GVZork

Note: For this project you are able to use an AI assistant such as ChatGPT or CoPilot. Just note in your comments the code that came from it.

One of the greatest text-based games of all time - Zork - remains popular with enthusiasts to this day. While modern games rely on impressive visuals, Zork relied upon a good story and imagination. If you've never played it, you can do so at

https://classicreload.com/zork-i.html

One of the best parts of the game was its flexible interface. Commands could have more than one syntax. For instance, we might type "Go north", "Proceed north", "Walk north", etc. Either would perform the same function.

We will write our own GV-themed version of the game using Object Oriented C++. We will have the following classes:

Item

An `Item` represents objects the player may encounter along the way, and is an object that has a name, a description, the number of calories it can provide (0 if it isn't edible), and a weight. It must provide:

- A constructor that can take a name, description, calories, and weight.
  - You must ensure the following (raise exceptions for violations):
    - The name variable cannot be blank.
    - The calorie variable cannot be less than 0 or more than 1000 and must be an `int`..
    - The description cannot be blank.
    - The weight must be a `float` and must be between 0 and 500.
- An overloaded stream operator that returns a string representing the `Item`. The format should be

  `NAME (X calories) - X lb - DESCRIPTION`, i.e.

  ```
  Rusty Nail (0 calories) - 1 lb - A rusty nail (I hope you've
  had a tetanus shot)
  ```

NPC

The NPC class represents information about a character that can be in a `Location`. An NPC has a name, description, message number, and vector of messages. Each time the player speaks to an NPC, the message number should increase by 1, resulting in the next message in the vector being printed the next time the player speaks to the character. The message number should go back to 0 after it goes past the length of the message vector. The class must have:

- A constructor that accepts a name and a description (neither can be blank).
- Getters for name and description.
- A getter for a message that returns the current message (as indicated by message number), then changes the message number appropriately.
- An overloaded stream operator that returns only the name of the NPC.

Location

Each `Location` represents a place on campus that may be visited. A location holds a name, a description, a bool that indicates if it has been yet visited, a map of neighbors, a vector of `NPC`s, and a vector of items. The neighbors map will have a key that is a direction (a string), that refers to a `Location` as its value. For instance, if we have a `Location` called `zumberg` and another `Location` called `kirkhoff`, `zumberg` would have a map entry "west" that holds `kirkhoff`. We then want `kirkhoff` to hold in its map a key-value pair of "east" that refers to `zumberg`. If we want to have two locations attached such that the player can enter but not exit the way they came, we can merely leave the entry off one of the objects. For instance, we could have a `portal` `Location` that has a "through" key pointing to `zumberg` but no corresponding key-value pair in the `zumberg` neighbor map.

`Location` must implement:

- A constructor that takes a name and description.
- A `std::map<string, Location> get_locations()` function that returns the neighbors map.
- A `void add_location(string direction, Location location)` method. This function will add the location into the map with the provided direction string. If the string is blank, raise an exception. If the key already exists in the map, raise an exception.
- An `void add_npc(NPC npc)` method for adding an `NPC` to the `Location`'s vector, and a `std::vector<NPC> get_npcs()` function that returns the vector of `NPC`s.
- A `void add_item(Item item)` method for adding an `Item` to the `Location`'s NPC vector, and a `std::vector<Item> get_items()` method to return the `Item`s.
- A `void set_visited()` method that changes the visited variable to true. Once a location is visited, it can no longer be false. Also, include a `bool get_visited()` function for checking if the location has been visited.
- An overloaded stream operator that returns a string in the form of

```
Padnos Hall - Lots of science labs are in this building.

You see the following NPCs:
    - Troll
    - Talking pumpkin
```

```
You see the following Items:
    - Cookie (10 calories) - .5 pounds - A delicious M&M cookie.
    - Rusty Nail (0 calories) - 1 lb - A rusty nail (I hope you've had a tetanus shot).

You can go in the following Directions:
    - East - Zumberge (Visited)
    - North - Unknown
```

Game

The game takes place in a world of connected `Location`s. The purpose of the game is to collect edible items and then bring them to the elf in the woods behind campus. The elf needs 500 calories of edible food before it will save GVSU. Each item can have 0 or more calories. If a player gives the elf something inedible (something with 0 calories), the elf will be displeased and will teleport the player to a random location in the world. The player can only carry 30 pounds at a time, so multiple trips to the elf may be needed. Once the elf has enough calories, it will save campus and the game will end.

The `Game` class holds all the logic for a game instance. This class will hold several pieces of data. Make sure it has

- a `map<string, Command>` of commands. Note that `Command` here is simply an alias for a function that returns nothing and takes a `std::vector<std::string>`.
- a `std::vector<Item>` of `Item`s the player currently has in their inventory
- an `int` that holds the current weight the player is carrying
- a `std::vector<Location>` of `Location`s that exist in the world
- a variable to hold the player's current location
- an `int` to hold the number of calories the elf needs before it will save the campus
- a bool to store whether the game is still in progress

`Game` will adhere to the **Command Pattern**. Instead of having a loop body with a large number of `if` statements, for instance (this is pseudocode, not valid C++):

```
command = input("What is your command? ")
if(command == 'help'){
    help()
}
else if(command == 'talk'){
    talk()
}
else if(command == 'go'){
    move()
}
```

```
... On and on ...

else if(command == 'quit'){
    quit()
}
```

We can instead create a data structure that maps commands to functions. C++ makes this simple with dictionaries. We could for instance do something like this:

```
void show_help(std::vector<std::string>){
    ... printout stuff ...
}

std::map<string, function<void(*)(std::vector<string>)> commands;
commands["show help"] = show_help;
commands["help"] = show_help;   // an alias for the original command
... Add more commands ...
```

In this example, the values of the map (i.e. `show_help`) are the *names* of functions. Note that we are not *calling* the functions by putting the parentheses after the name (we didn't type `quit()` for example); we are merely giving the name of the method. Now, our input loop can look something like

```
command_input = input("What is your command? ")
... tokenize strings such that command is a string ...
... the rest of the typed command will be a vector of strings ...
... i.e. "take the rusty_sword" command = "take", and the ...
... vector of strings will hold "the", "rusty_sword" ...
commands[command](tokens)
```

Notice here that we *did* use the parentheses to call the function. We can do this because C++ has **first-class functions** (well, pointer references to them at least!). This simply means that a function can be used just like any other piece of data. For instance, we can store a function in a vector or map (as we did here), or we could pass a function as a parameter to another function.

The Command Pattern allows us to abstract away all of the `if` statements. We merely call the function that corresponds with the typed command (if one exists). Note that we can check if the command is a key of our map with the `in` keyword:

```
if(input in commands):
    ... do something ...
```

The `Game` class will require the most code to be written. It needs

- a constructor that takes no parameters. The constructor will set the commands map equal to the return call from our `setup_commands` function. It will call the `create_world` method. It will then set default values for

all other variables. Set the current location to a random location selected from the `random_location` method.

- a `void create_world()` method that creates all the `Locations`, `Items`, and `NPCs` in the game. This function can get messy, as it will have a lot of text for names and descriptions of objects. Lines of code should not be more than 72 characters long. If a line goes beyond that, you should separate it into more than one line. You can do this by splitting the string:

  ```
  kirkhoff_upstairs = Location("kirkhoff upstairs", "The student union.\
                There are restaurants, a store, and\
                places to congregate.")
  ```

  This implicitly concatenates the two strings together, so formatting this way poses no issues.

  In this function you will need to add all `Items` and `NPCs` to the rooms in which they belong, as well as add each `Location` to the neighbors to which it needs to connect. Because there will be so much setup code, you may wish to break the function into commented regions, wherein each region focuses on the creation and setup of a single `Location`. Your game needs at least 8 `Locations`. There is no requirement for how many `Items` and `NPCs` must be in each `Location`, but your game does need at least 10 `Items` and 5 `NPCs`.

- a `std::map<std::string, void(*)(std::vector<string>)> setup_commands()` method. This method will create a new map. The keys will be a string such as `talk`, `give`, `go`, etc. The values of the map will be the names of the functions that should be called for each of those commands. Note that we can have more than one command per function; for instance, we could have both "?" and "help" correspond to a `show_help` method. Be sure this function returns the map.

- a `Location random_location()` method that selects a random `Location` from the locations vector and returns that `Location`.

- a `void play()` method. This is the core game loop. It should first print a message describing the game, then call the method to vector all commands. Then, while the game is still in progress, it will loop. In the loop, we will prompt the user for a command. The user may enter multiple words in a prompt. We will split the user's input into a vector of words. We can split on spaces with code `tokens = user_response.split()`. Once we have the tokens vector, create a variable called `command` and set it equal to the first element in the vector. Then, remove the first element with the `del(tokens[0])` command. Then, use the code `target = ' '.join(tokens)` code to put the remaining tokens together. Thus, if the user enters

```
talk ball of light
```

then `command` will be equal to `talk`, and `target` will be `ball of light` split into a std::vector on the spaces.

We will then call the function from the commands map by using this key. Pass `target` as a parameter to the called function. If the command does not exist in the map print a message to the user telling them so.

Once the loop ends (i.e. the in-progress variable is false), check if the elf got enough calories. If it did, print a success message and quit. Otherwise, print a failure message and quit.

- a `void show_help()` method that prints out a help message and all the commands from the command map's keys. This method must also post the current time. You will need to read the `datetime` documentation to do this - https://docs.python.org/3/library/datetime.html. Print the time in a nicely formatted string, but you can decide if you wish to use 12 or 24-hour time.
- a `void talk(target)` method. This method will check if the provided NPC is in the current room. If it is, it will call the NPC's `get_message` method and print the resulting message.
- a `void meet(std::vector<std::string> target)` method. It will check if the NPC is in the room, and if it is will ask the `NPC` for its description and print it.
- a `void take(std::vector<std::string> target)` method. If thestd::vector target item is in the room it will remove it from the room's inventory, add it to the user's inventory, and add the weight of the item to the user's carried weight.
- a `void give(std::vector<std::string> target)` method. Removes thestd::vector target item (if it exists) from the user's inventory, adds it to the current location's inventory and decreases the user's carried weight. The function will then check if the current location is the woods. If it is, it will check if the item given was edible. If the item is edible, reduce the number of calories the item has from the total the elf needs. If the item was not edible, transport the player to a new location by setting the current location equal to the return from `random_location`.
- a `void go(std::vector<std::string> target)` method. Sets the current location's visited status to True. Checks if the player has over 30 weight; prints a message and returns if so. Otherwise, it checks if provided direction exists in the current location's neighbor map. If so, sets the current location equal to the value from the map.
- a `void show_items(std::vector<std::string> target)` method. This method doesn't need any parameters but has a parameter so it can be called with the same syntax as the other commands. It should print all items the player is carrying, and the current amount of weight carried.

- a `look(std::vector<std::string> target)` method. This method doesn't need parameters either but has a parameter for the same reason as given above. This method will print the current location, a vector of `Item`s in the location or a message indicating there are none, a vector of `NPC`s in the room or the message "You are alone.", and a vector of directions in which the player can go. If a location has been visited previously, print the direction and the location. Otherwise, simply print the direction.
- a `quit(std::vector<std::string> target)` method that prints a failure message and exits the game.
- two additional command functions that you design and create. You could add teleportation, magic, etc.

Submission

You may work on this project in groups of up to 3. While you may work on it individually or with a single partner, you are assuming the risk and responsibility of taking on the extra work. This project can be challenging as it makes use of multiple types of data structures, a design pattern you may not have encountered before, and is made up of several classes. And - if you aren't already used to C++ code, the syntax can be tough at first.

Submit only the C++ file(s) that you created. **Do not submit a .zip file**. Zipped files cannot be marked up in Blackboard, meaning I have no easy way to give you feedback on your code.

Ensure that **every** file has all team members' names at the top in the documentation, as well as a date. If your name is not on the file, you will not receive a grade. By allowing your name to be placed in the file you are attesting that you worked on this file an adequate amount and that you understand **all** code in it. Failure to adhere to these criteria may be academic dishonesty and can result in penalties.

Grading

Your code will be graded on the following by the following rubric. You can receive a maximum of the vectored percentage of points for each section:

- 10% Commented well, including headers for all classes and methods.
- 10% Uses the C++ style we discussed in class, with declarations in .h files and definitions in .cpp files.
- 10% Item class works as specified.
- 10% NPC class works as specified.
- 10% Location class works as specified.
- 10% Adheres to the Command Pattern.
- 10% Includes at least 8 connected `Location`s, 10 `Item`s, and 5 `NPC`s. You can add more (but no extra credit will be given).

- 10% Does not make use of the `using` directive. Uses complete namespaces.
- 10% Two extra commands and corresponding functions are created and implemented correctly.
- 10% The game is playable as described.