

## Requisiti labirinto:

```
* * * E * * * *  
* * * * * * *  
* * * * * * *  
* * * * * * *  
* S * * * * * *  
* * * * * * * *
```

- 1 sola partenza 'S'
- No labirinti impossibili  
(esempio: robot intrappolato)
- Almeno un'uscita 'E'  
(in teoria nessuno vieta che il labirinto sia circondato da uscite)
- $9 \times 9$  caselle
- i muri sono contrassegnati dal carattere '\*'

## Classe Maze:

Si occupa di importare il labirinto da un file, di controllare la validità del labirinto, della legalità degli spostamenti del robot e se esso sia arrivato alla fine

## Classe Robot:

Il robot conosce solo le 8 caselle adiacenti ad esso (quindi può anche sapere da sé se è arrivato alla fine, ma preferisco che sia Maze a dichiararlo).

È la classe base per due derivazioni di robot:

### • RandomRobot:

si muove in maniera completamente casuale

### • Right Hand Rule Robot:

si muove in modo casuale SE non ha pareti sulla destra, altrimenti si muove mantenendo SEMPRE il muro sulla destra.

## Maze :

- Maze () costruttore di default, crea un labirinto vuoto
- Maze (std::filebuf \*) costruttore che crea un labirinto a partire dal puntatore al buffer del file aperto, legge solo i primi 81 caratteri (= 9·9), escluso il carattere di newline '\n' ed effettuerà i vari controlli sul labirinto -
- void import\_maze (std::filebuf \*) importa il labirinto (sostanzialmente è la funzione che invoca il secondo costruttore)
- bool is\_maze\_valid () controlla che il labirinto sia valido (da implementare con i grafi per vedere se l'uscita è raggiungibile)
- const std::vector<char> get\_maze\_chunk () const ritorna le 8 caselle adiacenti al robot
- int move\_robot (int) prende come parametro la posizione relativa del robot, controlla se è una mossa valida, se non lo è ritorna (-1), se è valida ritorna (0), se la mossa è valida e il robot raggiunge l'uscita allora la funzione ritorna (1)
- std::vector<char> maze contiene le 81 caselle del labirinto
- int current\_robot\_position tiene aggiornata la posizione del robot nel labirinto, inizializzata a (-1)



privati



pubblici

## Robot :

- Robot () = delete non serve
- virtual void move (const Maze &) = 0 si occupa di "muovere" il robot nel labirinto, però in questa classe non viene implementata
- int gen\_rand\_value () genera un intero randomico

RandomRobot e RightHandRuleRobot fanno solo l'override di move()

Il labirinto lo implemento come un vettore unidimensionale

0 1 2 3 4 5 6 7 8

9 11 12

18

27

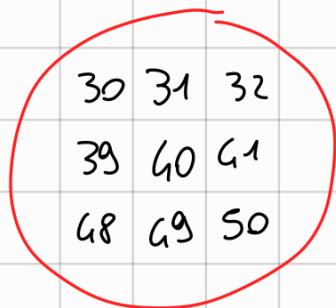
36

65

54

63

72 73 74 75 76 77 78 79 80



17

26

35

64

53

62

71

80

POSIZIONE RELATIVA

-10 -9 -8

-1 0 +1

+8 +9 +10

posizione  
corrente del  
robot



! Al robot non mostra la casella in posizione relativa (0) perché è lui stesso -

Casi limite :

0 1 2 3 4 5 6 7 8

9

18 Caselle da mostrare

27 28

POSIZIONE DEL ROBOT

36 37

65 66

17

26

35

64

53

62

71

80

Utilizzando le posizioni relative per ritornare le celle adiacenti, ritornerei pure 26, 35, 64 (in questo esempio)

che però sarebbe sbagliato

Penso notare che questa situazione (ovviamente) accade solo nei boroli; quindi a seconda del caso posso utilizzare delle proprietà comuni tra quelle posizioni -

### Caso 1:

Lato sinistro :

0	1	2	3	4	5	6	7	8
9								17
18	caselle da mordere							26
27	28							35
36	37							44
45	46							53
54								62
63								71
72	73	74	75	76	77	78	79	80

Sono tutti multipli di 9, quindi posso restituire al robot solo queste caselle

-9 -8

0 +1

+9 +10

### Caso 2:

Lato destro

Sono caselle del tipo  $(k \cdot 9) - 1$ , con  $k \in \{1, \dots, 9\}$ .  
posso restituire al robot solo queste caselle

-10 -9

-1 0

+8 +9

0	1	2	3	4	5	6	7	8
9								17
18	caselle da mordere		25	26				
27			34	35				
36			43	44				
45								53
54								62
63								71
72	73	74	75	76	77	78	79	80

### Caso 3:

caselle da mordere

Lato alto

Sono caselle di posizione  $k < 9$ .  
posso restituire al robot solo queste caselle

0	1	2	3	4	5	6	7	8
9			12	13	14			17
18	caselle da mordere							26
27								35
36								44
45								53
54								62
63								71
72	73	74	75	76	77	78	79	80

-1 0 +1

+8 +9 +10

Caso 4:

0	1	2	3	4	5	6	7	8
9							17	
18							26	
27							35	
36							44	
65							53	
54							62	
63	65	66	67				71	
72	73	74	75	76	77	78	79	80

Lato basso

Sono caselle di posizione  $71 < k < 81$

posso restituire al robot solo queste caselle

-10 -9 -8

-1 0 +1

POSIZIONE DEL ROBOT

Ci sono anche altri 6 casi, ma per farli basta fare l'unione di questi 6 casi appena visti, nelle giuste situazioni, per esempio

POSIZIONE DEL ROBOT

0	1	2	3	4	5	6	7	8
9	10						17	
							26	
18							35	
27							44	
36							53	
65							62	
54							71	
63							79	
72	73	74	75	76	77	78	79	80

Faccio l'unione del caso 1 e 3  
mostro quindi:

0 +1

+9 +10

Quindi quando viene invocata la funzione `get_maze_chunk()` calcolo le posizioni relative rispetto al robot, per mostrare le celle adiacenti ad esso.

Parto da un vettore che le contiene tutte

`std::vector<int> relative_position { -10, -9, -8, -1, 1, 8, 9, 10 }`

e vado a togliere quelle che non mi servono, quindi farò passare questo vettore tra `if` e `for` per rimuovere ciò che non mi serve.  
Alla fine utilizzerò il vettore scemato per ritornare le celle adiacenti al robot.

Allo stato attuale però il robot "conosce" le celle adiacenti ma NON sa quali siano, per esempio se il robot si trova in questa situazione :

Dal metodo get\_maze\_church() il robot attualmente riceve questo vettore : [ \*, vuoto, vuoto ]

Con vuoto intendo il carattere spazio  
( = 32 ASCII TABLE )

Pero'esso non SA che sono le  
celle in posizione relativa -9,-8,+1  
perche' otterrebbe lo stesso vettore se  
si trovarse in questa situazione :

A grid of 100 asterisks arranged in a 10x10 pattern on lined paper. The grid is bounded by vertical lines at each column and horizontal lines at each row. A green rectangular box highlights the cell at the top-right corner (row 1, column 10). A blue arrow originates from the bottom-left corner cell (row 10, column 1) and points towards the right edge of the grid.

In questa situazione invece le posizioni relative sono -1, 8, 9

Quindi per far capire al robot anche le posizioni relative, mantengo il formato  $\{-10, -9, -8, -1, 1, 8, 9, 10\}$  come l'ordine delle posizioni restituite rispetto al robot, le celle che prima non venivano ritornate, ora avranno come valore 0 (NULL char)

Considerando il secondo esempio fatto prima, allora il vettore ritorna =  
to zero' 0 0 0

$$\left[ \begin{matrix} 0, 0, 0, *, 0, \text{vuoto}, \text{vuoto}, 0 \\ -10, -9, -8, -1, 1, 8, 9, 10 \end{matrix} \right]$$

## Conosciuti

Nella classe base **Robot** bisogna quindi aggiungere un metodo che calcoli il vettore di posizioni relative utilizzato che chiamerò `const std::vector<int> calc_relative_vector()`

## Random Robot (movement)



Per come è stato strutturato il progetto, il `Robot` conosce fino ad un massimo di 8 celle, adiacenti ad esso, del labirinto.

Ovviamente non tutte saranno caselle in cui puo' andare, quindi quando genererà una mossa casuale bisogna controllare che non ci siano ostacoli ( comunque anche il labirinto controlla che sia un movimento valido ) -

Bisogna scemmare le celle su cui generare il movimento casuale per esempio :



■ celle in cui puo' muoversi  
■ celle in cui NON puo' muoversi

Quindi dopo essere venuti a conoscenza delle celle adiacenti e aver calcolato le posizioni relative, scemmo sempre dal vettore delle posizioni relative le celle a cui non posso andare, quindi considerando questo esempio il vettore delle posizioni relative scemato sarà :  $[-1, 1]$

Dopo generero' un numero randomico da 0 a  $m-1$  dove  $m$  è la lunghezza del vettore PR

PR = posizioni relative

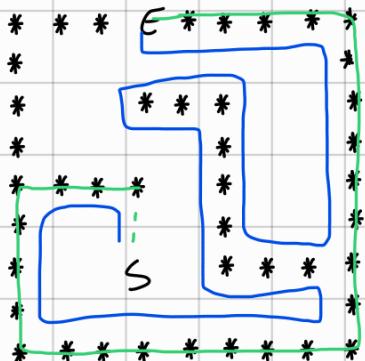
Per come è strutturata la classe Maze, ovvero che quando importa un labirinto effettua i dovuti controlli per verificare che sia risolvibile, il robot avrà SEMPRE una casella su cui andare

## RightHandRule (movement)

Se il robot non ha muri (ostacoli) nelle vicinanze si muoverà a caso (vale tutto quello che è stato detto per il RandomRobot)

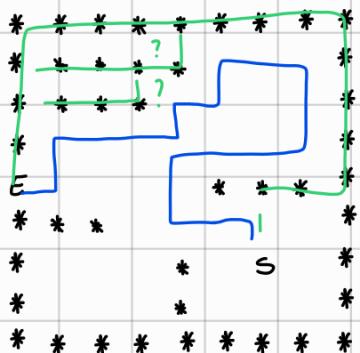
Se invece il robot si trova vicino ad uno ostacolo deve seguire la regola della mano destra e muoversi mantenendo il muro alla sua destra.

Questo introduce un ulteriore controllo sul labirinto perché sia valido, ovvero che l'uscita sia raggiungibile muovendosi solo verso destra lungo le pareti, quindi è come se si potesse raggiungere l'uscita camminando solo lungo le pareti



Da qui si può capire che un RHR Robot (RHR = right hand rule) ha bisogno di un labirinto che abbia l'uscita SOLO sul bordo (a quanto pare)

Caso limite di percorrere le pareti:



Cammino percorrendo lungo i muri:  
cammino da percorrere

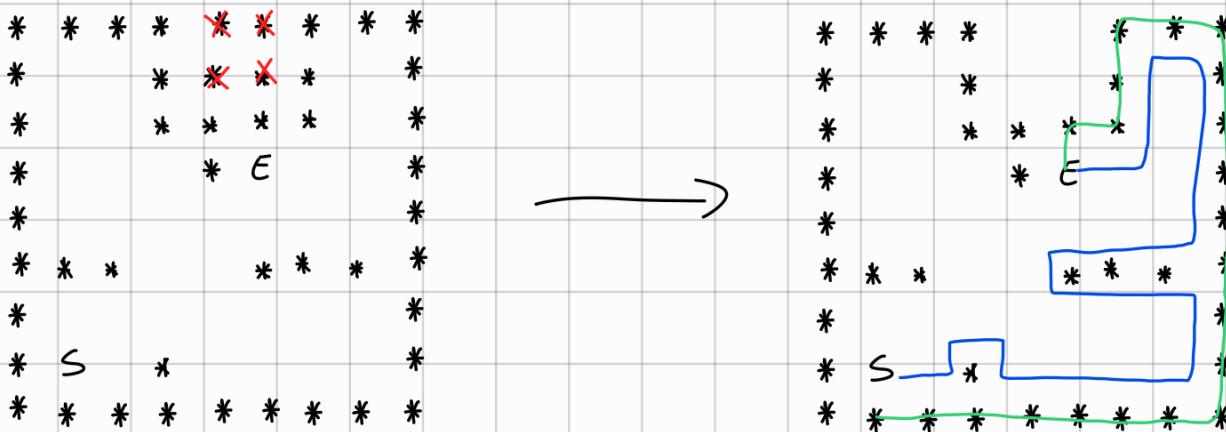
In questo caso si potrebbe avere più di un cammino lungo gli ostacoli -

Dall'ultimo esempio possiamo semplificare il caso limite lavorando su un labirinto semplificato (solo per verificare la raggiungibilità dell'usata) ovvero eliminando gli ostacoli che sono COMPLETAMENTE circondati da ostacoli.

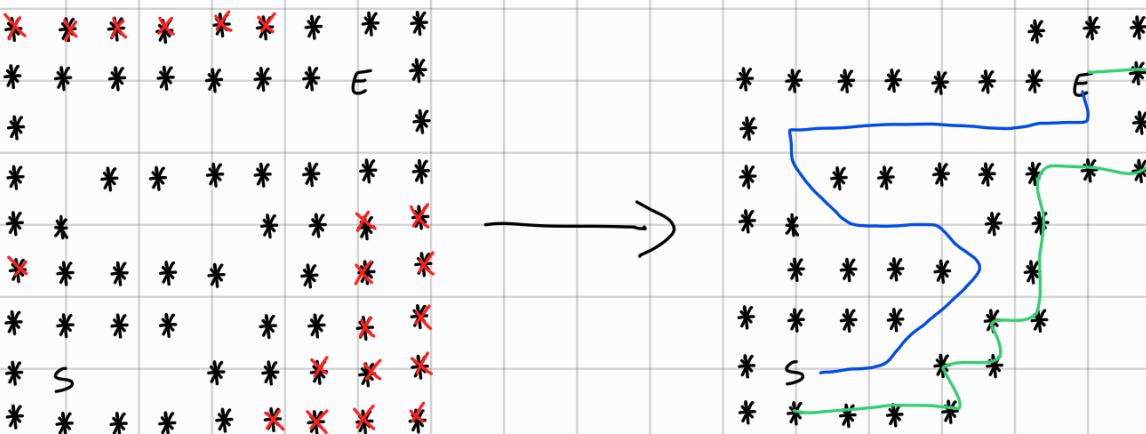
L'esempio di prima diventa:

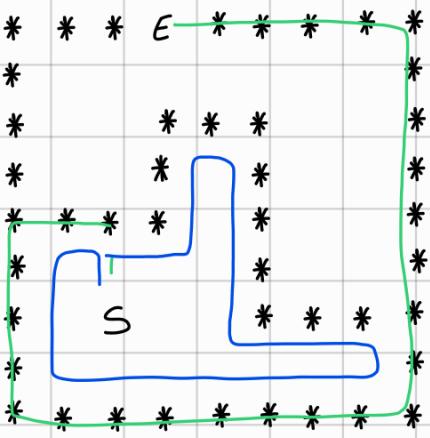


Ulteriore caso limite:



Ulteriore caso limite





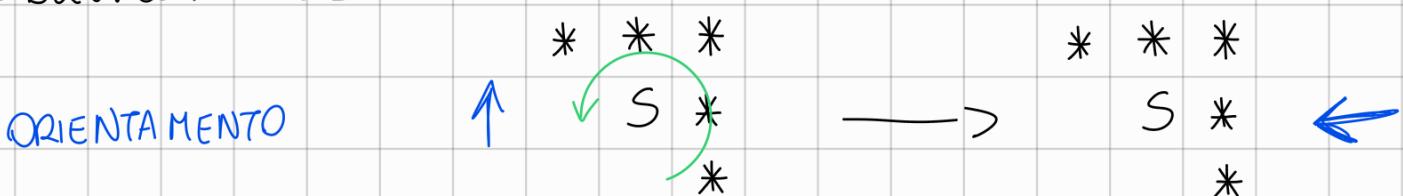
Non mi devo preoccupare di una situazione del genere perché prima di tutto controllo se l'uscita è raggiungibile e poi che rispetti i requisiti perché RTR Robot possa risolvere lo.

Quindi un labirinto del genere viene sicuramente scartato.

RTR Robot quando ha un ostacolo vicino percorre il labirinto tenendo le mura alla propria DÉSTRA -

Bisogna quindi definire la destra per il robot, ovvero deve avere un **orientazione** -

All'inizio il robot è orientato verso l'alto ↑, dopo controlla in senso antiorario ostacoli nelle vicinanze per cambiare l'orientamento -



Caso limite :



Per rispettare la regola delle mani destra e visto che il robot si può muovere in diagonale ritengo che sia corretto inserire orientamenti diagonali:

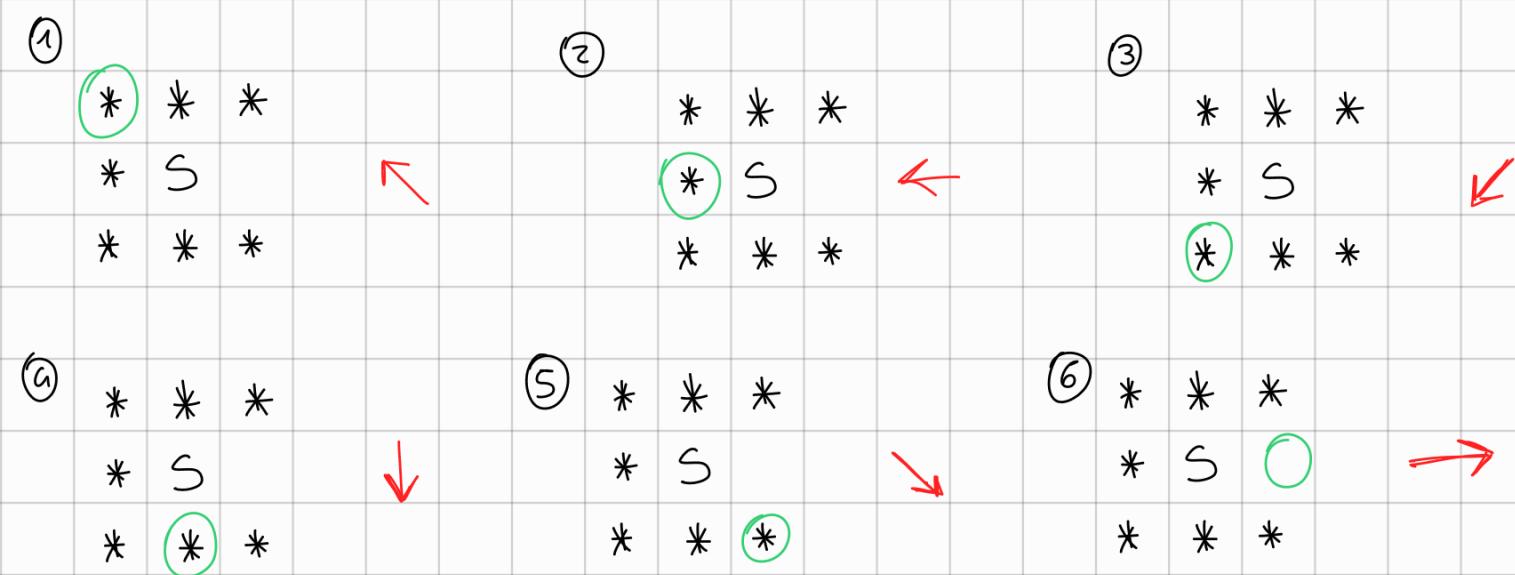
=> ORIENTAMENTO → in questo esempio

## Come controllare l'orientamento

Visto che di default è rivolto verso l'alto ( $\uparrow$ ) decido che oltre a indicare l'orientamento, rappresenta la casella da controllare per decidere l'orientamento.



Giuntamente se c'è un ostacolo passo alla casella successiva, in senso anti-orario, aggiornando l'orientamento



L'aggiornamento dell'orientamento finisce al 6<sup>a</sup> step, il processo si ferma appena trova una casella libera.

Questa funzione non è implementata tramite una funzione membro privata alla classe **RTR Robot** e ritornerà l'orientamento corretto del robot come valore intero seguendo questo schema:

1 0 7      valore iniziale di default

2 R 6

3 G 5      Per mantenere questo schema quando verrà invocata la funzione inizierà SEMPRE a controllare prima verso

l'alto e poi nelle altre direzioni -

Questa funzione verrà invocata quando il robot si deve muovere e solo quando RHR Robot ha un muo nelle vicinanze -

int robot\_orientation (const std::vector<char>& )

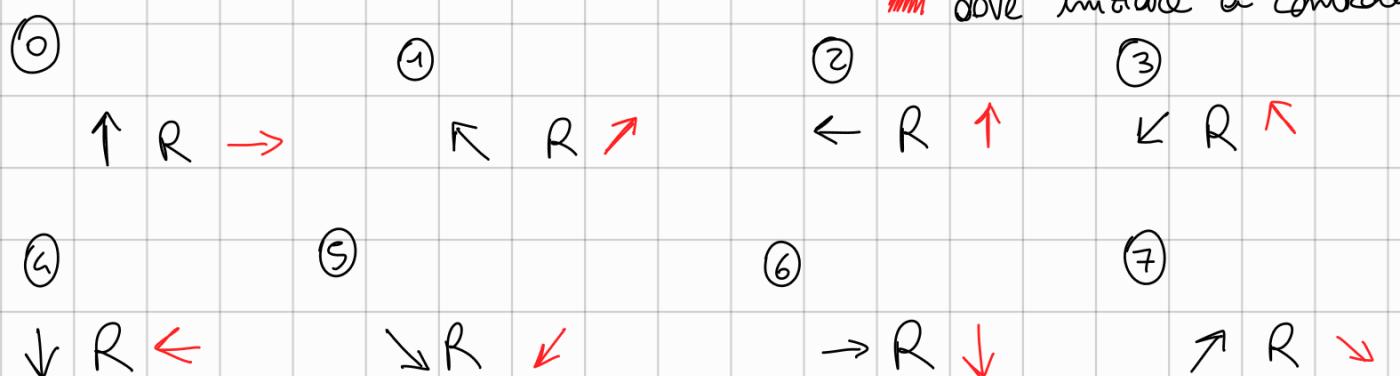
int current\_robot\_orientation definisce l'orientazione corrente del robot e di fatto dove iniziare a controllare per prima (il valore di default è 0 = up) -

Non va bene controllare SEMPRE verso l'alto in qualsiasi situazione, ma bisogna controllare in base all'orientamento corrente -

Quindi ci sono 8 casi :

■ orientamento corrente

■■■ dove iniziare a controllare



1 0 7

2 6

3 G 5

0 6

1 7

2 0

3 1

G 2

5 3

6 G

7 5

Ottengo che devo iniziare a controllare da  $(co + 6) \% 8$  dove co è l'orientamento corrente -

Pero per essere eseguito in un ciclo for diventa:

$$(co + i + 6) \% 8, \text{ con } i \in \{0, 1, 2, \dots, 7\}$$