The Game

Design and implement a game of Sokoban++. The object of the game is to order the workers around a nondescript cave to push all the treasure chests to a specified exit location. The cave has a flat floor and is filled with various boxes and crates along the treasure chests. The cave is represented as a rectangular board divided into $M \times N$ ($M, N \geq 5$) square fields. The board is surrounded by cave walls.

Some of the crates are too heavy to be moved (but some of them might be blown up with explosives). Other crates are mounted on wheeled platforms and any force applied to them causes them to move continuously unless stopped on the nearest obstacle. Unfortunately a worker cannot be such an obstacle as it is too dangerous for his health. Another type of crates are very fragile and cannot be moved by pushing, only by picking them up and putting down delicately.

The Player has at his (or hers) disposal a number of workers placed in the cave. Some of them can only push the crates around, others can only blow the crates up. Another type of worker can only move crates by picking them up and placing on the neighbouring field. On each field there can be at most one worker or at most one crate (here we treat the treasure chests as crates). Finally there are universal workers who can perform all of the above actions but are easily becoming tired – they can only perform a limited number of moves and actions. Afterwards they become useless and unable to move. The workers can move only vertically and horizontally and can step into free squares only. They can only push the crates in the direction corresponding to direction of their moves and can only be pushed on free squares (so you cannot push two crates in one move). The crates can’t be pulled.

One turn of the game consists of choosing a worker and issuing a single (legal) command, which the worker obeys. The command can be either a move command, a push command (which implies moving of the worker) or an explode command (for the workers who possess sticks of dynamite). The treasure chests pushed on the exit location disappear, but others do not. The game ends when all the treasure chests are moved into exit location (win) or after reaching a predefined number of moves (lose). The exit location is not necessarily one of the border squares on the board. The treasure chests can’t be blown up by explosives.

The game starts with reading a specified text file with description of the board (the format is described below) and proceeds into commencing the game itself. The program should verify basic correctness of supplied data – that does not mean that we’ll try to trip your programs on some particularly devious and obviously incorrect board configurations but mistakes happen and you should detect situations like two objects placed in the same square of the board.

During the game you should display the number of already performed moves and the limit of moves on the board. There should be an option for undoing the moves already performed – preferably without storing all the previous board configurations but rather a sequence of actions taken and „unwinding” them. You will need to devise a convenient interface for choosing the workers and issuing commands – the interface can be entirely console and keyboard-based and can also allow for using a pointing device (e.g. mouse). You may find the ncurses library useful for providing good-looking console interface without too much hassle. We encourage you to at least use the color printing functions for your projects, as it enourmously enhances the gaming experience. Documentation, tutorials and code samples can be found on many websites. A few of them:

- http://invisible-island.net/ncurses/ncurses-intro.html
- http://tldp.org/HOWTO/NCURSES-Programming-HOWTO
For the very ambitious there’s a full fledged GUI library called Qt. You can find many examples, tutorials and a complete documentation at http://qt-project.org/doc. For any questions regarding technical aspects of using the ncurses or Qt libraries send emails to szreder [at] mimuw (dot) edu (dot) pl.

You are not required required to use any of these libraries to score a maximum amount of points. The standard C++ library is sufficient.

Board format

The board description starts with the integers $M$, $N$ and $T$ on the first line. The $T$ stands for the turn limit for the board. Each of the next $M$ lines contain a sequence of $N$ characters # (denoting an immovable and indestructible crate), . (denoting normal cave floor) and X (denoting the exit location). Next line contains a single integer $K$ denoting the number of objects (workers and crates) on the board. Each object is described in a separate line in this format:

- Name of the object.
- A single colon (:).
- Opening brace ({).
- List of attributes separated with a comma (,).
- Closing brace }.

Each attribute is described using the notation name=value. The attributes need not be in any specified order.

A list of standard objects and attributes follows:

- The standard objects are treasure, crate, heavy_crate, fragile_crate, wheeled_crate, worker, sapper, lifter, omni.
- Each object has a position described by row (from 1 to $M$) and column (from 1 to $N$).
- Sappers have explosives and can blow up each type of crate, except for the indestructible ones and the treasure chests. The heavy crates cannot be moved in any way but can be blown up (though it may be not necessary to do so for the completion of a particular puzzle).
- Lifters can move fragile crates. Omni are the universal workers.
- Sappers and omnis have a limited number of dynamite sticks (dynamite attribute).
- Omnis have limited energy. Each action (a move or using dynamite) consumes one point of energy.

Example

5 8 50
###.###
##.X....#
.......##
##...##
##....#

5
heavy_crate : { row = 3, column = 4 }
heavy_crate : { column = 5, row = 3 }
treasure : { row = 4, column = 5 }
sapper : { row = 1, column = 4, dynamite = 2 }
worker : { row = 3, column = 1 }
Extensions

You are welcome to introduce extensions not specified in the standard rules of the game. The extensions can span various aspects of the game: you can add new object types, alter the basic rules or introduce new rules etc. Be creative. We encourage you to share your ideas on extensions and implement them freely in your projects. You can use the discussion board on Moodle to share your ideas and receive comments.

The extensions should be implemented in the way not interfering with programs not supporting them. For example, let’s say that we have extensions called A, B and C. If your program supports only extension B then it should ignore all the other extensions, but still be able to read the supported extension and use it in the game. If you support extensions in your program, you should add an option to disable them, either selectively or entirely.

When you devise an extension, you should post the information regarding its name and format on the Moodle, so that others can implement them at will and (more importantly) not use the same name for their extensions.

We propose the following format for specifying extensions:

- Extension description is started with the string `Extension start NAME`.
- Then the contents of the extensions are presented in the following lines, according to a relevant format description for a specific extension.
- The description is ended with the string `Extension end NAME`.

This should allow to easily bypass not supported extensions.

**Example extension: Demolition** This is a simple extension altering the rules of the game. It allows the sappers to destroy previously indestructible crates at the expense of using $D$ sticks of dynamite. The only content of the extension is the integer $D$.

```
Extension start Demolition
3
Extension end Demolition
```

Grading

The basis of your grade will be the correctness and „object-orientedness” of the program. Your implementation isn’t required to use any of the suggested software libraries for defining the user interface, though you may or may not find using them actually decreasing the amount of work on the project. Your programs should also carefully manage memory and other resources (like open files) and not leak them.

The basic testing environment will be the students machine. For testing purposes you can access it from home using the ssh on Linux or the PuTTY software on Windows. The hostname is `students.mimuw.edu.pl`. Notice that if you want to use the Qt software library, you are most likely going to create a cross-platform project, i.e. you will be able to compile and run the program on Windows, Linux and MacOSX systems.

The Contest

There will be a forum open on the Moodle platform, where students will have the possibility for storing, downloading and grading sample boards. The three best boards will be awarded with 3, 2 and 1 extra point respectively. In case of a larger number of boards with the same highest score we reserve the right of arbitrary choosing the three awarded ones.