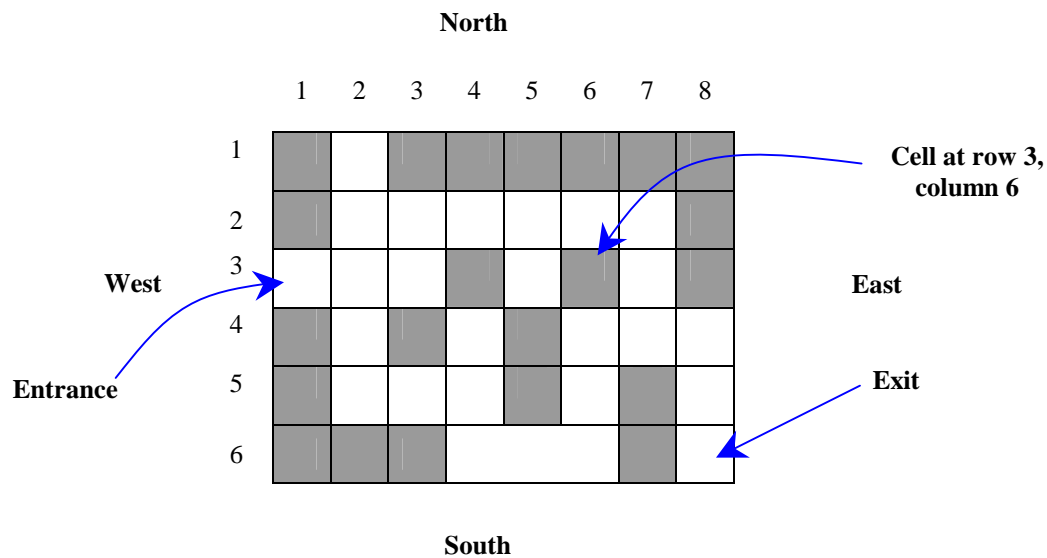


House of Horrors

For this project, you will be modifying and extending the design for MazeCrawler, and then implementing a revised prototype for a simple game. As before, the “moves” in the game will be specified by a list of commands given in a text input file. There is no graphical interface, but feedback will be provided in an output file.

Maze (no changes): a rectangular grid of cells, each identified by a pair of integers specifying its location (row and column) in the grid. Initially, each grid cell may be empty or be a wall. The locations of walls will be specified in an input file and will not change during the course of a game. Grid locations are labeled as shown below, with the indicated conventions regarding directions:



The maze will always have two cells designated as the Entrance and Exit. These cells may be at any location on the boundary of the maze, including corners, but may not be in the maze interior.

Dramatis personae (changes): As before, four types of creatures, three species of MazeMonsters and Human, may inhabit the maze. Each creature will always have a particular location (cell) and may move in any of the four cardinal directions (East, North, West, or South) but not diagonally. Also as before, neither type of creature knows whether an adjacent cell is open or a wall cell; it must discover that fact by attempting to move into that cell.

Each creature will also have an energy level, represented by a nonnegative integer. Moving from one open cell to an adjacent open cell uses one unit of energy. If a creature’s energy level drops to zero, it dies (and disappears from the game).

A creature may also attempt to move out of its current cell in a direction that is forbidden (e.g., blocked by a wall, or outside the maze boundaries). In that case, it remains in its current cell and loses one unit of energy.

A MazeMonster is familiar with the maze and always knows its current location, not that a MazeMonster is smart enough to actually take advantage of that information. A Human never knows his/her current location within the maze.

A Human will not enter a cell containing a MazeMonster. If a MazeMonster (except for a Wampyr) enters a cell containing a Human, it immediately eats the Human, increasing its energy level by the amount of energy the Human possessed. If a MazeMonster enters a cell containing another MazeMonster the result depends on their relative energy levels. If both have the same energy level, then both die immediately. If they have different energy levels, then the one with more energy eats the one with less energy; in this case, the winner’s energy level remains unchanged.

At any given time, there may be zero to ten MazeMonsters in the maze and zero or one Human.

MazeMonster Taxonomy: MazeMonsters come in three varieties (not counting the “standard” type introduced in the previous project). Except for the explicit behavioral modifications given below, each variety of MazeMonster has exactly the same properties as the “standard” type from the previous project.

Juggernaut A Juggernaut can only step, not trek. As compensation, a Juggernaut can step into a wall cell. When a Juggernaut steps into a wall cell, the wall is not destroyed, and the Juggernaut uses 10 units of energy. Also, while within a wall cell, a Juggernaut is invisible to Humans. A Juggernaut uses 1 unit of energy in stepping from a wall cell to an open cell.

Teleporter A Teleporter can “jump” from its current location to another location within the maze (so we add a new command, `jump`). Unfortunately for Teleporters, they have no way to check if the target of a jump is an open cell or a wall cell, and unlike a Juggernaut, a Teleporter does not thrive while inside a wall. So, if a Teleporter jumps into a wall cell, it dies immediately. The energy required for a jump is the number of units distance from the Teleporter’s original location to the target cell, measured using the taxi-cab metric. A Teleporter cannot jump to a location outside the maze, and a command to do so has no effect.

Wampyr A Wampyr moves in exactly the same manner as the “standard” MazeMonster from the previous project. If a Wampyr enters a cell containing a Human, it consumes 10 units of the Human’s energy (increasing its total energy by the amount it consumes). If the Human’s energy level is less than or equal to 10, the Wampyr consumes all of it and the Human dies. Otherwise, the Human survives and the Wampyr does not drain any additional energy while both remain in the cell where the attack occurred.

Obviously, these varieties of MazeMonster should be related within a single inheritance hierarchy.

Victor’s Laboratory: MazeMonsters are created in Victor’s laboratory (on command from the input file).

Once a MazeMonster is created, regardless of its type, Victor’s Lab will associate it with the Maze (or other suitable controller) by passing the address of the new MazeMonster to the Maze/controller.

The Maze/controller will then create a reverse association with the MazeMonster by passing the MazeMonster its (the Maze/controller’s) address.

The make monster command may or may not specify which type of monster is to be created (see next page). Victor’s Lab will always be running a daily special on one of the three monster types; initially, the special will be for Juggernauts. If a make monster command does not specify a monster type, Victor’s Lab will deliver the current daily special.

See the section on Design Requirements for additional details.

After the associations are established, Victor’s Laboratory and the MazeMonster will have no further interaction. Humans are not created in Victor’s laboratory.

Driver file description:

Your program **must** read its input from a file named `Maze.in` — use of another input file name will annoy the person conducting your project demonstration. The format and contents of the input file are exactly the same as in the previous project.

The `make` command is altered to specify the type of the `MazeMonster` that is to be created. We add three reserved words: `juggernaut`, `teleporter` and `wampyr`. A `make` command has the following syntax, where

```
make      [juggernaut | teleporter | wampyr | monster] <name> <location> <energy>
```

If a monster type is specified then Victor's Lab creates that type of monster. If no monster type is specified (i.e., `make monster ...`) then Victor's Lab will deliver the daily special.

If the specified name is not in use, and the location is valid (within the maze boundaries and not a wall) and the energy value is positive, then Victor's laboratory will create a `MazeMonster` as specified. Otherwise, the command does not alter the state of the game.

```
make      human      <name>      <energy>
```

If a Human is already in the maze, the command does not alter the state of the game. Otherwise, if the specified name is not in use, and the energy value is positive, then a Human will be created in the Entrance cell for the maze. In the unfortunate event that there is a `MazeMonster` in that cell, the Human is immediately eaten.

A command is added to set the Daily Special in Victor's Lab; `daily` and `special` are reserved words.

```
daily      special      [juggernaut | teleporter | wampyr]
```

If the third parameter is one of the monster types shown, this command sets the daily special for Victor's Lab. Otherwise it has no effect.

```
<name>      step          [East | North | West | South]
```

If the name is not the name of an existing `MazeMonster` or Human, the command does not alter the state of the game.

If the name is that of an existing `MazeMonster` or Human, the named creature will attempt to move to the adjacent cell in the specified direction (the target cell). If the target cell is a wall or lies outside the maze (see special cases below), the creature will lose one unit of energy and remain in its current cell. If the cell is empty, the creature will move into that cell and lose one unit of energy.

If the named creature is a Human, and a `MazeMonster` occupies the target cell, the Human will lose one unit of energy and remain in his/her current cell. If the named creature is a `MazeMonster`, and the target cell is occupied by a Human or another `MazeMonster`, then the creature will move into that cell and the rules stated earlier apply (someone's probably going to die).

Special cases: If the creature is in either an Entrance or Exit cell, and the specified direction would take it out of the maze, then it will make the move and disappear from the game. If the creature is a Juggernaut, and the target cell is a wall, then the Juggernaut will enter the wall cell as described earlier.

```
<name>      trek          [East | North | West | South]
```

This is simply an iteration of the `step` command. The named creature (unless it is a Juggernaut) will move as far as possible, given the rules for the `step` command, in the specified direction. Juggernauts do not comprehend the `trek` command and will not trek if asked to. Warning: be careful with your calculation of the energy consumption.

<name> **jump** <location>

This command applies only to Teleporters. If the named creature is not a Teleporter the command does not alter the state of the game. If the named creature is a Teleporter, its location will change to the one specified in the command unless that is outside the maze. If that location is a wall cell, the creature dies immediately. If it is an open cell, the Teleporter loses an amount of energy equal to the distance it has moved (using the taxi-cab metric). Warning: be careful with your calculation of the energy consumption.

<name> **see** **monster?**

If the named creature is an existing Human, then the Human will “look” (not move) as far as possible in each direction from its current location, attempting to spot a MazeMonster. (This will evidently require that the Human have some sort of communication with the game entity that keeps track of current locations of creatures and the maze layout.) If the Human spots any MazeMonsters, he/she should produce output signaling that and specifying the direction(s). This command does not use any energy.

This command applies only to Humans, so if the named creature is a MazeMonster, the command should be treated as an error.

<name> **status**

If the name is that of an existing creature, the name, creature type, location and energy level of the named creature will be displayed to standard output (cout).

Note: the type should not be stored as a data member in each object. This should be handled by polymorphism.

display status

For every existing creature, display its name, type, location and energy level to standard output (cout).

display maze

Display to standard output (cout) a map of the maze. The map should indicate clearly which cells are empty (space), which cells are walls ('W'), which cells contain a Human and which cells contain a MazeMonster. This map should be relatively compact, and include row and column numbers. This is one situation where adding whitespace does not increase clarity.

Indicate the presence of a creature by printing the first letter of its name (input will never specify two creatures with the same first initial existing at the same time, nor will any names begin with 'W'). There are two special cases. If a Juggernaut is inside a wall cell, print its first initial in lower-case (names will always be capitalized in the input). If a Wampyr and a Human are in the same cell, print the first initial of the Wampyr in lower-case.

Legend: in the commands above:

<name>	a string, which may contain any character except a newline or tab, and which may not be one of the keywords (in boldface) used in the command descriptions above
<location>	a pair of integers formatted as (r, c)
<energy>	a positive integer

You may assume that the input file will conform to the given syntax. It is certainly possible that commands may name nonexistent creatures, request the creation of an excessive number of Humans and/or MazeMonsters, or specify locations that lie outside the maze boundaries. Your program should be prepared to deal gracefully with any of those situations.

A sample input file will be posted shortly.

Output description and sample:

All output for the program is to be written to standard output (`cout`), but this should be easily reconfigurable. All specified output should be produced, and any command that cannot be successfully processed should produce a useful error message. In addition, each command must be echoed, preceding any output generated for it. Finally, the echoed commands must be numbered, starting at 1, to make it easier to compare output files.

Sample output will be posted shortly. The format does not have to be exactly identical, but should be neat, and readable.

Design Homework Problem:

For this project you will produce an interim design document, which you will submit as a homework assignment.

The submitted design should only address the issue of the inheritance relationships among the various monster types. Draw an inheritance tree, indicating how the classes are related, and supply a class form for each class (including abstract base classes, if any). In the class form, indicate clearly which member functions will be virtual.

Since the assignment of responsibilities to each class was a particular problem for many of you on the previous project, be sure you make a careful analysis of that now. This may require some changes from your earlier project design and implementation. Remember that each class is modeling the abstraction of some “real” thing, and that the class should have the responsibilities that would ordinarily belong to that thing and that need to be modeled.

You must submit this design by 12:00 midnight on Thursday, April 13. Your design will be graded; discussions of reasonable design alternatives will be held in class during the week of April 10, and you may certainly apply those discussions to your design. Submit your design either as a PDF file or an MS Word document. The class diagram should be represented using the notations presented in the course notes and earlier project specifications. The class forms should follow the template given below.

Name:	
Purpose:	
Constructors:	
Operations: <i>Mutators:</i>	
<i>Reporters:</i>	
<i>Other:</i>	
Data Members:	

Submit your file (not zipped) to the Curator System as P4Design at: <http://spasm.cs.vt.edu:8080/curator/>

Evaluation:

Your score on this project may be based on several factors:

- Runtime testing of your program, and correctness and completeness of output.
- Quality of your final design, and conformance of your implementation to the design representation.
- Adherence to good software engineering practice.
- Quality of internal documentation.

The relative weighting of these factors will be decided later, but proper execution will be worth approximately 50% and the quality of internal documentation will be considered. To receive partial credit for programs that are non-working, or are not fully functional, a brief one or two paragraph description of the problem(s) must be included in the assignment submission, in an ASCII text file named `problems.txt`. The location of the problem, minimally identified to a specific function or functions, must also be specified along with possible corrections that need to be made.

Design Requirements

The major new concepts that are embodied in this project are inheritance and polymorphism. Your design and implementation should illustrate good use of each. The MazeMonster varieties should be related in a single inheritance hierarchy. That hierarchy should be designed and implemented to trigger polymorphic behavior when appropriate.

To make that requirement concrete, the following requirements are imposed:

- The controller entity that has responsibility for controlling the MazeMonsters must access the monsters via pointers.
- Those pointers must be of the base type (from the inheritance hierarchy).
- The controller entity must not explicitly store the type of each monster in any fashion, nor may the monsters be “segregated” by type.

As an example of what we’re after here, if a `jump` command is being processed, and the named creature is a monster of some sort, then that creature should be “told” to carry out a jump and it should respond correctly, which would mean that it should not do anything at all unless it is a Teleporter.

Also take care to correctly establish and maintain associations between objects. Associations should be continuous (via a stored pointer) rather than reestablished whenever member functions are called. It would be appropriate, but not required, to aggregate game objects within a controller object or function.

Programming Standards:

We will be evaluating your source code on this assignment for the quality of internal documentation and programming style, so you should observe good practice. The specifications from projects 1 through 3 apply. In addition:

Coding:

- Use classes where appropriate.
- Avoid the use of public data members in your classes.
- Divide each of your class implementations into a pair of matching header (`.h`) and source (`.cpp`) files, named to correspond to the class.
- Use explicit association where appropriate, managed by the passing and/or storage of class pointers.
- Use aggregation where appropriate.
- Design your classes so that each has a well-defined and reasonable set of responsibilities.
- Since there is no constraint on its dimensions, use a dynamic data structure to represent the maze grid.

Project Deliverables:

Your final project submission must include the following:

- All source code (*.cpp and *.h files) comprising your project.
- MS Visual C++ project files (dsp and dsw) or Unix makefile, as appropriate. (**VC++ users:** do not submit unnecessary files, such as ncb, opt, ilk, obj, pch, or pdb files).
- Your updated final design, including a class diagram showing association, aggregation and inheritance relationships, and class forms. This should be an MS Word document or PDF file as described in the preceding section.
- One input file and the corresponding output from your testing.
- A brief ASCII text readme file, named `readme.txt`, containing:
 - Your name and e-mail address, and the name of the project.
 - Your instructor's name, and the section of the course you're enrolled in (example: 8:00 MWF).
 - The platform and compiler you're using (example: MSVC++ 6.0 under NT).
 - Any special execution instructions. (If your program is not fully functional, be sure to mention that here and to follow the directions in the Evaluation section above.)
- An ASCII text file, named `pledge.txt`, containing the Honor Code Pledge listed at the end of this file.

Your project submission will consist of a zipped archive file containing all of the items specified above.

You are allowed to submit your solution up to five times, in case you detect (or solve) problems after your first submission. Your last submission will be the only one tested and graded. Note that, due to late penalties, it is possible that a fixed but late submission may receive a lower score than a faulty but on-time submission.

You will submit this zip file to the Curator System (as Project4), and you will demonstrate its operation to the GTA assigned to your section. In order to receive any credit for this assignment, you are required to meet with the GTA, even if your program does not compile and run.

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header preceding your `main()` function:

```
//      On my honor:
//
//      - I have not discussed the C++ language code in my program with
//        anyone other than my instructor or the teaching assistants
//        assigned to this course.
//
//      - I have not used C++ language code obtained from another student,
//        or any other unauthorized source, either modified or unmodified.
//
//      - If any C++ language code or documentation used in my program
//        was obtained from another source, such as a text book or course
//        notes, that has been clearly noted with a proper citation in
//        the comments of my program.
//
//      - I have not designed this program in such a way as to defeat or
//        interfere with the normal operation of the Curator System.
```

Taxi-Cab Metric

The taxi-cab metric computes the shortest distance between two grid cells if movement is restricted to the cardinal directions. Very simply, given two cells A and B, the taxi-cab distance between them is computed by moving horizontally from A to the column in which B occurs, and then vertically in that column until B is reached, counting cell transitions as you go.

For example,

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
1			■	■					■	■	■	■	■	■	■	■
2	■		■			B	■					C				■
3	■		■		■	■	■		■	■	■	■	■		■	■
4	■															
5	■		■	■	■	■	■		■	■	■		■		■	■
6	■						■				■		■		■	■
7	■	■		A							■				■	■
8	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

The taxi-cab distance from A to B is 7 (2 over and 5 up) and the taxi-cab distance from C to A is 13 (5 down and 8 over).