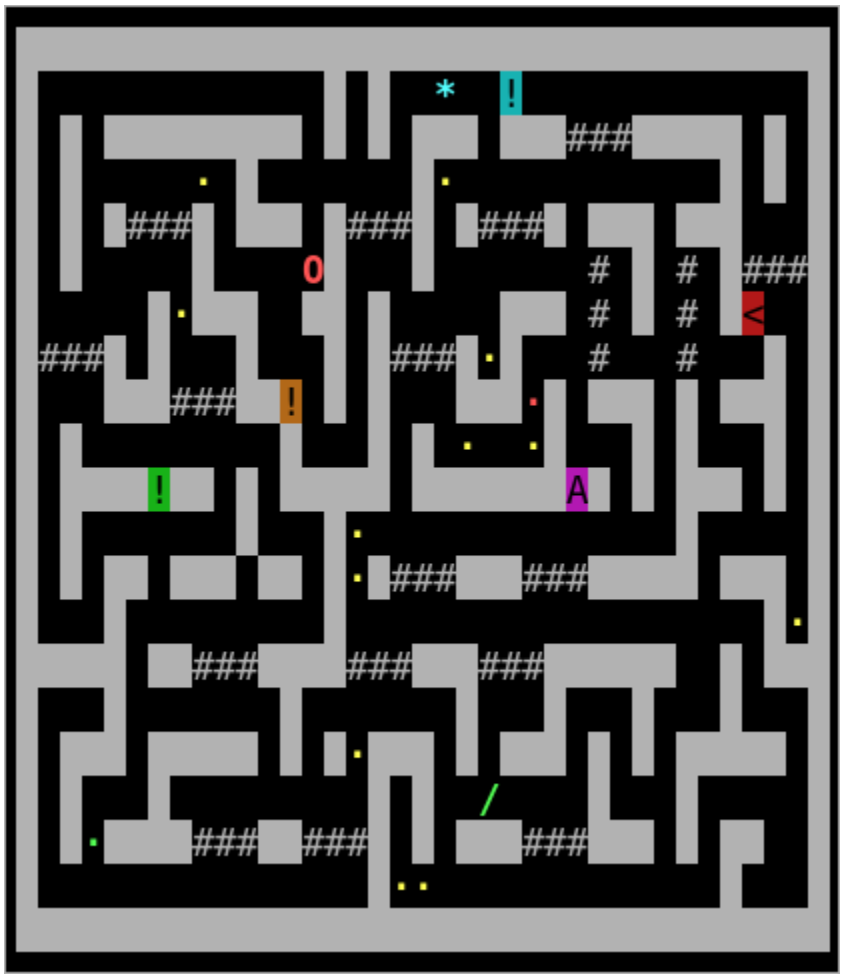
Relazione scritta del progetto di

Laboratorio di Algoritmi e strutture dati

Mirko Alicastro N86001437



compilami su linux con: gcc *.c -o gioco -lm -lcurses
compilami su windows con: gcc.exe *.c -o gioco -lm -lpdcurses

Indice

1. Funzionamento del gioco sviluppato Pag. 3

2. Funzionalità previste Pag. 4

3. Diverse strategie delle guardie implementate Pag. 6

4. Algoritmi sviluppati e strutture dati principali Pag. 9

5. Scelte implementative attuate Pag. 11

6. Altro Pag. 13

7. Lettura labirinto da file Pag. 14

Funzionamento del gioco sviluppato

Il funzionamento del gioco sviluppato è spiegato nell' "intro" del gioco. Quando si esegue il gioco, vengono stampate a video – con un effetto stile macchina da scrivere – le **regole** del gioco. Il suo funzionamento è molto semplice: l'utente muove con le frecce direzionali il **main character**. Questo si continuerà a muovere in automatico fin quando non incontra un ostacolo nell'ultima direzione premuta. Se il main character, muovendosi, dovesse collidere con un muro, o un centro di una **porta girevole**, allora si fermerà. Le porte girevoli possono essere passate solo dal main character e solo nei lati. Il centro della porta girevole farà dunque da perno nella rotazione, e non lo si può oltrepassare. Se una **guardia** cattura **–** ossia collide con – il main character, allora si possono verificare due situazioni, in base allo **score** attuale:

- Se l'utente ha meno di 100 punti: verrà mostrata una schermata di **game over** (vedi schermata 1) e il gioco verrà reinizializzato al livello 1. Tutti i **coltelli usa e getta** e tutti **proiettili** verranno persi. Il punteggio riparte da zero.
- Se l'utente ha 100 punti o più: l'utente sarà collocato – secondo un algoritmo specifico – in una posizione **apparentemente casuale**, e l'utente riprenderà il livello corrente. Tutti i **modificatori** (come la velocità aumentata o diminuita) restano attivi **per scelta implementativa**.

Lo scopo del gioco è quello di raccogliere tutti gli **oggetti obbligatori –** rappresentati da puntini colorati (in base al colore vi è un effetto) – senza farsi catturare dalle guardie. Dopo che tutti gli oggetti obbligatori sono stati raccolti, allora una porta d'**uscita** sarà resa visibile e l'utente dovrà far sì che il main character la raggiunga, per poter **salire di livello**. Quando si sale di livello si guadagnano dei punti bonus e tutti i modificatori si resettano. Ogni livello è rappresentato da un certo numero di guardie, di oggetti obbligatori e di oggetti bonus collocati all'interno di un **labirinto** generato volta per volta in modo **randomico**. Se si preme il tasto **1** si apre un menù con delle funzionalità aggiuntive: **caricare un labirinto da file** testuale, **andare ad un livello specifico**, o **terminare il gioco.** Quando si accede il menù tutti gli oggetti nel gioco smettono di funzionare, o meglio di muoversi. La **pausa** così ottenuta è stata volontariamente di tipo **oscurante**. Non si permette cioè all'utente di vedere la composizione del labirinto, la posizione delle guardie e degli oggetti, per non far pensare con calma all'utente il percorso da seguire. Tutto il gioco deve essere effettuato senza "pause" riflessive.

Funzionalità previste

Molte delle funzionalità previste sono state già spiegate nel funzionamento del gioco, in quanto sono proprietà strettamente legate tra loro. Specifichiamo adesso gli oggetti obbligatori da raccogliere e gli oggetti bonus, ossia quelli la cui raccolta non è necessaria per sbloccare la porta d'uscita e dunque avanzare al livello successivo.

Gli oggetti obbligatori da raccogliere sono:

1) '**Pozione di velocità**': rappresentata da un **puntino** di colore **verde**. Incrementa la velocità del personaggio. Le pozioni di velocità sono cumulative all'interno del singolo livello.
2) '**Strano oggetto**': rappresentato da un **puntino** di colore **magenta**. Inverte la direzione corrente del personaggio "confondendolo".
3) '**Panino**': rappresentato da un **puntino** di colore **rosso**. Rallenta sensibilmente il main character. I panini sono relativamente cumulativi, in quanto vi è un minimo di velocità sufficiente che il main character manterrà.
4) '**Monetina**': rappresentata da un **puntino** di colore **giallo**. Non ha alcun effetto sul main character.

Ogni qualvolta si raccoglie un oggetto, il sistema conferisce dei punti all'utente in base all'oggetto raccolto. Gli oggetti bonus, ossia quelli non obbligatori, sono:

1) '**Bomba**': rappresentata da una **O** ('o' grande) di colore **rosso**. Distrugge tutto ciò che si trova attorno alla bomba ed esplode immediatamente. Se vi è una porta la distrugge. Se vi è un muro lo distrugge. Se vi è una guardia la **uccide**. Se distrugge un muro che delimitava una porta girevole, allora rimuove anche la porta girevole annessa. Anche questa è stata una scelta implementativa, per non lasciare porte girevoli **non funzionanti**, oppure per non lasciare che le porte girevoli ruotassero senza chiudere un passaggio (ossia senza attaccarsi ad un muro).
2) '**Proiettile**': rappresentato da un ***** (asterisco) di colore **ciano**. I proiettili raccolti vengono aggiunti all'**inventario** (visualizzato sulla destra). Si spara premendo '**x**' (con il caps lock disattivato) e il proiettile compirà esattamente un giro intero (anche **circolare**) del labirinto. Se trova qualcosa lungo il suo percorso (sia un muro, sia una porta, sia una guardia, **tranne l'uscita**) la distrugge o la uccide, a seconda di cosa incontra, e scompare. Se un proiettile collide con un 'pezzo di porta' allora, per scelta implementativa, il proiettile distrugge l'intera porta.
3) '**Coltello usa e getta**': rappresentato da uno **/** (slash) di colore **verde**. I coltelli raccolti vengono aggiunti all'**inventario** (visualizzato sulla destra). Lo si usa semplicemente collidendo con una guardia. Non appena si collide con una guardia, se si ha a disposizione almeno un coltello, la guardia muore e il coltello viene buttato via.

Per quanto riguarda le guardie, ogni aspetto ad esse riguardante è spiegato nella pagina successiva. Si tenga a mente che, **il numero**, oltre che il tipo, di **oggetti obbligatori**, di **oggetti bonus**, di **porte girevoli**, di **guardie**, **è** sì **randomico**, **ma** strettamente **legato al** numero del **livello** che si sta giocando. Inoltre **il numero** di tali elementi **è** anche **limitato dalle dimensioni** del labirinto.

Un'altra funzionalità che è bene sottolineare, è che ogni qualvolta il main character perde e riparte il livello, **la sua posizione è** anche in questo caso sì **randomica**, **ma** strettamente **legata alla posizione delle guardie**. È brutto, oltre che avvilente, iniziare un livello o ripetere il livello, partendo già accanto ad una guardia, in quanto è certo che si perderà. Per ovviare a ciò, è stato implementato un algoritmo che determina la posizione **pseudo-randomica** del main character. Il funzionamento di tale algoritmo è il seguente:

1) Parte con un range di 4 caselle, e un numero massimo di 50 tentativi.
2) Cerca, entro il numero di tentativi massimi, una coordinata (**x**,**y**) che non ha guardie nell'arco di *range* caselle in tutte le direzioni. Se la trova, ha finito il suo compito, altrimenti continua.
3) Se il numero di tentativi ha superato il valore di 50, allora se range > 1 si riparte dal punto 2) con range = range-1 e con il numero di tentativi corrente = 0. Se range < 1, non è stato possibile trovare tale coordinata, quindi si ripartirà esattamente là dove il main character aveva cominciato il livello.

L'algoritmo appena descritto è definito dalla funzione:

void get_random_pac_xy(GAMESTATUS *game, **int** *x, **int** *y);

dove:

1) GAMESTATUS *game: è la struttura del gioco;
2) **int** *x: è l'indirizzo della variabile che, al termine della funzione, conterrà la coordinata **x**;
3) **int** *y: è l'indirizzo della variabile che, al termine della funzione, conterrà la coordinata **y**;

Diverse strategie delle guardie implementate

Le guardie del gioco sono i personaggi che cercano di **catturare** il main character, collidendo con esso. Questo sono quattro e sono denominate nei seguenti modi:

1) '**Poliziotto**': rappresentato con un punto esclamativo con background di colore **arancione** .
2) '**Militare**': rappresentato con un punto esclamativo con background di colore **verde**.
3) '**Cecchino**': rappresentato con un punto esclamativo con background di colore **ciano** .
4) '**Forze speciali**': rappresentato con una **A** ('a' grande) con background di colore **magenta**.

Il funzionamento delle guardie è il seguente: una guardia si muove un modo del tutto **casuale** fin quando il main character non entra nel suo raggio d'azione, chiamato **visibilità** della guardia. Se il main character si trova all'interno del riquadro di visibilità della guardia allora il carattere che rappresenta la guardia **lampeggierà** fin quando il main character non riesce ad uscire dal suo raggio d'azione, scappando effettivamente dalla guardia. Quando la guardia lampeggia, ossia quando il main character è *abbastanza* vicino alla guardia, ossia si trova nel suo riquadro di visibilità, allora la guardia adotterà un **sistema di inseguimento** determinato da un algoritmo differente in base alla guardia. Più in particolare, per quanto concerne la guardia di tipo:

1) '**Poliziotto**': ha un riquadro di 5 caselle (in ogni direzione). L'algoritmo di inseguimento è il seguente: viene effettuata una **Breadth-first search** a partire dalla posizione corrente del poliziotto. Se esiste un percorso tra il poliziotto e il main character, allora il poliziotto comincia a seguire il main character spostandosi al suo successore (ricostruendo la lista di pred[]) verso il main character. Altrimenti***** si muove in modo casuale.
2) '**Militare**': ha un riquadro di 6 caselle (in ogni direzione). L'algoritmo di inseguimento è il seguente: viene effettuato **Dijkstra** a partire dalla posizione corrente del militare. Se esiste un percorso tra il militare e il main character, allora il poliziotto comincia a seguire il main character spostandosi al suo successore (ricostruendo la lista di pred[]) verso il main character. Altrimenti***** si muove in modo casuale.
3) '**Cecchino**': non ha un riquadro limitato, o meglio il suo riquadro è delimitato dall'intero labirinto. Quindi questo tipo di guardia si trova perennemente nello stato di inseguimento. L'algoritmo di inseguimento è lo stesso adottato dal *Poliziotto*, però cercherà, avendo un riquadro illimitato, sempre di raggiungere *intelligentemente* il main character.
4) '**Forze speciali**': non ha un riquadro limitato, o meglio il suo riquadro è delimitato dall'intero labirinto. Quindi questo tipo di guardia si trova perennemente nello stato di inseguimento. L'algoritmo di inseguimento è il seguente: viene effettuata un **A*** a partire dalla posizione corrente della guardia verso il main character. Se esiste un percorso,allora la guardia segue il main character spostandosi al suo successore (ricostruendo la lista di pred[]) verso il main character. Altrimenti***** si muove in modo casuale.

Perché vi è un ‘Altrimenti*****’? La risposta è immediata: per specifica di gioco le guardie non devono essere capaci di poter passare le porte. Infatti una guardia non differenzia tra un muro e una porta girevole. Quindi, sebbene il labirinto sia interamente connesso, ossia sebbene da ogni corridoio si può raggiungere un qualsiasi altro corridoio, il main character potrebbe trovarsi ‘rinchiuso’ per scelta all’interno di mura delimitate da una porta girevole. Fin quando l’utente non ruota la porta girevole **non esisterà un percorso** tra nessuna guardia, al di fuori di questo *box*, e il main character.

Ogni guardia ha inoltre velocità differente:

1) ‘**Poliziotto**’: la guardia si muove esattamente **5 volte** più lentamente rispetto la velocità iniziale del main character.
2) ‘**Militare**’: la guardia si muove un po’ meno di **7 volte** più lentamente rispetto la velocità iniziale del main character.
3) ‘**Cecchino**’: la guardia si muove esattamente **5 volte** più lentamente rispetto la velocità iniziale del main character.
4) ‘**Forze speciali**’: si muove un po’ meno di **7 volte** più lentamente rispetto la velocità iniziale del main character.

Ogni guardia la si definisce facilmente tramite l’ausilio delle due funzioni implementate nel gioco. È infatti estremamente semplice definire un nuovo tipo di guardia. Se la si vuole collocare in modo casuale all’interno del labirinto, allora basterà usare questa funzione:

void **add_enemy**(GAMESTATUS *game, **void** *algo, **int** CHARx, **int** color_pair, **int** duration, **int** velocita, **int** visibilita);

dove:

4) GAMESTATUS *game: è la struttura del gioco;
5) **void** *algo: è il puntatore alla funzione che determinerà la strategia di inseguimento della guardia;
6) **int** CHARx: è il carattere che rappresenterà la guardia;
7) **int** color_pair: è la coppia di colori (NB: deve essere già stata inizializzata) con cui rappresentare la guardia;
8) **int** duration: è il numero di turni che la guardia dovrà durare: -1 se infinito;
9) **int** velocita: è la velocità, espressa in “ogni quanti turni” si deve muovere, che la guardia avrà. Ricordare che il main character non si muove ogni turno, ma ogni **5** turni;

10) **int** visibilita: è la visibilità della guardia, ossia il riquadro entro cui comincerà ad inseguire il main character; se si vuole creare una guardia che non ha **teoricamente** riquadri limitati, gli si dovrà mettere come valore il **max**(righe, colonne), dove righe è il numero di righe del labirinto, e colonne è il numero di colonne del labirinto;

Se si vuole creare una guardia ed inoltre la si vuole posizionare esattamente in una coordinata (**x**,**y**) allora è fornita la funzione:

void add_enemy_here(**int** x, **int** y, GAMESTATUS *game, **void** *algo, **int** CHARx, **int** color_pair, **int** duration, **int** velocita, **int** visibilita);

dove, gli argomenti sono gli stessi identici della funzione precedente, ma la posizione non è più randomica bensì quella definita dagli argomenti **x** e **y**. Questa funzione è per esempio, utilizzata quando si deve creare il **labirinto da file**, per poter inserire le guardie esattamente là dove il progettista del labirinto voleva.

Algoritmi sviluppati e strutture dati principali

Diversi algoritmi sviluppati sono già stati esplicitati nelle pagine precedenti. Tra gli algoritmi sviluppati degni di nota vi è anche quello concerne la **generazione casuale di un labirinto**.

La funzione che rappresenta tale algoritmo è la seguente:

void make_recursive_maze(**struct labyrinth** *lab, **int** x_ing, **int** y_ing);

dove:

1) **struct labyrinth** *lab: è la struttura del *maze*;
2) **int** x_ing: è l'intero rappresentante la **x** del punto di ingresso – *di partenza – del main character;*
3) **int** y_ing: è l'intero rappresentante la **y** del punto di ingresso – *di partenza – del main character;*

L'approccio dell'algoritmo è una sorta di **Depth-first search**: inizializza l'intero labirinto murandolo interamente; parte dalla coordinata (**x_ing**,**y_ing**) e rompe il muro in quella casella; si muove di due caselle in una direzione randomica: se la casella di arrivo è un muro rompe queste due caselle, altrimenti cerca un'altra direzione. Se non vi è più alcuna direzione in cui andare si ritorna ricorsivamente alla casella precedentemente analizzata. Al termine di questo algoritmo si ha un labirinto perfetto: con uno ed un solo percorso. Per ciò, è stato creato un **algoritmo che rompe della mura in modo pseudo-casuale**. Tale algoritmo è definito nella funzione:

void destroy_random_wall(**struct labyrinth** *lab, **int** howmany);

dove:

1) **struct labyrinth** *lab: è la struttura del *maze*;
2) **int** howmany: è l'intero rappresentante il numero di mura da rompere nel maze;

L'algoritmo romperà *howmany* mura nel seguente modo: crea una coppia randomica (**x**,**y**). Se la coppia è un corridoio, allora sceglie un'asse: o verticale o orizzontale. Rompe le eventuali mura a nord – sud oppure ad est – ovest , in base all'asse scelto (in modo casuale). Ripete l'operazione appena descritta *howmany* volte. In questo modo nel labirinto vi saranno dei percorsi circolari che ci permetteranno di poter giocare, senza *eccessivi* punti impossibili. Essendo il labirinto randomico, ed essendo la posizione di guardie randomica, è difficile evitare che il main character non abbia chance di vittoria, ma con questo algoritmo si ovvia a questo problema nella maggior parte dei casi.

Altri algoritmi sono descritti nella sezione **Altro**.

Per quanto riguarda le pricipali strutture dati utilizzare abbiamo quattro strutture dati principali create appositamente per il gioco. Queste sono:

1) La struttura **FINESTRE**: contenente i puntatori alla **WINDOW** (di curses) **del cuore gioco**, alla **WINDOW del box di informazioni** (contenente **l'inventario**, il **tempo di gioco trascorso**, i **crediti**, **il** numero di **livello** attuale, **il punteggio** e **le istruzioni** per muoversi e sparare), e alla **WINDOW del box di descrizione**;

2) La struttura **GAMESTATUS**: contenente tutta una serie di informazioni sul gioco, come lo stato, il livello, il punteggio, il numero di oggetti da raccogliere, ecc.
Questa struttura contiene, nel campo **lab** la struttura **labyrinth** definita nella **libreria di labirinti** che gestisce il maze e le visite (BFS, DFS, Dijkstra, A*) all'interno del maze.
Questa struttura contiene inoltre **due strutture heap** (di tipo minheap ordinate in base a chi si deve muovere prima): una che rappresenta le guardie presenti nel livello (nel campo **enemy**), un'altra che rappresenta i proiettili sparati nel livello (nel campo **bullets**).
Questa struttura consente inoltre lo scandire dei turni di gioco attraverso **clock_t**;

3) La struttura **CHARACTER**: contenente informazioni riguardo un personaggio generico del gioco: coppia di colori che lo rappresentano, carattere che lo rappresenta, velocità, durata del personaggio, coordinate correnti, coordinate successive, ultima direzione in cui si è mosso (per consentire al main character di *sparare nella sua direzione*);

4) La struttura **ENEMY**: contiene un identificativo *id_enemy* utile per verificare rapidamente se due enemy sono uguali (l'id è unico), un intero che descrive la visibilità della guardia, un puntatore alla funzione che descrive il movimento della guardia.
Questa struttura contiene inoltre **una struttura CHARACTER** (nel campo **character**) permettendo così il riutilizzo di quei campi (necessari anche per le guardie) e aumentando il riutilizzo del codice;

Ecco il codice delle strutture in questione:

```
struct finestre {
        WINDOW *main;
        WINDOW *info;
        WINDOW *desc;
        int success;
};
```

```
struct gamestatus {
        long started; /* Tempo di quando si inizia a giocare */
        long lasttimesaved; /* Variabile appoggio per il tempo (ultimo tempo di gioco salvato) */
        long timepaused; /* Tempo di gioco in pausa per poter contare il tempo di gioco effettivo */
        long lasttimepaused; /* Variabile appoggio per il tempo (ultimo tempo di pausa salvato) */
        int score; /* Punteggio */
        int over; /* Vale 1 se il gioco è finito, 0 altrimenti */
        int level; /* Livello di gioco */
        clock_t delay; /* Delay tra i round */
        clock_t lasttime; /*  Numero di clock dall'ultimo controllo */
        float delta; /* Tempo passato dall'ultimo round */
        struct labyrinth *lab; /* Struttura del labirinto con la matrice di char */
        int x_uscita; /* Coordinata x dell'uscita */
        int y_uscita; /* Coordinata y dell'uscita */
        int x_ingresso; /* Coordinata x di partenza del main character */
        int y_ingresso; /* Coordinata x di partenza del main character */
        int remaining_items; /* Numero di oggetti rimanenti da raccogliere */
        int total_items; /* Numero totale di oggetti da prelevare assegnati al livello */
        int key; /* Tasto premuto corrente */
        int lastkey; /* Tasto precedente premuto */
        struct heap *enemy; /* Heap contenente tutte le guardie del livello attuale */
        int knife; /* Numero di coltelli disponibili */
        int bullet; /* Numero di proiettili disponibili */
        struct heap *bullets; /* Heap contenente tutti i proiettili correntemente sparati */
};
```

```
struct characterstatus {
        char character; /* Rappresentazione grafica (tramite carattere) del personaggio */
        short color_pair; /* Numero identificativo del colore di fondo/sfondo del carattere */
        int x_next; /* Prossima x */
        int y_next; /* Prossima y */
        int x_curr; /* X corrente */
        int y_curr; /* Y corrente */
        int duration; /* Durata del carattere (-1 per durata infinita) */
        int velocita; /* Ogni quanti turni si deve muovere (0 mai) */
        int velocita_tmp; /* Ad ogni turno velocita_tmp -= 1. Se vale 1, si muove e vale velocita */
        int lastdir; /* Ultima direzione del personaggio */
};
struct enemystatus {
        CHARACTER *character; /* Struttura per la rappresentazione grafica e spaziale */
        void (*move_enemy)(GAMESTATUS *, struct enemystatus *, CHARACTER *pac);
        int id_enemy; /* Id univoco della guardia */
        int visibilita; /* Raggio di visibilità della guardia */
};
```

move_enemy rappresenta il puntatore che punta alla funzione che contiene l’algoritmo di inseguimento della guardia verso il main character.

Scelte implementative attuate

Diverse sono le scelte implementative attuate, e diverse di queste sono state già trattate nelle pagine precedenti. Tra le scelte implementative possiamo comunque sottolineare:

A) La struttura **ENEMY** che ha un campo character di tipo **CHARACTER**. In questo modo si è potuto evitare l'introduzione di campi uguali (ENEMY deve avere praticamente tutti i campi contenuti in CHARACTER, più altri campi), e quindi la modifica di tali campi, o l'aggiunta di altri campi risulta essere più immediata.

B) Il campo '**move_enemy**' della struttura ENEMY, contenente il **puntatore alla funzione** che gestisce il movimento della guardia. Con una sola struttura così possiamo definire più guardie, senza dover creare una struttura per ogni tipo di guardia.

C) La scelta di avere gli elementi più personalizzabili definiti come costanti tramite le **#define**. In questo modo risulta immediato cambiare il tipo di bordo di una finestra, la larghezza del box di informazioni, i tasti associati alle funzioni del personaggio, ecc.

D) Lo scandire del tempo di gioco, con l'assegnazione di punti ogni tot *minuti di gioco effettivo*, trattato nella sezione **Altro**.

E) La politica **anti-cheating**, trattata nella sezione **Altro**.

F) Il **ridimensionamento automatico** della finestra contenente il labirinto in base alle dimensioni di quest'ultimo. Inizialmente, *per scelta*, il labirinto ha una dimensione fissata. Se si carica un labirinto da file, però, lo schermo viene ridimensionato. Se si procede con i livelli, dopo aver caricato il labirinto da file, la dimensione del labirinto generato casualmente sarà esattamente quella del precedente. Inoltre, sempre *per scelta*, la dimensione di un labirinto caricato da file non può essere minore di 20x12. Questo valore è facilmente modificabile sostituendo alla funzione che carica il labirinto da files gli argomenti. La funzione in questione è: load_maze_from_file, ben documentata nella libreria dei labirinti.

G) Un box di **interazione con l'utente**. Sotto il box di gioco e' presente un box di testo che contiene dei messaggi, diretti all'utente, in base a *ciò che accade* nel gioco. Se l'utente uccide una guardia, in base a come la uccide sarà mostrato un messaggio in tale box. Se l'utente raccoglie un oggetto obbligatorio, in base all'oggetto sarà mostrato un messaggio in tale box. Se l'utente raccoglie un oggetto bonus, in base all'oggetto sarà mostrato un messaggio in tale box. Se l'utente ha raccolto tutti gli oggetti, scaturisce un evento importante. Se l'utente viene catturato da una guardia, scaturisce un evento importante. Se scaturisce un evento importante, sarà mostrato un messaggio **lampeggiante** in tale box, per attrarre l'attenzione dell'utente. Se l'utente guadagna punti (per la sua *permanenza* nel gioco), sarà mostrato un messaggio in tale box. La presenza di questo box nasce per **non annoiare** l'utente, ma anzi coinvolgerlo, anche in maniera scherzosa, in ciò che sta facendo.

H) La struttura **FINESTRA**. Tale struttura è stata pensata per tutte le funzioni che gestiscono più WINDOW: anziché passare più puntatori a WINDOW, si passa l'intera struttura contenente le varie WINDOW, tutte quindi accessibili dalla struttura.

I) Non si può mettere in **pausa** guardando il labirinto. Il gioco è pensato per essere un gioco di velocità e impulsività, non di calma e riflessività. Pertanto, *per scelta*, per mettere pausa si deve aprire il menù – ciò bloccherà tutto ciò che concerne il labirinto. La pausa realizzata è così una **pausa oscurante**.

Altre due scelte implementative, seppur di impatto nullo dal punto di vista algoritmico, sono:

J) L'utilizzo della **lingua inglese** per qualsiasi tipo di **output** visivo;

K) L'introduzione di un **intro di gioco**, dove vengono spiegati i vari pulsanti con cui muovere e azionare il main character e i vari oggetti con cui si interagisce. L'intro è stato diviso in due schermate, oltre che per una questione di spazio, anche per evitare di sovraccaricare l'utente con troppe informazioni e **annoiarlo**.

Il coinvolgimento dell'utente è stato considerato come fattore di grande rilevanza, e questo spiega a pieno anche quest'altra scelta implementativa:

L) La presenza nel menù della voce **Jump to level**, per far giocare e vedere all'utente cosa accadrebbe se, per esempio, arrivasse al livello **999999**. Questo sarebbe un raggiungimento del tutto teorico, perché nessun utente arriverebbe mai ad un tale livello, ma è appunto per stuzzicare la curiosità dell'utente. Del resto è un po' come i trucchi, c'è sempre chi li vuole usare, anche se solo per curiosità.

Un'altra scelta implementativa nella realizzazione del progetto è stata la seguente:

M) Il labirinto è considerato **circolare**. Sia il main character, sia i proiettili, sia le guardie possono muoversi da un lato all'altro come se fosse collegato. Tuttavia, *per scelta implementativa*, le guardie non guardano al di là dei confini del labirinto. Nel calcolare i percorsi non considerano mai dunque i confini come circolari. Questo è stato implementato nel seguente modo per facilitare l'utente nello scappare dalle guardie, ma le funzioni che calcolano gli adiacenti *circolari* sono state ugualmente realizzate nella libreria. La funzione in questione è **int** fix_circular(**int** i, **int** max) , che ritorna la conversione di **i** (che è più grande o uguale di max o è negativo), nella coordinata circolare corrispondente. Se i non è negativo ed è più piccolo di max, allora ritorna i stesso.

Queste sopra elencate sono le principali scelte implementative attuate nella realizzazione del codice. Le scelte implementative D) e E) sono spiegate nella sezione successiva.

Altro

Questa sezione puntualizza alcuni punti che riguardano non solo scelte implementative.

1) Innanzitutto, nello sviluppo del progetto, è stato costantemente utilizzato **valgrind** come **strumento di analisi**, in particolare è stato usato come *memory error detector*. Il risultato è stato quello di realizzare un progetto che non abbia alcuna perdita di memoria, così facendo questo *tool* ha permesso lo sviluppo di funzioni *corrette*, garantendo una certa *robustezza* al progetto.
2) **L'intero codice è pieno di commenti**, **in italiano** e non inglese a differenza delle scritte di output di gioco – non ridondanti né inutili – e le **funzioni** sono generalmente molto **contenute in termini di linee** (commenti esclusi) al fine di poter continuare il progetto, perfezionare le funzioni e superare le difficoltà in modo estremamente naturale.
3) È stato deciso di **assegnare** dei **punti bonus allo scorrere del** tempo di gioco. A tal fine è stato necessario aggiungere ben 2+2 campi contenenti informazioni sul tempo, in particolare un campo principale contiene il tempo in cui l'utente ha iniziato a giocare, e un altro l'ultimo tempo salvato. Poi vi sono altri due campi che hanno la stessa logica, ma il principale contiene il tempo in cui l'utente è stato nella schermata di pausa. In questo modo si è potuto lavorare sul **tempo effettivo di gioco**. E a tal proposito è stata inserita una **politica anti-cheating**: se l'utente porta avanti di un'ora, per esempio, l'orario del sistema operativo, il gioco mostrerà un messaggio 'simpatico' che scoraggia l'utente ad imbrogliare nella vita, e che anzi, gli impedisce di giocare ancora, lasciando il main character solo in un labirinto vuoto, senza mura, porte, guardie né oggetti.
   L'utente è **forzato** *per scelta implementativa*, a chiudere l'applicazione e a rigiocare – dopo aver riflettuto sul voler imbrogliare!
4) Quando l'utente viene catturato da una guardia, e non ha sufficienti punti per ripartire dal livello corrente, allora viene mostrata una schermata con un **ascii art** lampeggiante rappresentante la scritta **Lose**.

Lettura labirinto da file

Assieme al progetto, sono forniti tre file testuali: *lab1.txt*, *lab2.txt*, *lab3.txt*. Questi tre files possono essere aperti all'interno del progetto tramite l'apposita funzione (**Menù > Load maze from file**). Per creare un files che sia un labirinto **valido** bisogna che innanzitutto, come spiegato nella sezione 5, pag. 11, il labirinto abbia dimensioni minime maggiori di 20x12 e dimensioni massime minore di 40x70, per *scelta implementativa*. Affinché il labirinto venga importato correttamente però, non basta questo vincolo delle dimensioni.

Per creare un labirinto valido bisogna innanzitutto conoscere la sintassi, ossia l'associazione tra caratteri testuali e gli elementi che possono far parte del labirinto. I caratteri sono i seguenti:

1) '**0**': (zero) rappresenta il corridoio.
2) '**-**': (trattino) rappresenta il muro.
3) '**#**': (cancelletto) rappresenta un pezzo di una porta girevole.
4) '**R**': ('r' grande) rappresenta l'oggetto che inverte la direzione del main character.
5) '**C**': ('c' grande) rappresenta l'oggetto monetina.
6) '**F**': ('f' grande) rappresenta l'oggetto che aumenta la velocità del main character.
7) '**S**': ('s' grande) rappresenta l'oggetto che diminuisce la velocità del main character.
8) '**B**': ('b' grande) rappresenta l'oggetto bonus del proiettile.
9) '**K**': ('k' grande) rappresenta l'oggetto bonus del coltello usa e getta.
10) '**O**': ('o' grande) rappresenta l'oggetto bonus della bomba.
11) '**D**': ('d' grande) rappresenta il muro che poi diventerà l'uscita del labirinto.
12) '**X**': ('x' grande) rappresenta il corridoio di partenza del main character (ingresso labirinto).
13) '**1**': (uno) rappresenta la guardia poliziotto.
14) '**2**': (due) rappresenta la guardia militare.
15) '**3**': (tre) rappresenta la guardia cecchino.
16) '**4**': (quattro) rappresenta la guardia forze speciali.

Avendo presente questa sintassi, si può creare un labirinto testuale semplicemente scrivendo un file di testo con tante righe e tante colone quante le righe e le colonne del labirinto – purché maggiori del minimo consentito – e con questi caratteri laddove vogliamo che nel labirinto del gioco vi sia l'elemento in questione. Il vincolo sulle dimensioni del labirinto è facilmente modificabile. È sufficiente cambiare i valori passati come argomento della funzione load_maze_from_file specificando i nuovi *minimi* e *massimi*.