C++

Thierry Géraud, Philipp Schlehuber
theo@lrde.epita.fr; philipp@lrde.epita.fr

2020

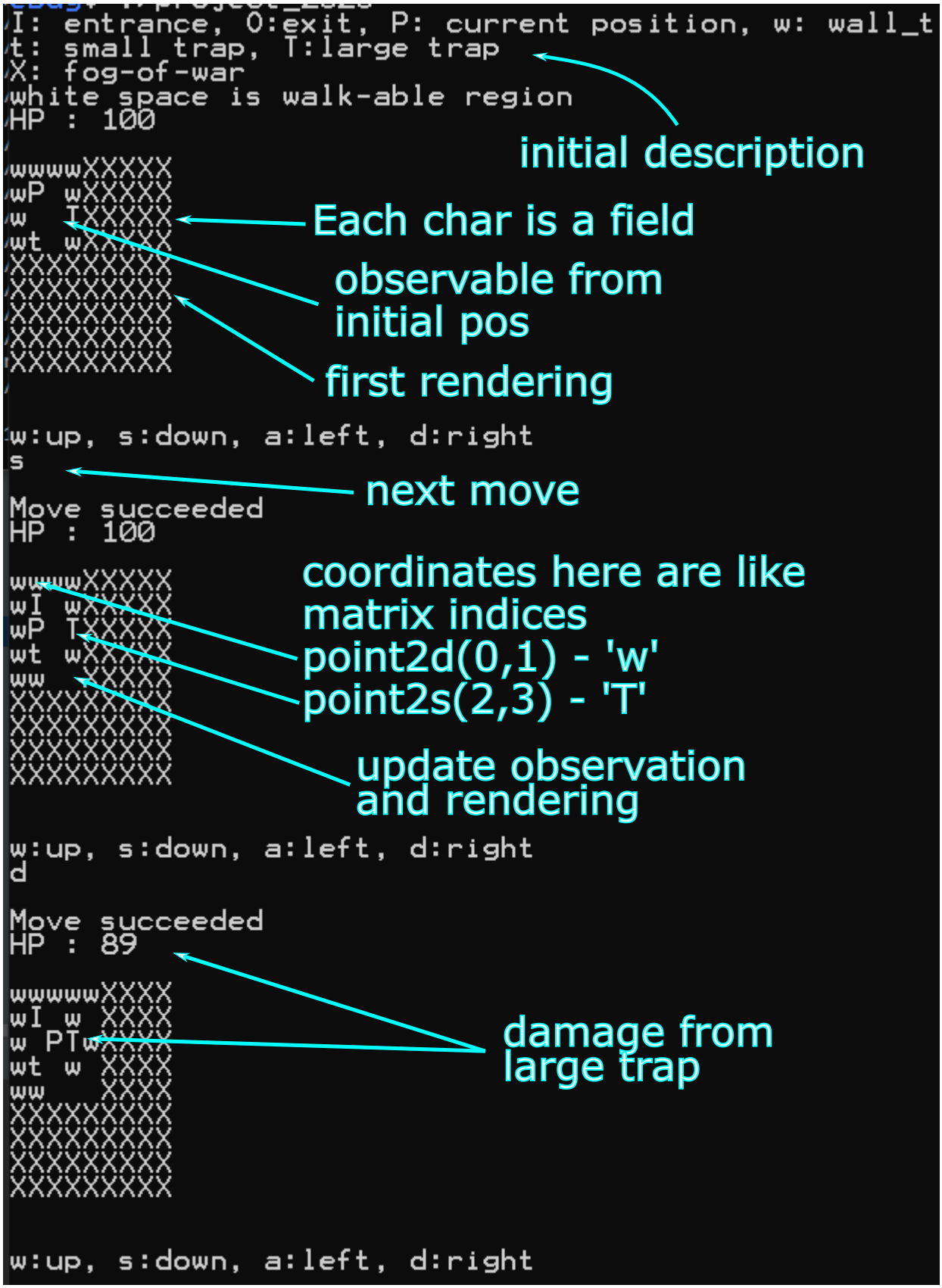
Project: A maze game

In this project you will develop a program that simulates an escape game from a maze. In the first part of the project you will consider this problem from the angle of game developpement. You will have to define the different classes needed for the game and program its main loop which interacts with the player. In the second part you will view it from an AI-perspective: You will develop an algorithm which can escape the maze on its own.

The pseudo-code and the list of classes below gives you the big picture of what your code should do and which classes are involved in the first part of the project:

```
1: procedure Game Main
2:     M ← Load maze from string
3:     P ← Define the player
4:     G ← Game defined by M and P
5:     while Neither WON nor DEAD do // Game Loop
6:         P observes M from current position
7:         MOV ← P decides on next move
8:         EFF ← M computes the effects of MOV
9:         Apply EFF onto P
10:    end while
11: end procedure
```

A typical game output should look like this



Even-though this seems straight forward, a lot of different concepts, types and classes hide behind this pseudo code, the most important ones are:

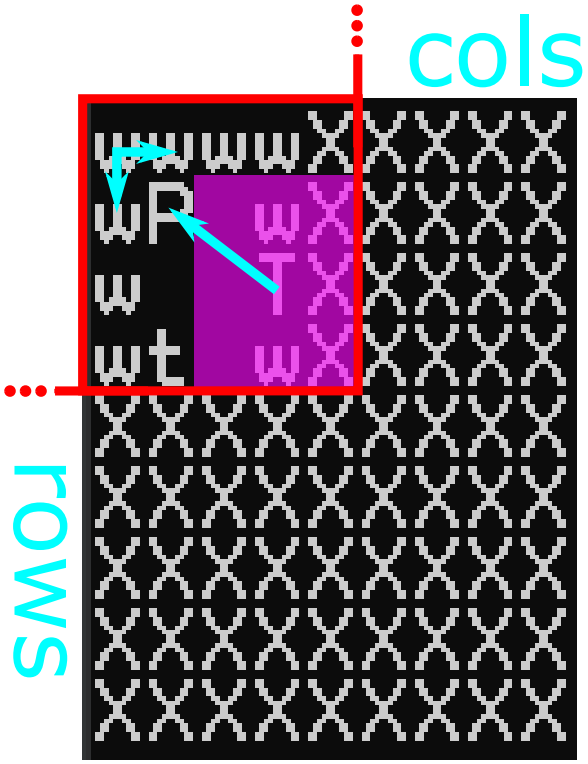
struct point2d Defines a point in the grid, mainly a pair of coordinates.

struct offset2d Defines the difference between two points. Note that you should define the basic operations of addition and substraction between points and offsets.

class player Define a base class for player, defining the (virtual) functions of processing maze observations and computing a desired move (an offset) based on the current state of the game. From this base class two specialization will be derived, defining a manual and an AI player.

class fields Defines the behaviour of fields composing the maze. Fields need to be convertible to chars, to represent the field in the output. Moreover, we have to be able to compute the effect of a field. The effect of a field depends on the relative offset between the position of interest and the field. For instance, the field defining the exit field_exit_t returns the effect "EXIT" iff the offset is zero, indicating the player is at the same position. For further explanation see the figure below. So as you can imagine, field_exit_t is a class derived from fields_t. This concept can be extend and you will at least need to define walls, paths, entrance, small traps, large traps and hidden traps:

- walls: Are blocking, the player can not enter this field.
- paths: The player can go there with no additional effect.
- entrance: Serves as initial position.
- exit: If attained while being alive, the player wins.
- small trap: If the player steps on a small trap, he dies. Has otherwise no influence.
- large trap: If the player is within a square of side length 3 around the trap, the player loses between 5 and 15 hit points (See offset figure).
- hidden trap: (Bonus field) This trap is visualized as a path until it deals damage to the player for the first time. From there on, it behaves like a regular large trap.



The absolute position of the player P is point2d(1,1). The large trap T is located at point2d(2,3). The offset from the trap to the player is point2d(1,1)-point2d(2,3) = offset2d(-1, -2). The region affected by the large trap is shown in magenta. The player is outside of it, so he does not take damage.
The region observed by the player (for his current position) is shown as a red box. It is the intersection of a square box with side length 5 centered on the player (indicated by the dotted extension) and the maze itself. Your iterators have to be such that: repeating first dereferencing (operator*) then incrementation (operator++) 16 times on the first iterator, all currently shown fields have to appear once and the (incremented) first must equal the second iterator (the incremented iterator equals the end iterator indicating we have exhausted the implicit list).

enum class field_state This enum class is used to enumerate the different states a position can have for the player. We will also use the enum to declare an order or importance between them. This will be useful to safely accumulate effects of fields in a single object.
The states are *NONE*, *FREE*, *DAMAGING*, *ENTRANCE*, *EXIT*, *DEADLY*, *BLOCKED*.
There is an order between these elements with respect to the gameplay. For instance, given the position is within the zone of influence of a large trap: This causes the player to recieve damage, so the associated state should be *DAMAGING*. However if this position coincides with the location of a small trap, the state *DEADLY* should take precedence over *DAMAGING*.

struct field_effect Struct used to regroup the damage accumulated and the field_state of the highest precedence encountered. Define `field_effect& operator+=(const field_effect& other)` to faciliate usage of this struct later on (See Part 2.1).

class maze The maze holds a square matrix of shared_ptrs to the different fields composing it. A maze has to be observable from a given point, as we want to explore the maze during the game and not simply show the entire maze. This function should return two maze_iterators, corresponding to the beginning and the end of the observable region. To make the game more realistics, a player can only observe fields that are within a square box of side length 5 centered on the player (See offset figure).
Moreover, it must provide a function to evaluate the effects of all fields onto a given position in the maze. For instance, given the position coincides with the position of a small trap, then no matter what the effect of other fields are, the player will die.

Utilities: String to maze Creates a maze from a string. Remember mazes are always square matrices, the string will not contain a special character to denote the start of a new row, instead it is just the concatenation of all rows. Each field has its own character: entrance 'I', exit 'O', walls 'w', paths ' ', small traps 't', large traps 'T' and hidden traps 'H'.

Utilities: Iterators At multiple instances throughout the code it is convenient to work with iterators defined over (parts of) the maze, as they allow to interface with several algorithms from the STL. You will define iterators over boxes, mazes and neighbours. As this part is sometimes a bit technical, parts of the code will be provided and automatically evaluated as moodle questions.

Part 1 - Coordinates Define the struct point2d and offset2d such that they pass all tests on the moodle. This should ensure that your implementation is suitable for the project.

Part 2.0 - Fields Given the abstract base class (which you can modify if you find it necessary) define the concrete derived classes for the fields entrance, exit, path, walls, small trap, large trap and hidden map.

Part 2.1 - field_effect Define the operator `field_effect& operator+=(const field_effect& other)` such that when summing effects the following holds:

- The damages received from multiple traps is cumulative.
- If the position is blocked, no move is possible and no damage will be received
- Entrance and Exit implies that that the position is free, but takes precedence over it
- The player can only exit the maze (and win the game) if he is still alive after receiving the accumulated damage.
- Immediate death is correctly reported independently of the accumulated damage

Part 3 - Maze storage and Iterators In order to increase the modularity of your code, derive the class maze from a generic class called storage. This class represents a square matrix holding arbitrary objects of type T, with T being a template. We call the number of rows / columns the size of the matrix
To ease access onto this storage, we want to define different iterators. The most basic iterator is called box_iterator: It is defined by four values i_m, i_M, j_m and j_M and returns all point2d's that are inside of the box, so the set $\{point2d(i,j)|i_m \leq i < i_M, j_m \leq j < j_M\}$.
Secondly we want to define a maze_iterator. A maze_iterator derives from the box_iterator, but has additional restrictions:
It is used to access the maze and not only the point, so it is dereferenced to a pair of point2d and the corresponding shared_ptr to the field.
It is restricted to points that are inside the box AND the maze.
Finally we want to define a neighbour_iterator. Given an initial point2d it iterates over all points reachable in one step from the initial step. That is the point above, below, left and right. This iterator can derive from box_iterator and does not have to respect the maze dimensions, as it has no direct connection to it.
Your implementation should pass all moodle tests.

Part 4 - Manual Player Implement the player base class and the manual player. The player needs to do three things each round:

- Observe : Treat the observations obtained from the maze. The observations are given as a pair of maze_iterators, with the first one being the beginning and the second one being the end iterator.
- Act : Translate user input into a valid, desired movement.
- State : To facilitate rendering, one must be able to query the player whether or not he has already observed a certain point.

Part 5 - Maze, Game and Player interaction The game class contains all participants of the game and executes its main loop. When everything has been implemented correctly, there are only very few interactions between the main classes of the game in each round.
As it can be seen in the pseudo-code, each round consists of the following main steps. The player observing the maze. The game rendering the maze querying the player which points had been observed. The player deciding on its next move. The maze evaluating the effects of the desired move and whether it succeeds. The game checking if the player has won or is dead.
With these interactions properly defined you should have a playable game.

Part 6 - AI player This is the final part of the project, in which you are asked to design an algorithm capable of finding the maze exit. The only class to implement for this, is the specialization of the base player, with the rest of the code remaining unchanged. If your algorithm is capable of solving this problem you have in fact written a reactive control system that "knows" abouts its limitations and reacts to new observations and a changing environnement. This is a non-trivial problem.
In case you need some inspiration on how to tackle this problem: You could consider the maze to be a graph, with each field being a vertex. There are edges between the vertices iff the corresponding field have a manhattan distance of exactly 1 and the destination field is not a wall. Care has to be taken to avoid traps.
Then you could apply graph techniques to solve the (approximated) problem.

1 Organisation

1.1 Compilation

Your project has to compile with one of the following commands:

1. make
2. mkdir build; cd build; cmake ..; make
3. for i in *.cc; g++ -pedantic -Wall -Wextra -std=c++2a $i -c; g++ *.o

For option 1 and 2, make sure you set the flags "-pedantic -Wall -Wextra -std=c++2a".
Any project that does not compile for whatever reason (on my machine with ubuntu 20.04 and g++ version 9.3) **can not get more than 9 points**. If you submit your project at least 24h before the deadline you can ask me via mail if any build errors occured.

1.2 Content

You do not have to write a report for this project. A simple README and properly formatted and (where necessary) documented code is sufficient.
The README has to contain which Parts you have fully/partly implemented. If your project generates more than one executable, explain the purpose of each executable.
For each part of the project that you have fully implemented, your executable has to generate some output (on std::cout) that documents this.

1.3 Deadline

The project has to be uploaded to the moodle page. The deadline (31.01.2021 23:59h) is strict and automatically enforced.

Appendices

Containers, Algorithms and Iterators

During the course you have seen several containers provided by the STL (like std::vector, std::list etc). What you have not seen so far are the different algorithms provided by the STL (like for instance the sortign algorithm implemented in std::sort) and how to interface them with the containers (the infamous iterators).
Iterators provide an unified handle on containers:

- You can conceptualize iterators as "enhanced pointers" to elements in arrays
- Every container has a beginning (obtained via .begin()) and an end (obtained via .end()). Note that end() is the first element AFTER the container, so it cannot be dereferenced.
- Iterators over a container with element type T can be dereferenced to obtain a reference to object of type T in the container using the usual "star" notation: `T& my_obj_copy = *it;`
  Or call member functions of T like: `it->my_fun();`
- If iterators can be incremented (`++it;`) they are called ForwardIterators. Like iterators over `std::forward_list`, `std::list` or `std::vector`.
- If iterators can be incremented (`--it;`) they are called BackwardIterators. Like iterators over `std::list` or `std::vector`, but not `std::forward_list`.
- If iterators can be made to point to arbitrary elements of the container (`T& fourth_elem = it[3];`) they are called RandomAccessIterator[1]. Like iterators over `std::vector`, but not `std::forward_list` or `std::list`.

[1] Technically it is also demanded that the cost of accessing a random element can also be done in constant time.

- Different algorithms need different (minimal) types of iterators: std::any_of needs (at least) a ForwardIterator, whereas std::sort needs a RandomAcccessIterator.

The main reason to care about iterators is because they make code more readable and the are used as an interface to the great many algorithms provided by the STL. This is something you should really care about as these algorithms are thoroughly tested, almost certainly more efficient than your own code and evolve overtime when better algorithms are found for the same problem. But let's take a quick look at a hands-on example: Let's say you have a vector of ints and you want to sort it. With STL this is as easy as it gets (It's almost pythonish)

```
#include <vector>
#include <algorithm>

int main()
{
  auto v = std::vector<int>{4,6,2,3,1}; // Vector of ints
  std::sort(v.begin(), v.end()); // Sort from beginning to end -> the whole vector
  // v is now sorted in ascending order
}
```

Or you could write and debug your preferred sorting algorithm, your call...
Another common question is: But in my application I do not need to sort int or float but my own class "some_class_t", so I can't take advantage of the STL algorithms. Worry not, your are not the first developper to encounter this problem and C++ got you covered. std::sort is overloaded to take a third argument called "comp" which has to be a comparison function object.
What does this mean in practice? The passed object has to be callable and define a strict total order (the "<"-operation) between any two instances of your object. Or in practice:

```
#include <algorithm>
#include <string>
#include <vector>

// Your class holding an int and a string
class some_class_t
{
public:
  some_class_t() = default;
  some_class_t(int a, std::string s) = default;
  some_class_t(const some_class_t& o) = default;
  some_class_t(const some_class_t&& o) = default;
  some_class_t& operator=(const some_class_t& o) = default;
  some_class_t& operator=(const some_class_t&& o) = default;

  int a() const
  {
    return a_;
  }
  const std::string& s() const
  {
    return s_
  }

private:
  int a_;
```

```
  std::string s_;
};

// Define three different sensible ways to compare them
class comp_a_t
{
  bool operator()(const some_class_t& c1, const some_class_t& c2) const
  {
  // Ascending with respect to the integer
    return c1.a() < c2.a();
  }
};

class comp_s_t
{
  bool operator()(const some_class_t& c1, const some_class_t& c2) const
  {
    // reverse lexographical order with respect to the string
    return c1.s() > c2.s();
  }
};

class comp_l_t
{
  bool operator()(const some_class_t& c1, const some_class_t& c2) const
  {
    // lexographical order over both members
    // With the integer having priority over the string
    return std::tie(c1.a(), c1.s()) < std::tie(c2.a(),c2.s());
  }
};

int main()
{
  std::vector<some_class_t> v;
  v.emplace_back(3, "ab");
  v.emplace_back(3, "abc");
  v.emplace_back(1, "b");

  std::sort(v.begin(), v.end(), comp_a_t());
  // Gives [(1,"b"), (3, "ab"), (3, "abc)] or
  // [(1,"b"), (3, "abc), (3, "ab")]
  std::sort(v.begin(), v.end(), comp_s_t());
  // Gives [(3, "abc), (3, "ab"), (1,"b")]
  std::sort(v.begin(), v.end(), comp_s_t());
  // Gives [(1,"b"), (3, "ab"), (3, "abc")]

  return 0;
}
```

You can even reduce the amount of boiler-plate code by relying on lambdas: Instead of defining the comparison function object, just define a lambda directly where needed:
`std::sort(v.begin(), v.end(), comp_a_t());`
becomes

```
std::sort(v.begin(), v.end(), [](const auto& c1, const auto& c2){return c1.a() < c2.a();});
```

STL algorithm does however not only provide sort, but solutions to a lot of reoccuring problems. For instance we have `std::any_of`, `std::all_of`, std::find ...
The goal is to reduce the lines of code necessary while increasing readability, which often goes hand in hand with avoiding "raw" loops.
Consider the following example with v being a vector of bools:

```
//Version 1
bool res = true;
for (size_t i = 0; i < v.size(); ++i)
  if (not v[i])
  {
    res = false;
    break;
  }
//Version 2
bool res = true;
for (auto b : v)
  if (not b)
  {
    res = false;
    break;
  }
// Version 3
auto res = std::all_of(b.begin(), b.end());
```

While Version 2 is clearly preferable over Version 1 as the auxiliary variable i is avoided, Version 3 is even better as it unambiguously states the purpose.