



Description

Nemesis is on the lose, and for some reason it's after Leon (I guess for training purposes before Jill Valentine becomes the target, this is because I feel Mr. X isn't as cool as Nemesis, so we're sticking to this plot). Leon is trapped in the Raccoon City Police Department, and he needs to find a way out before Nemesis shows up. As always, the fun of Resident Evil is spending hours with puzzles and opening doors and of course the doors are locked so you have to find the key, and you spend weeks in resident evil 3 trying to find keys back before YouTube existed so you either had to struggle through it without any walk through or you were forced to but the play through guide.

Ok, enough with that rant. So in this program, there is a secret escape from the police station, each room has a set of doors that can be opened that goes to a new room, thus we can visualize the police department as a graph. Each room is a node and a door in a room is an edge that goes to a room (a node). The graph could be initially disconnected so you need to find rooms that have keys that unlock a door (i.e. add a new edge into the graph that simulates a new opened-able door). You will need to design the following classes to implement the graph.

Adjacency List

```
template <class Type>
class vertex
{
    struct node
    {
        Type item;
        node * link;
    };

public:
    class edgeIterator
    {
    public:
        friend class vertex;
        edgeIterator();
        edgeIterator(node*);
        edgeIterator operator++(int);
        Type operator*();
        bool operator==(const edgeIterator&);
        bool operator!=(const edgeIterator&);
    private:

```

```

        node * current;
};

vertex();
vertex(const vertex<Type>&);
const vertex& operator=(const vertex<Type>&);
~vertex();
edgeIterator begin();
edgeIterator end();
void addEdge(const Type&);
private:
    node * neighbors;
};

```

Each member of `edgeIterator` class will contain/perform the following

- `node * current` - stores the address of a `node` object where the `edgeIterator` object points to
- `vertex<Type>::edgeIterator::edgeIterator()` - default constructor that sets `current` to `NULL`
- `vertex<Type>::edgeIterator::edgeIterator(vertex<Type>::node * edge)` - a constructor that takes in a `node` object which gets assigned to `current`
- `typename vertex<Type>::edgeIterator vertex<Type>::edgeIterator::operator++(int)` - an operator function that sets the iterator to point to the next `node` object, you will need to set `current` to point to the next `node`
- `Type vertex<Type>::edgeIterator::operator*()` - an operator that dereferences the iterator, returns the `item` field of the `node` that `current` points to
- `bool vertex<Type>::edgeIterator::operator==(const vertex<Type>::edgeIterator& rhs)` - compares the address of the iterator on the left side with the iterator on the right side, returns `true` if they both point to the same `node`, and returns `false` otherwise
- `bool vertex<Type>::edgeIterator::operator!=(const vertex<Type>::edgeIterator& rhs)` - compares the address of the iterator on the left side with the iterator on the right side, returns `false` if they both point to the same `node`, and returns `true` otherwise

Each member of `vertex` class will contain/perform the following

- `struct node` - needed for the adjacency list
- `node * neighbors` - the head of the linked list, the linked list stores all the neighbors of the vertex
- `vertex<Type>::vertex()` - default constructor that sets `neighbors` to `NULL`
- `vertex<Type>::vertex(const vertex<Type>& copy)` - a copy constructor that deep copies the neighbor list of the object passed into the constructor to the object that calls the constructor
- `const vertex<Type>& vertex<Type>::operator=(const vertex<Type>& rhs)` - assignment operator, that performs a deep copy of the right side object with the left side object (the object that calls the operator function)
- `vertex<Type>::~~vertex()` - destructor, deallocates all the nodes in its neighbor list
- `typename vertex<Type>::edgeIterator vertex<Type>::begin()` - returns a `edgeIterator` object whose `current` will be the head of the neighbor list for the `vertex` object
- `typename vertex<Type>::edgeIterator vertex<Type>::end()` - returns a `edgeIterator` object whose `current` will be assigned to `NULL`
- `void vertex<Type>::addEdge(const Type& edge)` - adds a new node into the neighbor list (a head insert would be the best way to implement this)

Hash Map Class

You will use a hash to construct your adjacency list since the nodes are not labeled with indices but rather names (in a string form), thus a hash map is the perfect structure to use here. Once again, you can use STL `unordered_map` here, but if your program works with `hashMap`, you will be awarded extra credit points.

Contents of main

You will be given two input files (one with your edges), each line will contain two room names, separated by a space, each line is terminated with an end of line. You need to create an undirected graph, i.e. the following line of input

```
MainHall WaitingRoom
```

Means that there is an edge from MainHall to WaitingRoom and from WaitingRoom to MainHall. Since each door opens from both sides. Thus your structure to maintain your adjacency can be seen below.

- `hashMap< string, vertex<string> > map;`
- `unordered_map< string, vertex<string> > map;`

So if you want to add an edge from MainHall to WaitingRoom you would have

```
map["MainHall"].addEdge("WaitingRoom");
```

Since in the hash map, a `string` maps to a `vertex<string>`, then `map["MainHall"]` returns a `vertex<string>` object that stores all the neighbors of "MainHall", but in this case it was first found so an empty `vertex<string>` object is returned when the new key is added into the hash object. If you want to traverse all the neighbors of "MainHall" you can have the following code

```
vertex<string>::edgeIterator it;  
  
for (it = adjList["MainHall"].begin(); it != map["MainHall"].end(); it++)  
    cout << *it << endl; /*it is the name of a neighbor of "MainHall"
```

The other file you are given contains the rooms that have a key that opens some door in the police station. Each line contains 3 strings (each separated by a space) and each line is separated by an end of line. The first string in the line contains which room the key is located in, and the other two strings contain which door this key opens (i.e. the edge between two nodes), so for example if a line contains

```
FileStorage MainHall PayphoneCorridor
```

Then FileStorage contains a key that opens the door that connects MainHall to PayphoneCorridor and vice versa. So you would need to add this edge into your graph when a key is found. You will need to do several DFS traversals to find all the keys and open the doors. If a path can be found to a room called "Exit" exists then you win and Leon is saved, otherwise...I guess Ada might mourn for Leon. So I used the following structure to maintain the key locations

```
hashMap< string, vector<string> > keys;  
unordered_map< string, vector<string> > keys;
```

So for the above example I would have

```
keys["FileStorage"].push_back("MainHall");  
keys["FileStorage"].push_back("PayphoneCorridor");
```

Thus `keys["FileStorage"][0]` contains "MainHall" and `keys["FileStorage"][1]` contains "PayphoneCorridor" so when you arrive at this room, you add the edge (MainHall, PayphoneCorridor) to the graph. Here is the function that I wrote, this is a DFS traversal that gets called several times in main

```

string running(string curr, hashMap<string, vertex<string> > map,
              hashMap<string, vector<string> > keys, hashMap<string, bool>& visited)
{

}

```

This function returns a string back, which tells us which room of significance Leon has reached, i.e. if Leon reached "End" or if Leon reached a room with a key, so when we return back to main, we know which node to restart DFS from. From main when this function is called, if "" is returned then we know Leon is trapped, if "Exit" is returned we know Leon escaped, if any other string is returned, you add the edge into the graph (use the string returned as the string for the **keys** hash map), and then run this DFS again from the room where the key was found, so you would have a loop in main that would keep calling this function and pass a different string into the curr field each time. **YOU FIRST START IN THE MAIN HALL**

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- No global variables (global constants are ok)
- Make sure your program is memory leak free
- Extra credit will be awarded if you use **hashMap** instead of **unordered_map** for the adjacency list

Sample Run

Your program prompts for the police station map file and the keys file, if Leon can escape you output

```
"Ok Leon, you escaped the police station, now to find Ada"
```

If Leon cannot escape then you output

```
"Ok Leon, your first day will be your last day on the force"
```

Submission

Submit your source files to code grade by the deadline, you must submit a file called **main.cpp**, optionally you can also upload **hashMap.hpp** if you want to get extra credit

References

- Link to the top image can be found at https://residentevil.fandom.com/wiki/Nemesis-T_Type/gallery