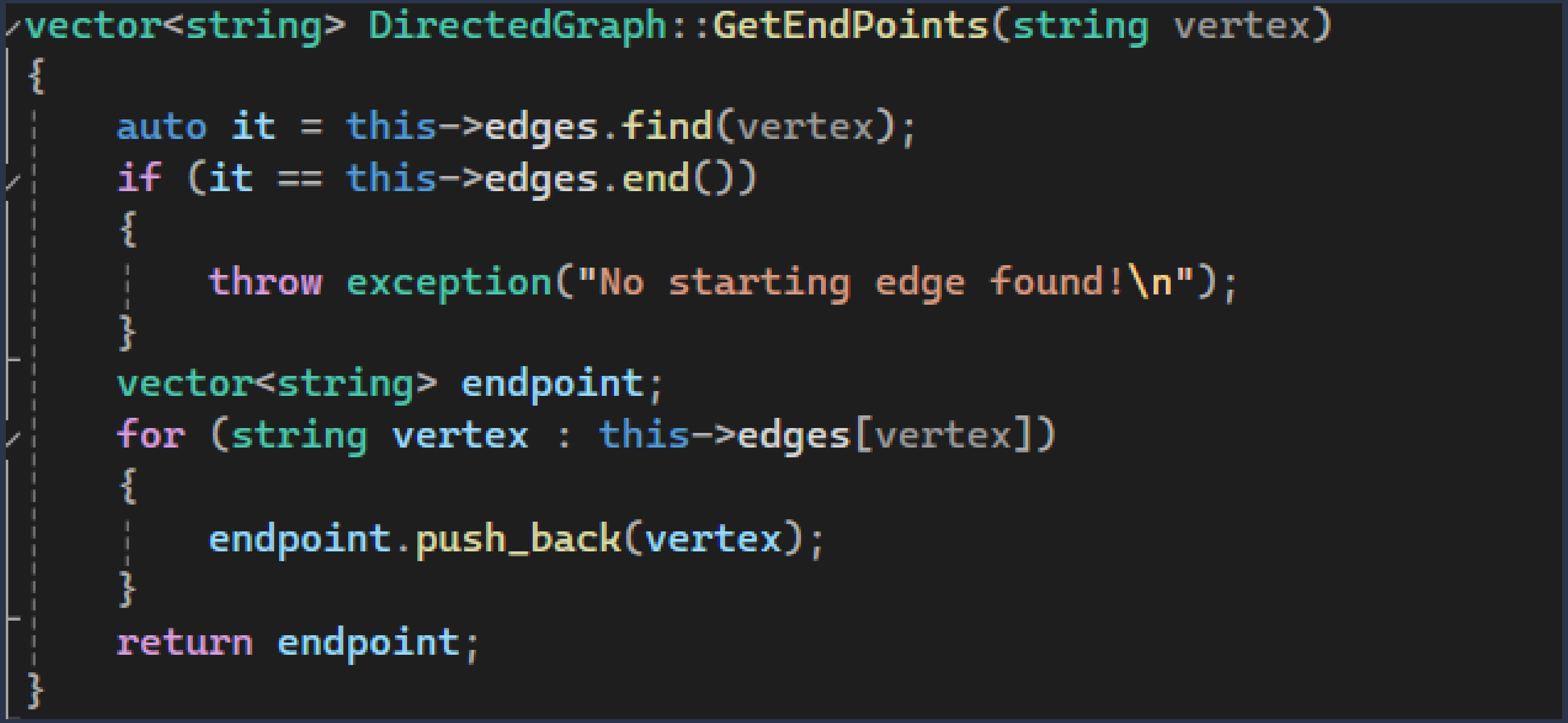
- Returns the Degree In/Out of a specified vertex



```
int DirectedGraph::DegreeIn(string vertex)
{
    if (this->InDegree.find(vertex) != this->InDegree.end())
    {
        return this->InDegree[vertex].size();
    }
    return 0;
}
int DirectedGraph::DegreeOut(string vertex)
{
    if (this->edges.find(vertex) != this->edges.end())
    {
        return this->edges[vertex].size();
    }
    return 0;
}
```

GetEndPoints

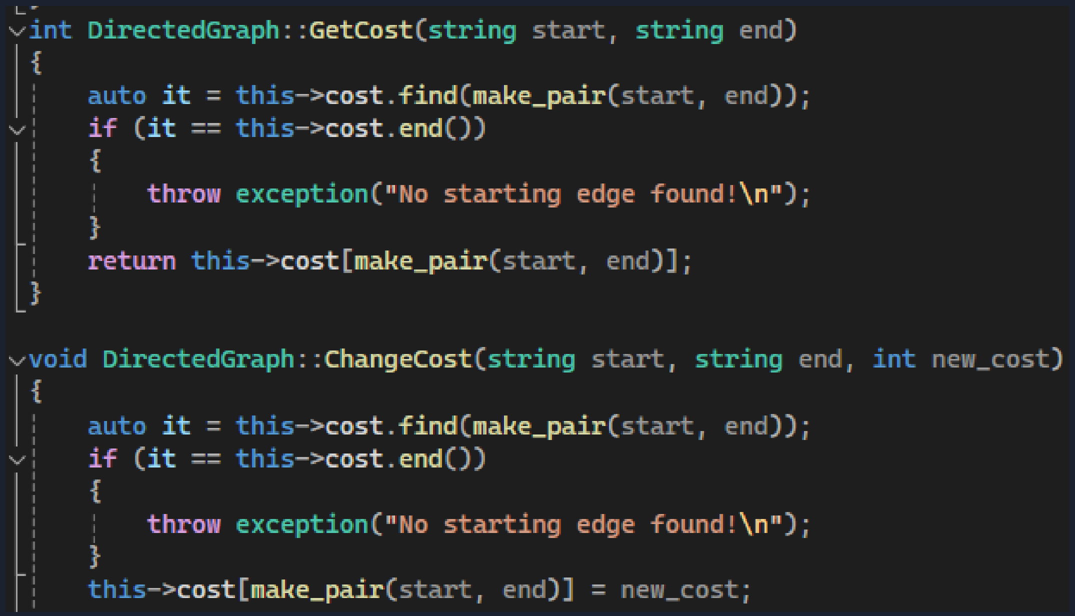
- Returns the endpoints of a vertex if found, otherwise throws exception



```
vector<string> DirectedGraph::GetEndPoints(string vertex)
{
    auto it = this->edges.find(vertex);
    if (it == this->edges.end())
    {
        throw exception("No starting edge found!\n");
    }
    vector<string> endpoint;
    for (string vertex : this->edges[vertex])
    {
        endpoint.push_back(vertex);
    }
    return endpoint;
}
```

Get/Change Cost

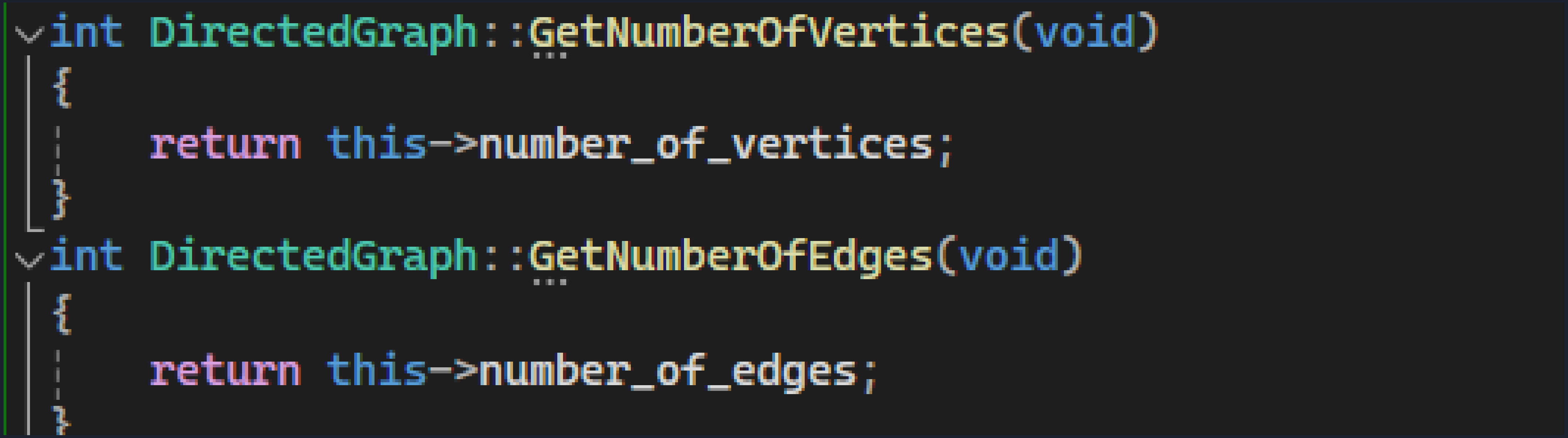
- Returns the cost of an edge / changes the price given
- Throws exception if no edge found



```
int DirectedGraph::GetCost(string start, string end)
{
    auto it = this->cost.find(make_pair(start, end));
    if (it == this->cost.end())
    {
        throw exception("No starting edge found!\n");
    }
    return this->cost[make_pair(start, end)];
}

void DirectedGraph::ChangeCost(string start, string end, int new_cost)
{
    auto it = this->cost.find(make_pair(start, end));
    if (it == this->cost.end())
    {
        throw exception("No starting edge found!\n");
    }
    this->cost[make_pair(start, end)] = new_cost;
```

GetNumberOf Vertices/Edges

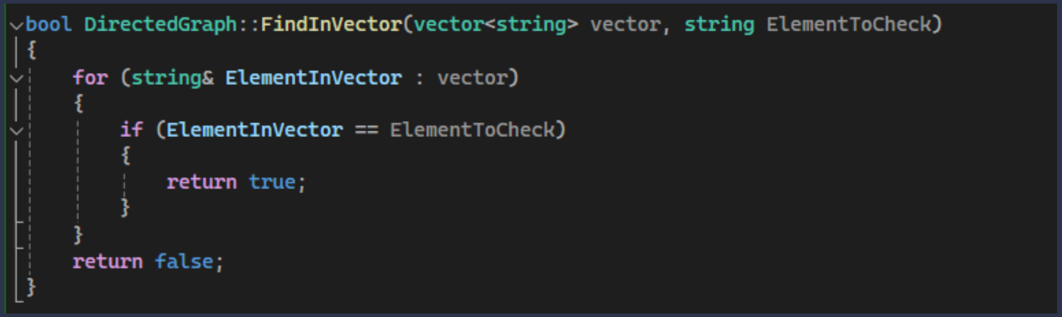
- Returns the number of vertices / edges



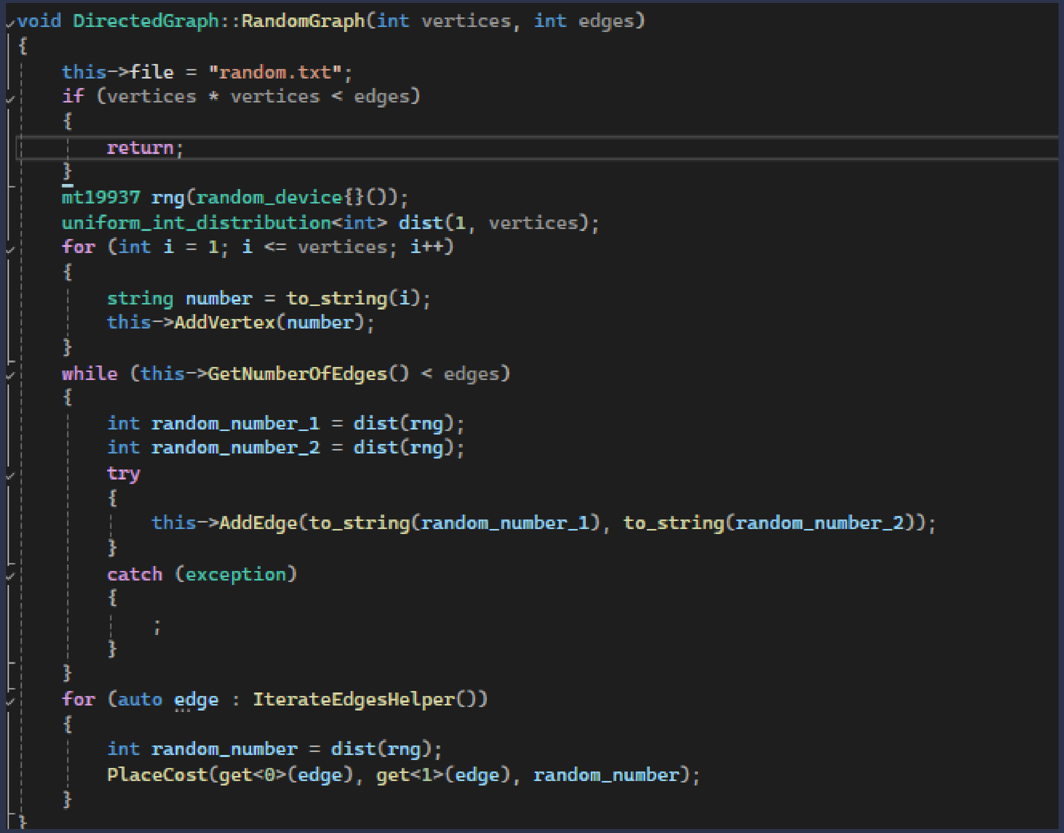
```
int DirectedGraph::GetNumberOfVertices(void)
{
    return this->number_of_vertices;
}
int DirectedGraph::GetNumberOfEdges(void)
{
    return this->number_of_edges;
}
```

FindInVector

- Helper function
- Returns true if the element is found, otherwise false



```
bool DirectedGraph::FindInVector(vector<string> vector, string ElementToCheck)
{
    for (string& ElementInVector : vector)
    {
        if (ElementInVector == ElementToCheck)
        {
            return true;
        }
    }
    return false;
}
```



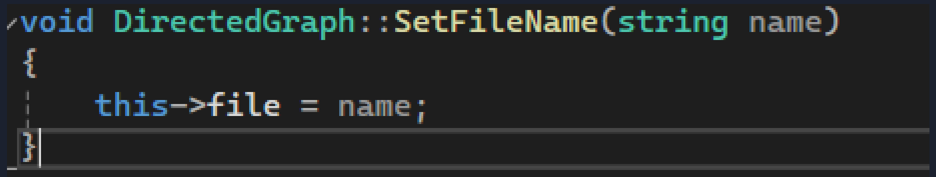
```
void DirectedGraph::RandomGraph(int vertices, int edges)
{
    this->file = "random.txt";
    if (vertices * vertices < edges)
    {
        return;
    }
    mt19937 rng(random_device{}());
    uniform_int_distribution<int> dist(1, vertices);
    for (int i = 1; i <= vertices; i++)
    {
        string number = to_string(i);
        this->AddVertex(number);
    }
    while (this->GetNumberOfEdges() < edges)
    {
        int random_number_1 = dist(rng);
        int random_number_2 = dist(rng);
        try
        {
            this->AddEdge(to_string(random_number_1), to_string(random_number_2));
        }
        catch (exception)
        {
            ;
        }
    }
    for (auto edge : IterateEdgesHelper())
    {
        int random_number = dist(rng);
        PlaceCost(get<0>(edge), get<1>(edge), random_number);
    }
}
```

RandomGraph

Generates a completely random graph using pre-built random functions or an empty one if it's not possible

SetFileName

Setter - changes the graph file name



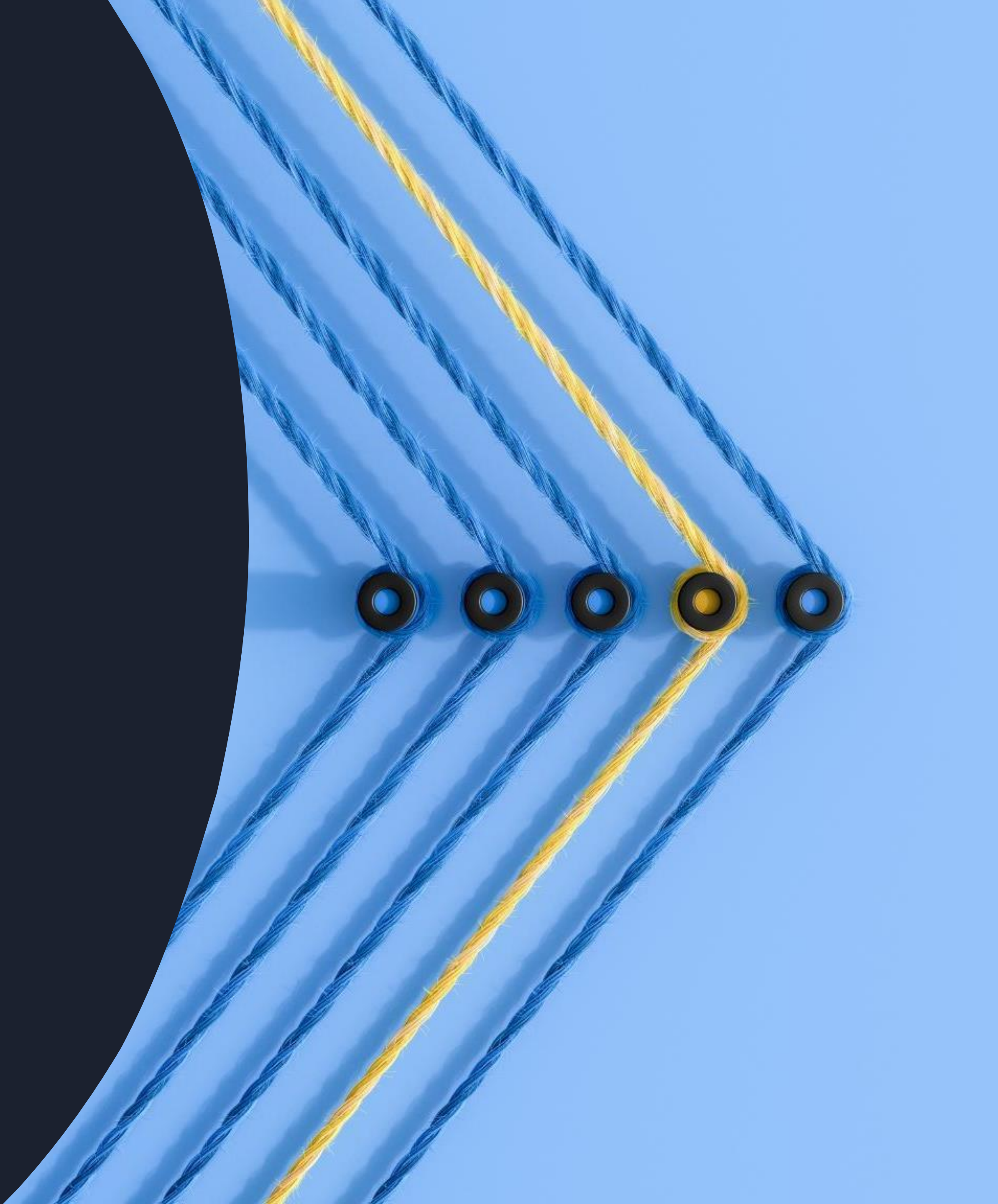
```
void DirectedGraph::SetFileName(string name)
{
    this->file = name;
}
```

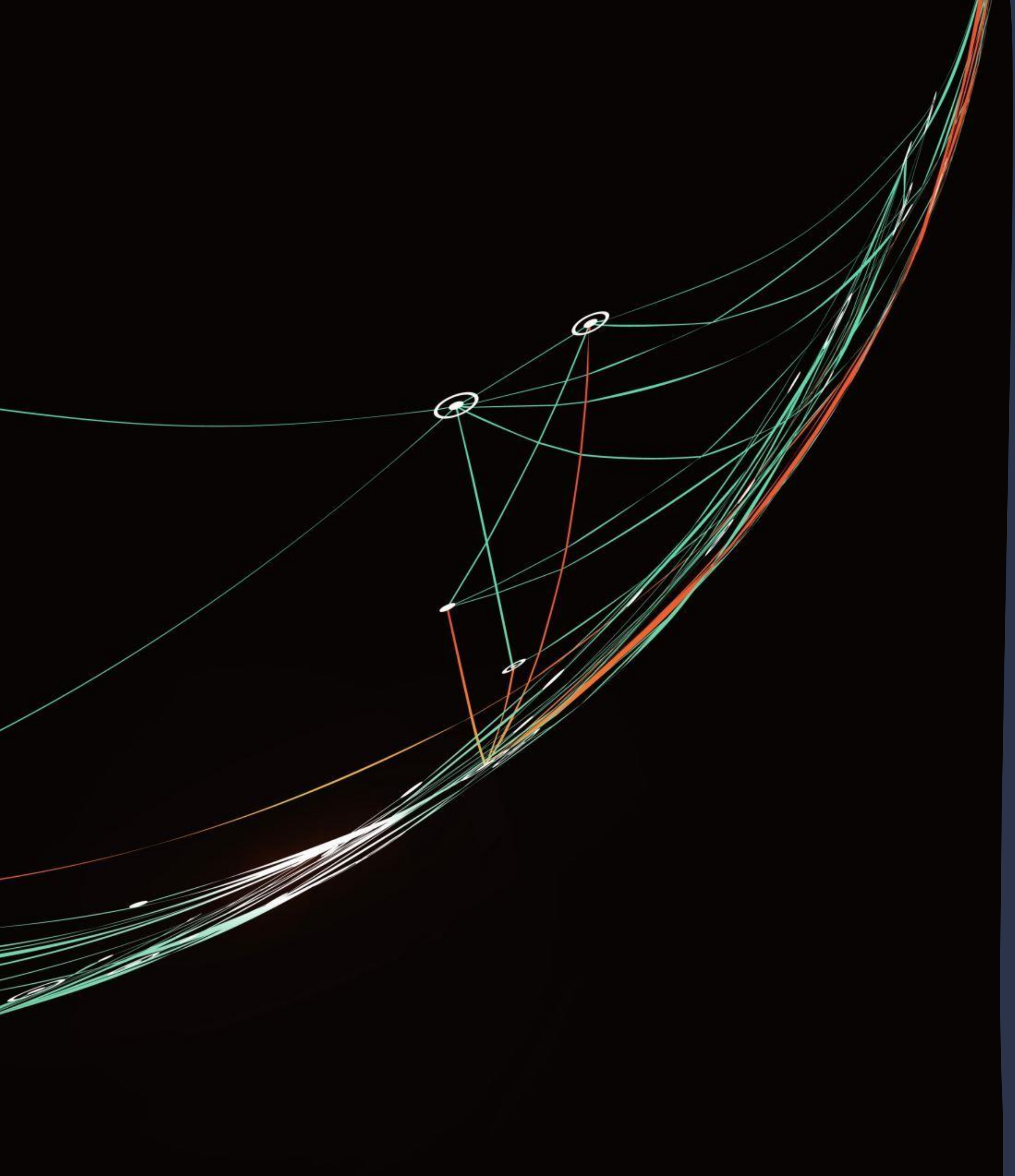
C++ Implementation of Class DirectedGraphIteratorEdges

Friend Class: DirectedGraph

Private Components

- Edges map (<string, vector<string>>)
- KeyIterator (map<string, vector<string>>)
- CurrentFr (int)
- CurrentKey (string)



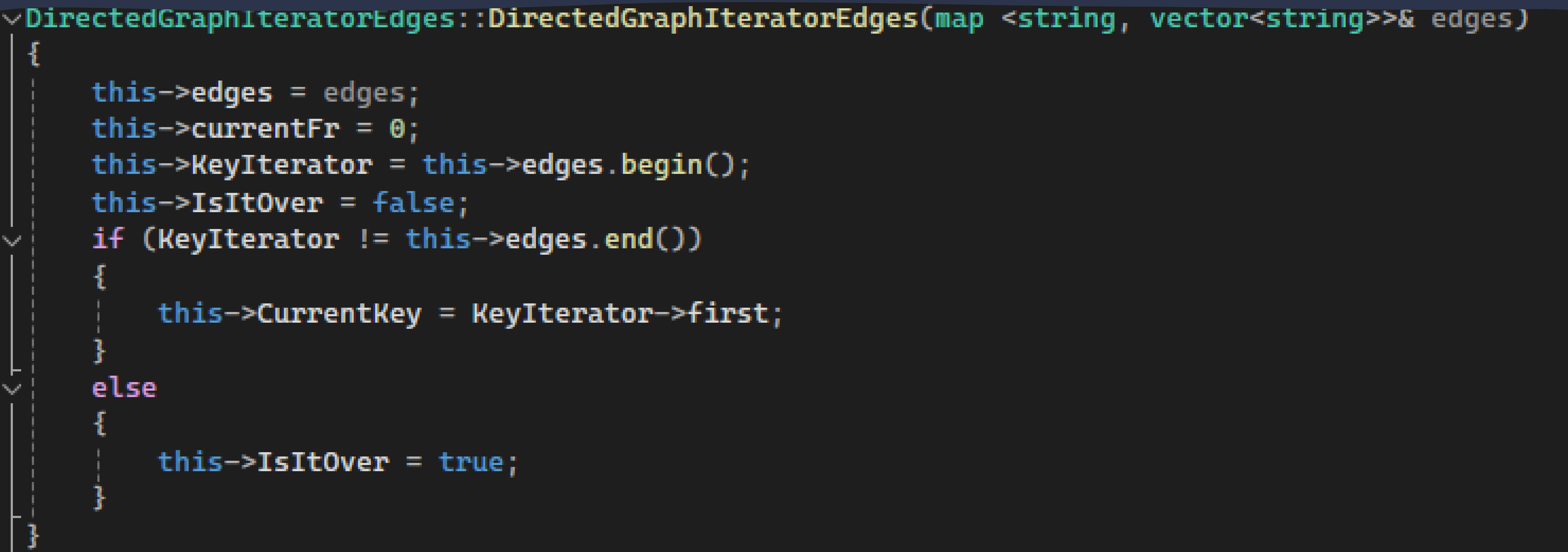


Private Functions

- DirectedGraphIteratorEdges(map <string, vector<string>>& edges) -> Constructor
- Valid(void) -> bool
- Next(void) -> tuple<string, string>

Constructor

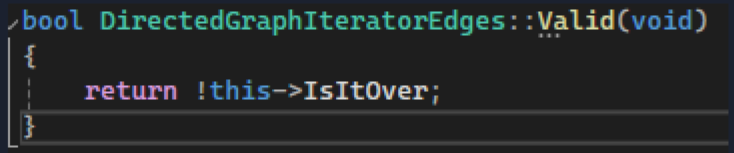
- Initializes the components
- Checks if there are any elements



```cpp
DirectedGraphIteratorEdges::DirectedGraphIteratorEdges(map <string, vector<string>>& edges)
{
    this->edges = edges;
    this->currentFr = 0;
    this->KeyIterator = this->edges.begin();
    this->IsItOver = false;
    if (KeyIterator != this->edges.end())
    {
        this->CurrentKey = KeyIterator->first;
    }
    else
    {
        this->IsItOver = true;
    }
}
```

Valid

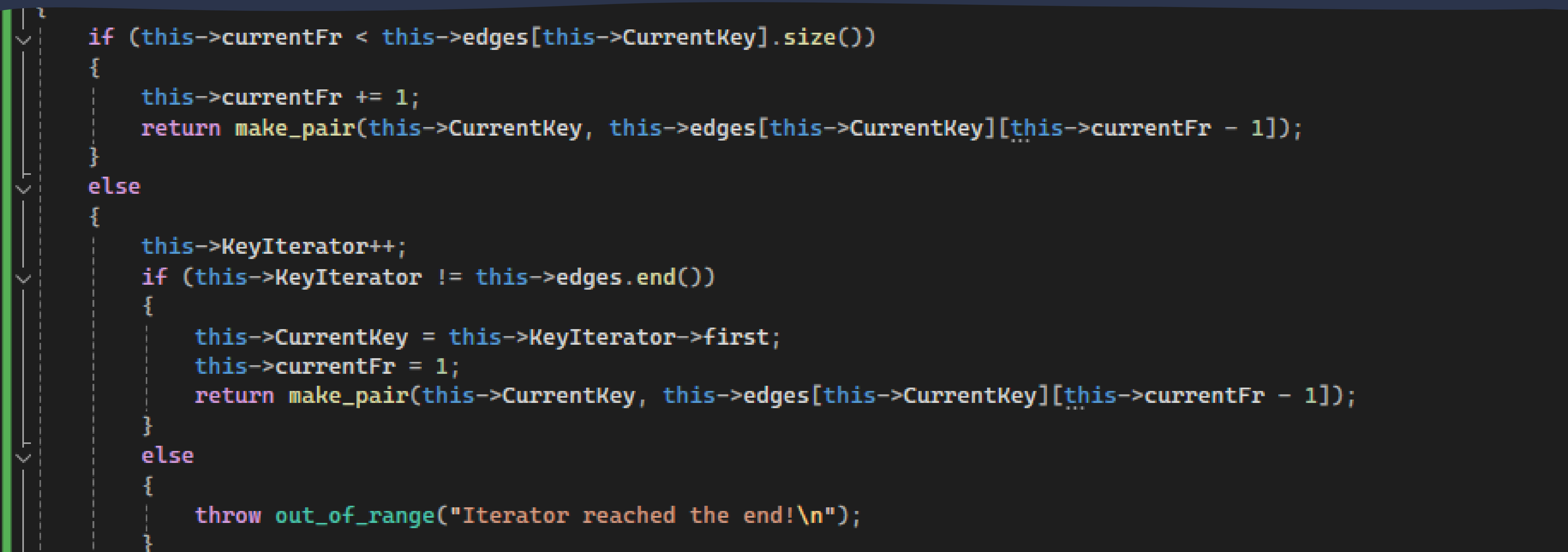
Checks if the end has been reached



```
bool DirectedGraphIteratorEdges::Valid(void)
{
    return !this->IsItOver;
}
```

Next

- Returns the current edge
- Throws exception if the end has been reached



```
if (this->currentFr < this->edges[this->CurrentKey].size())
{
    this->currentFr += 1;
    return make_pair(this->CurrentKey, this->edges[this->CurrentKey][this->currentFr - 1]);
}
else
{
    this->KeyIterator++;
    if (this->KeyIterator != this->edges.end())
    {
        this->CurrentKey = this->KeyIterator->first;
        this->currentFr = 1;
        return make_pair(this->CurrentKey, this->edges[this->CurrentKey][this->currentFr - 1]);
    }
    else
    {
        throw out_of_range("Iterator reached the end!\n");
    }
```

C++ Implementation of Class DirectedGraphIteratorBounds

- Friend Class: DirectedGraph

Private Components

- start (string)
- bounds (vector<string>)
- KeyIterator (vector<string>::iterator)
- IsItOver (bool)
- CurrentValue (string)

Private Functions

- DirectedGraphIteratorBounds(vector<string> vertices, string start) -> Constructor

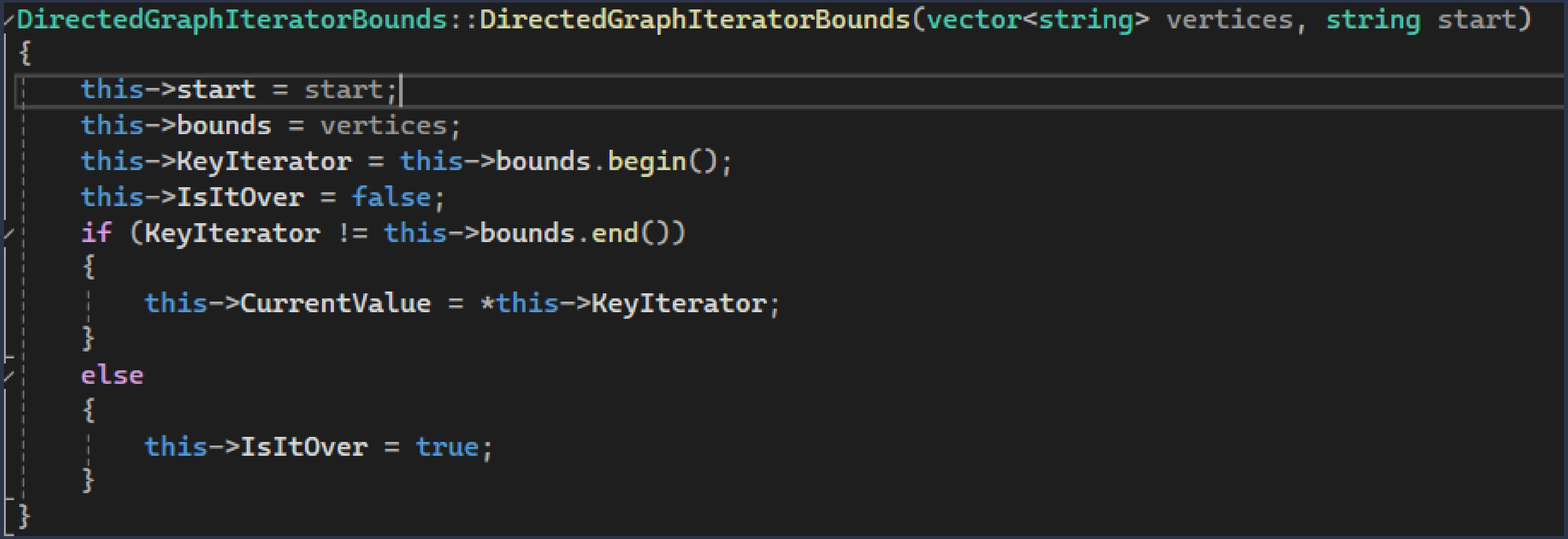
Public Functions

VALID(VOID) -> BOOL

NEXT(VOID) -> TUPLE <STRING, STRING>

DirectedGraphIteratorBounds

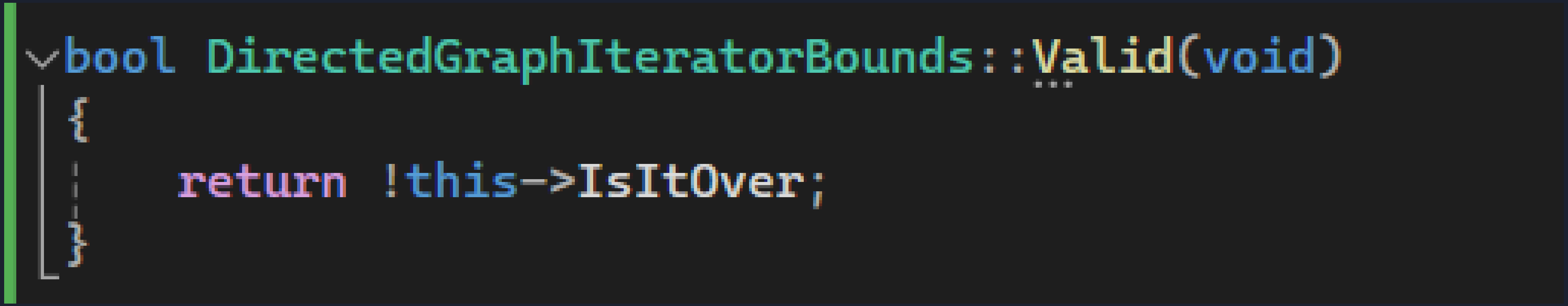
- Initializes the components
- Checks if there are any components



```
DirectedGraphIteratorBounds::DirectedGraphIteratorBounds(vector<string> vertices, string start)
{
    this->start = start;
    this->bounds = vertices;
    this->KeyIterator = this->bounds.begin();
    this->IsItOver = false;
    if (KeyIterator != this->bounds.end())
    {
        this->CurrentValue = *this->KeyIterator;
    }
    else
    {
        this->IsItOver = true;
    }
}
```

Valid

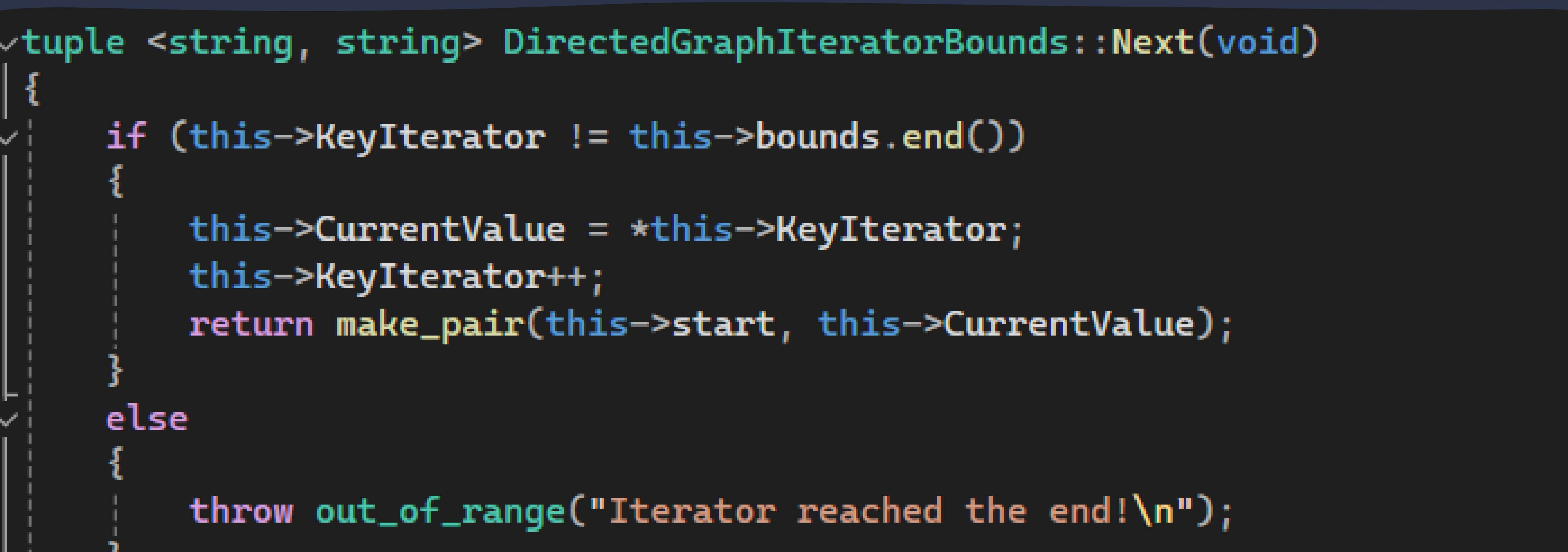
- Returns the state of the iterator



```
bool DirectedGraphIteratorBounds::Valid(void)
{
    return !this->IsItOver;
}
```

Next

- Returns the current edge in the iterator



```
tuple <string, string> DirectedGraphIteratorBounds::Next(void)
{
    if (this->KeyIterator != this->bounds.end())
    {
        this->CurrentValue = *this->KeyIterator;
        this->KeyIterator++;
        return make_pair(this->start, this->CurrentValue);
    }
    else
    {
        throw out_of_range("Iterator reached the end!\n");
```

C++ Implementation
# of Class DirectedGraphIteratorBounds

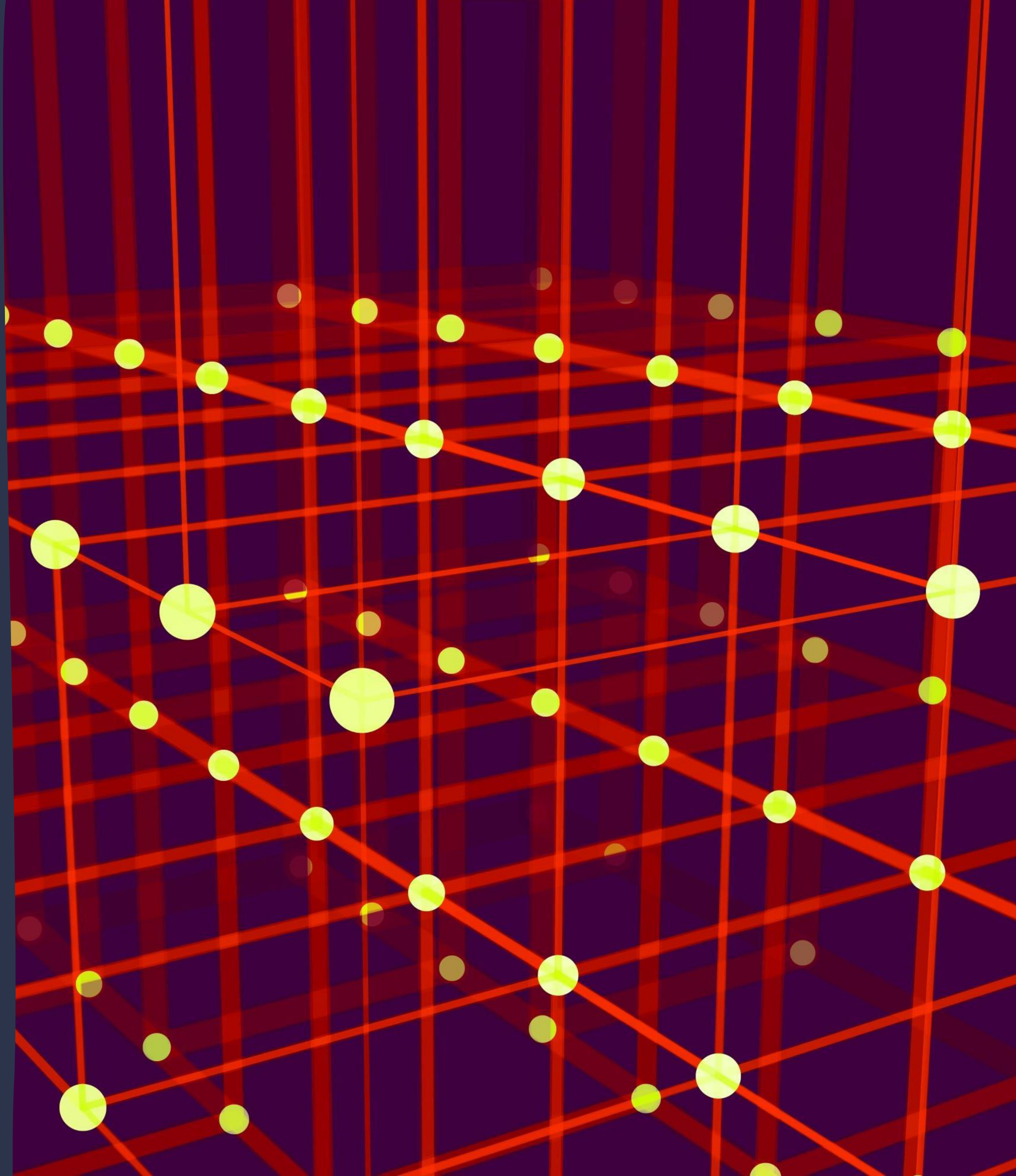
- Friend Class: DirectedGraph

Private Components

- start (string)
- vertices (vector<string>)
- KeyIterator (vector<string>::iterator)
- IsItOver (bool)
- CurrentValue (string)

Private Functions

DirectedGraphIteratorVertices (vector<string> vertices) -> Constructor



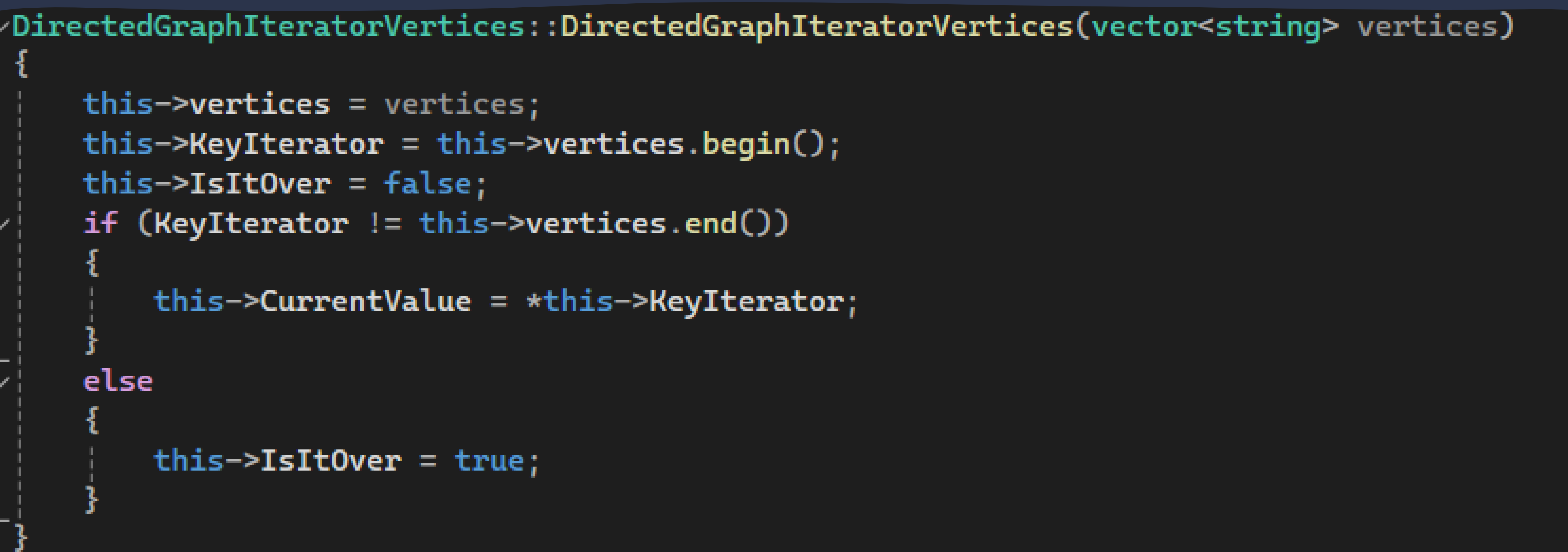
Public Functions

Valid(void) -> bool

Next(void) -> string

DirectedGraphIteratorVertices

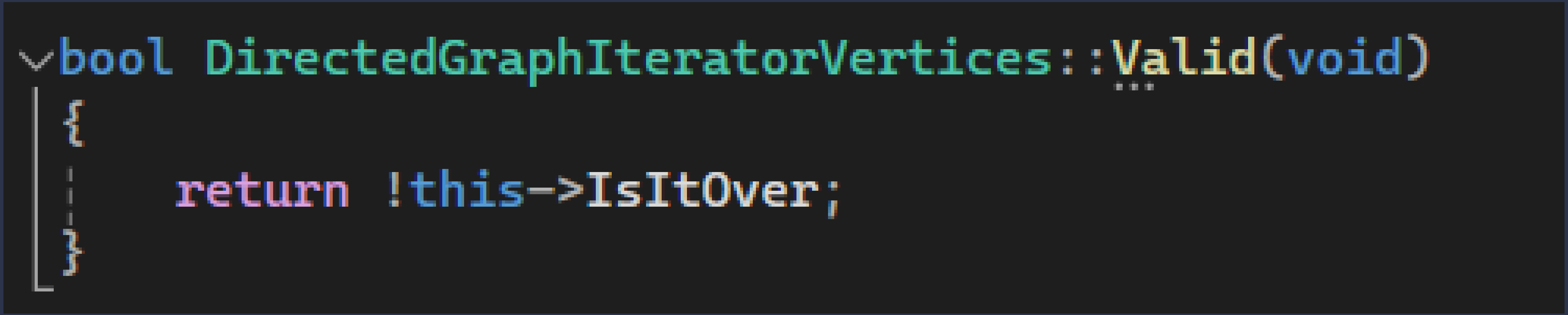
- Initializes the components
- Checks if there are any components



```
DirectedGraphIteratorVertices::DirectedGraphIteratorVertices(vector<string> vertices)
{
    this->vertices = vertices;
    this->KeyIterator = this->vertices.begin();
    this->IsItOver = false;
    if (KeyIterator != this->vertices.end())
    {
        this->CurrentValue = *this->KeyIterator;
    }
    else
    {
        this->IsItOver = true;
    }
}
```

Valid

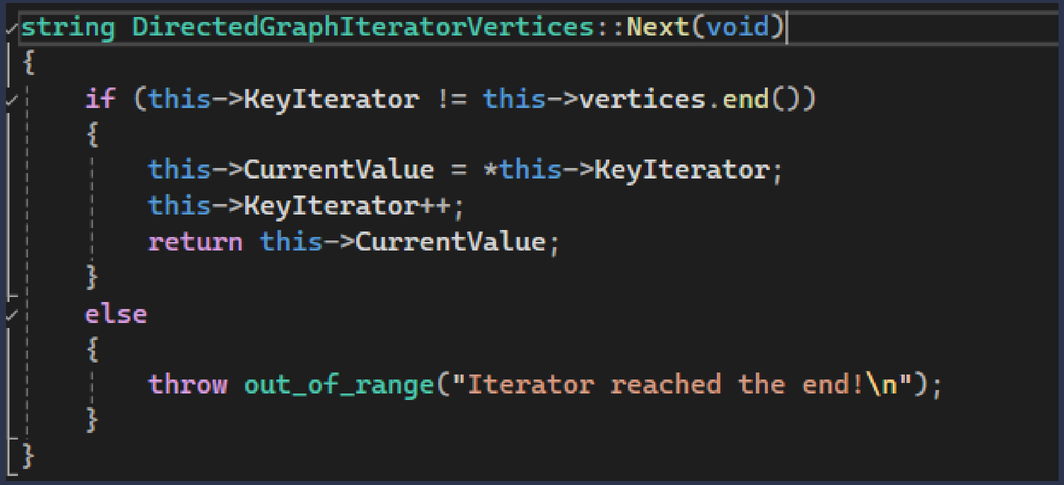
Returns the state of the iterator



```
bool DirectedGraphIteratorVertices::Valid(void)
{
    return !this->IsItOver;
}
```

Next

Returns the current vertex in the iterator



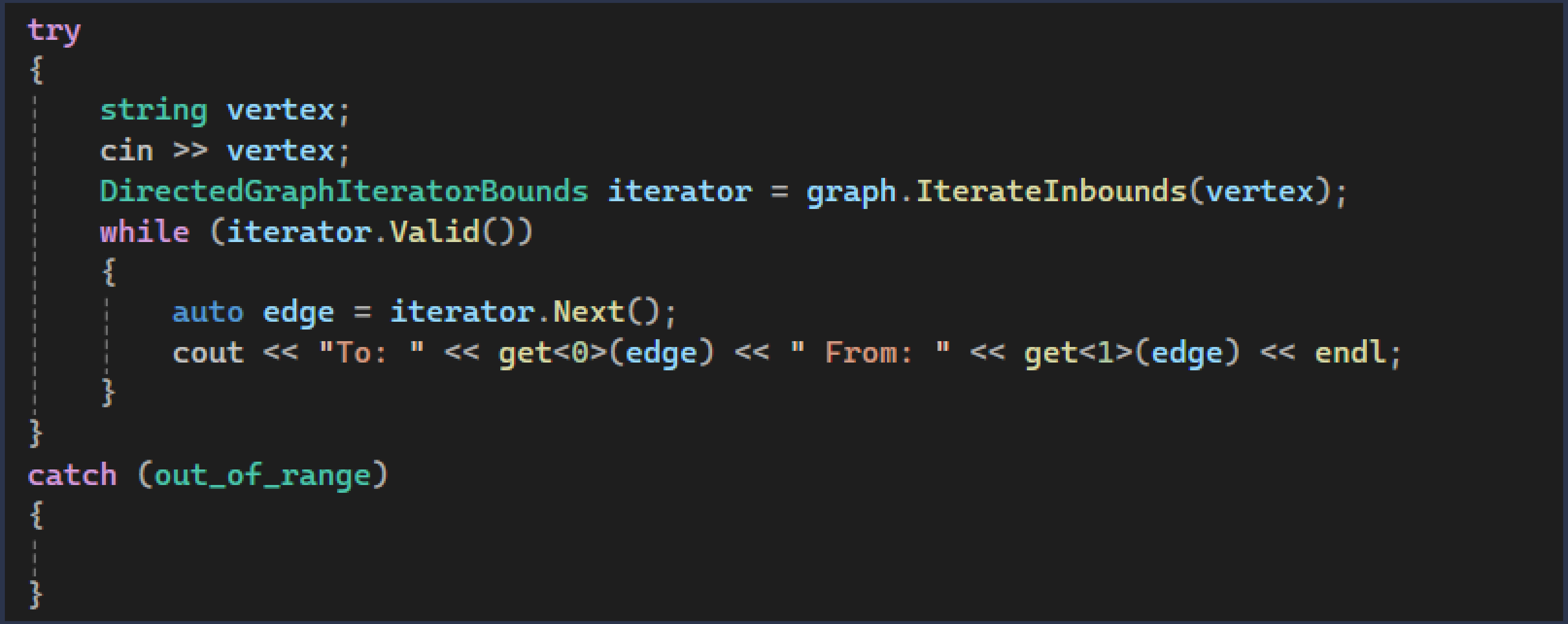
```
string DirectedGraphIteratorVertices::Next(void)
{
    if (this->KeyIterator != this->vertices.end())
    {
        this->CurrentValue = *this->KeyIterator;
        this->KeyIterator++;
        return this->CurrentValue;
    }
    else
    {
        throw out_of_range("Iterator reached the end!\n");
    }
}
```

Personal Notes

- Some of the files given contain less than the supposed number of vertices / edges and my implementation might raise an exception
- I ended up implementing the vertex and edges as strings
- For the bigger files, the amount of time required is REALLY REALLY big
- Main contains the methods to call most of the functions

Using Iterators

- This is an example from main (where UI is implemented)



```
try
{
    string vertex;
    cin >> vertex;
    DirectedGraphIteratorBounds iterator = graph.IterateInbounds(vertex);
    while (iterator.Valid())
    {
        auto edge = iterator.Next();
        cout << "To: " << get<0>(edge) << " From: " << get<1>(edge) << endl;
    }
}
catch (out_of_range)
{

}
```