C++ Project – write a Zork game

For the project we will be looking back to one of the first interactive fiction computer games called Zork, a text-based adventure game. If anything is incomplete, inconsistent, or incorrect in this specification, please ask questions on Piazza. The specification for the project and various steps may be changed to clarify items causing confusion. Note also that things that are not specified can be handled as you wish. Finally, there are example dungeons and output from playing the dungeons – your output should be similar in spirit to this output. For example, if the output is "Error!", I am ok with anything that indicates an error occurred.

You will implement a simpler variant of this game which will be able to read as input an XML file with complete information of a specific adventure and create the set of objects that forms the environment that the player interacts with.

An example game can be found at: http://textadventures.co.uk/games/play/5zyoqrsugeopel3ffhz_vq . Play these and see how the game play goes. This will be useful in interpreting the directions that follow. A video of Zork being played can be found at https://www.youtube.com/watch?v=TNN4VPlRBJ8.

Do not download any open-source program other than an XML parser, and modify it, for this assignment.

The overall idea of the game and project.

The game will be described by an XML file which describes the various game elements. There is a player, various rooms, containers (for example, a chest) and items (for example, a sword or key) in the rooms. The rooms are connected, and the player navigates by means of one letter commands (n, s, e, w) which moves the player up, down, right or left. If there is another room in that direction, the player will move to that room.

Players have an inventory of items they are carrying. Players can check their inventory (*i*), *take* an item from the room or a container and add it to their inventory, *drop* an item that is in their inventory, *open* a container, and so forth, as described below. Items in the room are described by the XML file.

Certain player actions are subject to *triggers*. Triggers contain conditions that must be fulfilled, and actions they perform when those conditions are met. These actions then allow certain activities to be performed. Triggers, their conditions, and their actions, and other properties, are described in the XML file.

Players can attack creatures. Creatures are only vulnerable to certain forms of attack, thus attaching a werewolf with a chest may be ineffective. The vulnerabilities of creatures are described in the XML file.

The document DungeonExplained.pdf gives a description of the information in the XML file.

In Game Commands:

n, s, e, w; movement commands to put the player in a different room. As you might have noticed from playing the sample game, once the player enters a room it doesn't make any difference where in the room the player is – everything in the room is equally accessible from anywhere in the room.

If there is another room in the direction given by a movement command, the player will be moved to the room and a description of the new room will be printed to the screen. If there is not another room in the direction indicated, print *Can't go that way*.

i: short for inventory. This command print *Inventor:* and then lists all of the items in the player's inventory, with items separated by commas if there is more than one. If there are no items in the inventory, display *Inventory: empty*.

take <item>: Changes the ownership of the item named *<item>* from the room or container (e.g., a chest) it was contained in to the inventory of the player. If successful, print *Item <item> added to the inventory*. Hint: can be written as shortcut for the *put* command.

open <container>: prints the contents of the container as *<container> contains <item1>, ..., <itemn>* , where *<container>* is the name of the container, and *<item1>, ..., <itemn>* is a comma separated list of the names of the items in the container. As well, the command makes the items available to be picked up using the *take* command. If the container is empty, the *open <container>* command will print *<container> is empty*.

open exit: If a room is of type exit this command prints *Game over* and gracefully (i.e., no seg fault, etc.) ends the program.

read <item>: If there is writing on an object, this command will print what is written, and otherwise will print *Nothing written*. The item must be in the players inventory. If it is not, then the command should say *<item> not in inventory*.

drop <item>: Removes an item from the inventory and makes the room the player is in the owner of the item. The command should print *<item> dropped*. If the item is not in the inventory, print *<item> not in inventory*.

put <item> in <container>: Removes *<item>* from the inventory and puts it in the container. The ownership of the item is given to *<container>. <container>* must be open, if not, print *Cannot add <item> to closed <container>.* Note that the inventory is a container, and is always open. If *<item>* is not in the inventory, print *<item> not in inventory.*

turn on <item>: Activates *<item>* if it is in the inventory, otherwise print *<item> not in inventory.* The item is activated, and any *turnon* activities associated with *<item>* are executed.

attack <creature> with <item>: prints *You assault the <creature> with the <item>.* For the command to be successful the item must match the creature (as specified in the .xml) and any existing conditions, also specified in the .xml, must be met. If successful, any *attack* elements associated with *<creature>* are executed.

Behind the Scenes Commands

Add <object> to <room/container>: creates an instance of <*object*> with the specified room or container being the owner. This does not work with the inventory.

Delete <object> Removes <*object*> from the game, but it can be returned to the game by using an *Add* command. Rooms cannot be added back in if deleted.

Update <object> to <status>: creates a new status for <*object*>. Object status can be checked by triggers.

Game Over: ends the game and prints *Victory!*

Order of Operations:

When the user enters a command:

1. Check if command is overridden by any triggers, and if not, execute the command.
2. Check if effects of command activate a trigger, and if so, activate the trigger and then see if the new status activates additional triggers, until no new triggers are activated.

Objects (If an element is followed by [ ] there may be several of them)

A Room may contain the following elements: name, status, type, description, border[ ], container[ ], item[ ], creature[ ], trigger[ ].

- *type* is assumed to be *regular* unless specified otherwise
- item may contain a *name*, *status*, *description*, *writing*, *status*, *turnon*, and *trigger*[ ]. If an item has a *turnon* element and the *turnon* command is issued by the user, action elements *turnon* are to be executed all conditions are met.
- *container* may contain *name*, *status*, *description*, *accept*[ ], *item*[ ], *trigger*[ ]. If an *accept* element is added, only specific items, specified by the *accept* element, may be put into the container *and the container need not be opened to insert them and cannot be opened until it is inserted! accept* elements are often keys.
- *Creature* may contain *name*, *status*, *vulnerability*[ ], *attack*, and *trigger*[ ]. If the *attack* command is issued by the user with creature's vulnerability, *action* elements in *attack* are executed if all given conditions are met.

Special Objects.

Triggers contains *condition*[ ] (including special conditions of type *command*, all of which need to satisfied for the corresponding actions to take place: *type*, *print*[ ], and *action*[ ].

Possible conditions are *owner*, and *status*. *Owner* will have an *object*, and owner elements. *Status* will have an *object* and *status* elements.

type is assumed to be *single* (only used once) unless specified as *permanent* (unlimited uses). The order of execution is

1. output any 'print' action
2. other 'action' commands in the order given in the .xml file.

Possible commands are any user command, and the entire string must match, e.g., *take scroll* will not match *take sword*. The trigger will pass the 'command' portion of one of its conditions if there is either no command element or if any one of the command element's contents are matched.

Context is important. Only triggers in the present context of the game should be checked, i.e, the inventory, the present room, and any items, containers, creatures, and items in containers in the present room. The actions triggers perform, however, can act on any object in the game, including triggers in other rooms.

XML Formatting

Root element: <map>

Each type of object will have elements associated with their given descriptions above with the addition of a 'name' element.

Triggers will have the additional complexity of containing a condition and action elements, with the condition having three additional elements to create an 'if' statement of the form "if (object) is/isn't in (owner)" with the is/isn't being determined by the <has> element.

Grading (total 100 points)

- 24, (3 test cases) rooms - movement between entrance, regular room(s), and exit
- 24, (4 test cases) items - take/drop, read, turn on, and add/remove
- 16, (4 test cases) containers - take, put in, open, restrictions (accepts element)
- 8 (2 test cases) creatures - attack and add/remove
- 28 (5 test cases) triggers - permanence and activation with each other object type

NOTES:

- Commands are case sensitive.
- Triggers always take precedence over default actions, and are only relevant in their given contexts (i.e, the trigger associated with a room will not be tested if player is not in that room). When testing for triggers, test only triggers of objects in your present context (the room the player is currently in room, objects in the room the player is currently in room, objects in containers in the room the player is currently in, and the player's inventory), however, they can affect any object in the map.
- The initial room will always be named "*Entrance*".
- If an error message is not specified, the general error message is "Error".

Tips:

- A good structure might have classes corresponding to each object type, with all objects inheriting from a common class which can be searched for triggers
- The program should be clearly divided between creation and play. For creation, use XML parser, don't write you own. Later in the week we should have some example parsers.
- Check Piazza for updates and don't be afraid to ask questions!

Sample Run Through:

- Here's a sample XML File: sample.xml
- And the corresponding run with output: RunThroughResults.txt
- NOTE: in the RunThroughResults.txt file the ">" character is to depict input and should NOT be included in your project! It is for ease of reading only!!!
- Sample pack for further testing: samplepack.zip

Submission

- Please submit your program to Brightspace. More detailed information will be given in the instructions for each step.