GIOCO DEL SERPENTE

Marchetti Leonardo, Mezdrichki Zanko, Rondini Andrea

Il nostro progetto si ispira al comune gioco dell'oca, composto da un tabellone con caselle con vari effetti e un massimo di quattro giocatori. Abbiamo deciso, per non rendere monotono il gioco, che la disposizione delle caselle sarà casuale ad ogni partita, con un'unica limitazione: due caselle dello stesso tipo non possono essere vicine.

Abbiamo diviso il nostro lavoro in più file:
-struct.hpp, contenente tutte le struct utilizzate.

-tabellone.hpp, nel quale abbiamo inserito le dichiarazioni delle funzioni che servono per creare il tabellone.

-gioco.hpp, dove abbiamo messo le dichiarazioni delle funzioni relative al movimento vero e proprio delle pedine.

-tabellonefunzioni.cpp, dove sono state date le definizioni delle funzioni di tabellone.hpp.

-giocofunzioni.cpp, file in cui abbiamo inserito le definizioni delle funzioni contenute in gioco.hpp.

-giocomain.cpp, nel quale è stata inserita la funzione int main().

-functiontest.cpp, in cui, attraverso "Doctest" abbiamo potuto testare gli effetti delle caselle.

-tabellonetest.cpp, nel quale, grazie a "Doctest" abbiamo verificato che il tabellone venisse creato correttamente.

Partiamo con la descrizione del codice.

Le fondamenta del programma sono due std::vector, il vettore "tabellone" (std::vector<std::string>), contenente le diverse caselle. A partire da esso abbiamo potuto dare ad ogni casella le proprie caratteristiche, come la posizione e il colore, oppure l'effetto che avrà sui giocatori; mentre l'altro std::vector è il vettore "giocatore" (std::vector<std::giocatore>), contenente i dati dei giocatori, come il suo nome, la sua posizione e il colore scelto per la pedina.

Partiamo delle funzioni descritte nel file tabellonefunzioni.cpp.

Appena viene avviato il programma verrà stampato sul terminale una piccola presentazione del gioco, in cui spieghiamo come giocare, le regole e gli effetti delle caselle speciali. Tutto ciò è possibile grazie alla funzione infoeregolamento, di tipo void.

In seguito, si attiverà la funzione richiestagiocatori. Questa funzione richiede il numero di giocatori, che abbiamo ristretto tra 2 e 4, attraverso un input da terminale che è controllato con std::cin.fail() affinché sia di tipo long unsigned int(utile per evitare avvertimenti di conversione di segno o di conversione di tipi), in caso sia diverso viene applicato il metodo std::cin.clear() per resettare lo status di carattere invalido e

```
std::cin.ignore(std::numeric_limits<std::streamsize>::max()
```

per ignorare ogni altro input restante sulla linea oltre al primo carattere. Una volta inserito un numero accettabile di giocatori si attiva la funzione richiestaLunghezza. Questa funzione richiede la lunghezza desiderata del tabellone al giocatore, con quattro opzioni disponibili: 32, 43, 54 o 65. Anche richiestaLunghezza utilizza, analogamente alla funzione precedente, `std::cin.ignore` e `std::cin.fail`.

Dopo che il numero di giocatori e la lunghezza del tabellone sono state inserite si attiva la funzione CreaTabellone. Questa funzione utilizza

```
std::random_device rd;
std::mt19937 g(rd());
```

Per generare numeri casuali. Prima viene creato il vettore di int "caselle", che con std::iota è riempito in ordine crescente a partire da 0 con i valori generati.

Poi viene creato il vettore di stringhe "tabellone", anche esso, così come il vettore "caselle", con la stessa lunghezza scelta dal giocatore.

Una volta creati i due vettori si attiva un ciclo for che controlla i valori di ogni elemento del vettore "caselle" e applica la funzione Casella. La funzione Casella controlla ogni int "I", corrispondente alla posizione dell'oggetto nel vettore "caselle", e vi associa una stringa nella posizione corrispondente del vettore "tabellone". Alla fine di questo ciclo il vettore "caselle" sarà copiato nel vettore "tabellone", il quale invece di contenere numeri conterrà stringhe associate alla tipologia di casella. Questa associazione numero -> stringa è fatta in base ad un range di numeri, in questo modo, nonostante una lunghezza variabile del tabellone, le proporzioni tra caselle Bonus, Prigione e tutte le altre tipologie saranno mantenute, eccetto per le caselle Via e Fine, le quali saranno sempre il primo e l'ultimo elemento del vettore.

Riempito "tabellone", inizia un ciclo for che scorre tutti gli elementi del vettore. All'interno del ciclo for, c'è un ciclo do-while che continua fino a quando non ci sono più coppie di valori consecutivi nel vettore che soddisfano la condizione sonoConsecutivi, cioè sono della stessa tipologia. Questa condizione è verificata utilizzando l'algoritmo std::adjacent_find. Nel corpo del ciclo do-while, il vettore tabellone viene prima mischiato casualmente utilizzando la funzione std::shuffle. Questa operazione esclude il primo e l'ultimo elemento del vettore, poiché la casella Via e Fine non devono cambiare di posizione. Successivamente, un altro ciclo for scorre il vettore tabellone (escludendo il primo e l'ultimo elemento). Se due elementi consecutivi soddisfano la condizione sonoConsecutivi, la funzione scambiaNonConsecutivi viene chiamata per scambiare l'elemento corrente con il primo elemento diverso nel vettore.

Infatti, questa funzione scorre il vettore e se trova un elemento che non è uguale rispetto all'elemento all'indice dato e all'elemento precedente, scambia questi due elementi. La funzione si interrompe non appena viene effettuato uno scambio.

A questo punto il vettore tabellone è stato creato con le caselle opportunamente posizionate, quindi si attiva la funzione getplayers.

La funzione getplayers restituisce un vettore "giocatori", di lunghezza Ngiocatori prima inserita. Per creare i giocatori è stata creata la struct "giocatore", la quale contiene, rispettivamente per ogni giocatore, nome, colore della pedina, posizione relativa nel tabellone, i bool "turno attivo"

e “prigione”, impostati rispettivamente a vero e falso, che saranno modificati da alcune funzioni delle caselle più avanti, e l’int “turniprigione”, anch’esso utilizzato per l’effetto prigione.

Creato il vettore “giocatori”, un ciclo for chiede all’utente i dati di nome e del colore desiderato della pedina per ogni giocatore. Per far sì che due giocatori non scelgano lo stesso colore, il programma controlla l’input dell’utente con un vettore “pedine” creato in precedenza, contenente i char “R” “B” “G” “V”, rispettivi per rosso, blu, giallo e verde. Ad ogni ciclo del for è creato un giocatore generico g, al quale saranno attribuite le caratteristiche desiderate prima di esser inserito nel vettore “giocatori” tramite .pushback(). Innanzitutto, verrà chiesto all’utente il nome del giocatore ad indice i + 1(dato che i parte da 0), in seguito viene creato un bool “colore_scelto”, inizialmente falso. Dopodiché inizia un ciclo do while, il quale chiede all’user di scegliere il colore della pedina, che sarà il valore della variabile temporanea “ColoreTemp”, finché il carattere inserito sarà uno dei 4 nel vettore “pedine”.

A questo punto è creato un altro bool “colore_disponibile”, anch’esso impostato come falso di base, e si avvia un ciclo for che opera tra l’inizio del vettore “PedineDisponibili” e la sua fine; questo ciclo compara l’input dell’utente con i 4 char, e se trova che è uguale ad uno dei 4 allora rimuove quel colore dal vettore e imposta “colore_disponibile” a vero, in modo da non far stampare sul terminale la frase contenuta in if(!colore_disponibile). Nel caso in cui l’input dell’utente sia diverso da tutti i char nel vettore “PedineDisponibili”, sia che il colore sia già stato preso, quindi è stato cancellato dal vettore, oppure che sia un carattere non valido come una lettera diversa o un numero, il programma richiederà il “ColoreTemp” finché l’input non sarà valido.

Una volta registrati nome e colore del giocatore g, attraverso .pushback() questo giocatore sarà inserito nel vettore “giocatori” e il ciclo for passerà al prossimo, ripetendo fino al numero di giocatori inserito.

Una volta generati sia i tabelloni e i giocatori, si avvia la funzione “Mostratabellone” che utilizza sfml per creare il tabellone di gioco. Questo tabellone si genera ogni volta che avviene un movimento di un giocatore e necessita di essere chiuso e riaperto ogni volta (viene chiuso automaticamente anche dopo dieci secondi).

Il tabellone verrà generato con una forma a serpente, creando prima i quadrati centrali e poi alternando una casella a destra e una a sinistra per collegare le diverse file di caselle. Questo viene fatto separando i casi a seconda delle colonne. Se le colonne in cui si creano le caselle non sono la prima o l’ultima, viene generata una riga di 8 caselle. Nella riga successiva non viene creata nessuna casella. Per la prima e l’ultima colonna, affinché si formi un serpente, devono essere create 3 caselle ogni 4 righe. Per questo motivo vengono utilizzati degli if, che si attivano a seconda se il numero di righe è divisibile per 4, controllando anche il resto che verrebbe nel caso non lo fosse.

Nel modo in cui abbiamo impostato il codice il numero di casella sarà uguale al numero di posizione della stringa nel vettore.

Ad ogni casella è poi associato una tipologia, già determinata con la funzione “CreaTabellone”, e con la funzione

```
square.setFillColor(ColoreCaselle(tipo));
```
ad ogni casella viene associato un colore.

Una volta create tutte le caselle con

```
sf::RectangleShape border(sf::Vector2f(49, 49));
border.setPosition(square.getPosition());
border.setFillColor(sf::Color::Transparent);
border.setOutlineColor(sf::Color::Black);
border.setOutlineThickness(2);
window.draw(border);
 window.draw(square);
```
Vengono creati i bordi neri per far risaltare le caselle rispetto allo sfondo.

Successivamente viene tratta la realizzazione delle pedine. Ad ogni colore dei giocatori viene associata un'icona in base al colore scelto, attraverso uno switch.

A questo punto viene trattato il movimento delle pedine sul tabellone.

Le pedine sono posizionate nelle caselle in base al parametro.posizione del relativo giocatore, ponendo maggiore attenzione ai casi in cui il giocatore sia in una delle caselle che si comportano come da ponte per le altre righe di caselle.

Da notare come nelle righe dispari (quindi a partire dalla prima, poi passando alla terza e così via) il movimento della pedina va da sinistra verso destra, mentre per le altre righe da destra verso sinistra, dunque bisogna distinguere i due casi.

L'ultima parte relativa a sfml contiene la creazione di una legenda con i colori e nomi delle varie caselle. Prima è creata una finestra, che verrà posizionata in alto a sinistra, assieme al relativo bordo e colore (bianco). Poi un ciclo for mostra per ogni tipologia di casella la sua icona sul tabellone; quindi, creerà un piccolo rettangolo simile alle caselle di gioco e lo colorerà con il colore associato accanto al quale verrà scritto la tipologia di casella, prima di muoversi alla riga successiva con un fattore di 40 punti.

Il processo sarà ripetuto 9 volte, il numero di diverse tipologie di caselle.

Compilando con Cmake la funzione Mostratabellone presenta errori di memory leak.

Essi non sono dovute ad altre funzioni, oppure oggetti, dato che, rimuovendo la funzione Mostratabellone, questi errori scompaiono (l'abbiamo testato commentando tutta la parte relativa a sfml).

Una volta creato quindi il tabellone, le pedine e i dati dei giocatori, il gioco può iniziare(si passa alla funzione giocomain.cpp).

Il gioco continua fino a che un giocatore non vince, cioè non supera la lunghezza massima del tabellone. Il ciclo di gioco si compone di un "while", che continua finché un giocatore vince, cioè quando la sua posizione è maggiore o uguale alla lunghezza scelta per il tabellone, dentro al quale un ciclo "for" contiene le funzioni necessarie per far giocare ogni giocatore.

Una volta iniziato il turno, si attiva la funzione "movimento", contenuta nel file giocofunzioni.cpp.

Questa funzione prima controlla 2 attributi del giocatore: il bool “giocatori[j].saltaturno”, e il bool “giocatori[j].prigione”, con “j” il numero del giocatore attuale (ad esempio “giocatore 2”). Nel caso “giocatori[j].saltaturno” sia vero, il giocatore non si potrà muovere in questo turno e “giocatori[j].saltaturno” sarà nuovamente impostato come falso, dovendo così aspettare il prossimo mentre se “giocatori[j].prigione” è vero ci sono due opzioni.

Il bool “giocatori[j].prigione” può essere modificato solo dalla casella “prigione”, la quale, una volta che un giocatore finisce il turno su una di queste caselle, imposta “giocatori[j].prigione” come vero e modifica l’attributo int giocatori[j].turniinprigione da 0 a 3. Al prossimo turno del giocatore in prigione questo avrà la possibilità di tirare un dado, creato con

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<int> dist(1, 6);
```

. Se il risultato del tiro è maggiore o uguale a 5, “giocatori[j].prigione” sarà ridefinito falso, altrimenti il giocatore resta in prigione e giocatori[j].turniinprigione verrà diminuito di 1. Questo processo viene ripetuto fino a che il giocatore tira un 5 o un 6 oppure giocatori[j].turniinprigione ritorna ad essere 0. I lanci dei dadi per uscire non vengono effettuati tutti nello stesso turno, ma “sostituiscono” la possibilità di movimento del giocatore. Appena il giocatore esce, nello stesso turno può lanciare i dadi per muoversi.

Nel caso in cui sia “giocatori[j].prigione”, che “giocatori[j].saltaturno” siano falsi, il giocatore prima di tutto tirerà il dado, creato in modo analogo al ciclo della prigione, il quale risultato sarà aggiunto al suo attributo “giocatori[j].posizione”.

Una volta che il giocatore ha tirato il dato, il programma controlla caso per caso il

```
std::string tipo = tabellone[static_cast<size_t>(giocatori[j].posizione)]
```

,cioè la stringa associata all’elemento del vettore “tabellone” con posizione del giocatore, con 7 diversi effetti disponibili (ad esempio se il giocatore è in posizione 6, il programma leggerà la stringa associata all’elemento 6 nel vettore “tabellone”).

Da notare che la casella Via è la casella numero 0, mentre la casella Fine sarà di conseguenza la numero ‘lunghezza-1’.

Oltre all’effetto associato alla stringa “prigione” già discusso è possibile attivare l’effetto “pausa”, il quale imposta “giocatori[j].saltaturno” vero, così il giocatore dovrà restare fermo il prossimo turno.

Gli altri effetti possibili sono: “scambio”, “bonus”, “SCF”, “serpente”, “domanda” e “niente”.

L'effetto scambio cambia l’attributo “giocatori[j].posizione” con la posizione di un altro giocatore scelto a caso utilizzando

```
std::uniform_int_distribution<size_t> scambio(0, giocatori.size() - 1);
altro_giocatore = scambio(gen);
```

,il quale genera il numero casuale “altro_giocatore” tra 0 e il numero di giocatori meno uno. Questa parte di codice è inserita in un ciclo do-while, dal momento che “altro_giocatore” deve essere diverso da “j” affinché avvenga lo scambio, il quale è possibile grazie all’algoritmo std::swap.

L'effetto “bonus” permette invece al giocatore di lanciare nuovamente il dado, attivando l’effetto della casella sulla quale si ferma.

La casella “serpente” è un analogo della casella “oca” nel gioco dell’oca, trasportando il giocatore alla successiva casella della stessa tipologia nel tabellone. Una volta attivato l’effetto un ciclo “for” cerca nel vettore di stringhe “tabellone” una casella con stringa associata “serpente”, partendo dalla posizione attuale dal giocatore +1 fino alla fine: nel caso vi sia una casella di questo tipo, la posizione del giocatore attuale sarà sostituita da quella della casella trovata, altrimenti la posizione non cambierà.

La casella “SCF” attiva la funzione GiocoSFC, la quale ritorna un numero intero che dipende dall’esito della partita tra il giocatore e il computer a “sasso carta e forbici”. Il computer sceglierà una delle tre opzioni in base ad un generatore di numeri casuali tra 1 e 3, senza mostrare la scelta all’utente. Dopodiché verrà chiesto all’utente di scegliere una delle tre opzioni, da inserire come parole intere in caratteri minuscoli, altrimenti l’input non sarà accettato. Una volta che le scelte sono state fatte, vengono comparati i vari casi con degli “if” statement per determinare se il giocatore ha vinto o meno. Nel caso di pareggio, la funzione restituirà il valore 1, mentre restituirà 2 per una vittoria e 3 per una sconfitta.

Dopo che la funzione “GiocoSFC” avrà restituito un valore, si attiverà la funzione EffettoSCF, che in base al risultato di sasso carta forbici modificherà la posizione del giocatore attuale. Se il giocatore ha vinto, avanzerà di 2, se ha perso tornerà indietro di una casella mentre nel caso di pareggio resterà fermo.

La casella niente, intuitivamente, non ha alcun effetto sul giocatore.

L'ultima casella possibile è quella della Domanda. La funzione quiz prende in ingresso un vettore di stringhe (in cui ogni elemento corrisponde a una linea del file domande.txt), genera poi un numero casuale, che va da zero alla lunghezza del vettore diviso per otto, diviso come il file contenente le domande.

Il file con le domande contiene ogni otto righe un blocco contente l’indice della domanda (ad esempio Domanda 1), poi la domanda, 4 righe con le opzioni, la risposta (scritta [Risposta D] per es), e una riga vuota per distanziarla dalla prossima. Successivamente converte il numero ottenuto nella riga corrispondente moltiplicandolo per 8 e sommando 1, in modo da mostrare la domanda, dopodiché mostra le 4 righe successive contenenti le 4 risposte possibili.

A questo punto sarà richiesto all’utente di digitare il carattere corrispondente alla risposta desiderata, con uniche opzioni “A”, “B”, “C” e “D”, il quale sarà il valore della variabile temporanea “risp”. Una volta registrata la risposta dell’utente la funzione costruisce una stringa concatenando attraverso il + le stringhe “[Risposta”+ risp +”]”, creata la stringa la confronta con l’elemento corrispondente al numero casuale generato prima*8 +6, corrispondente alla riga nella quale è contenuta la risposta, attraverso un “if” statement controlla se siano uguali, se è così restituisce true, altrimenti false. Se risponde correttamente il giocatore avanzerà di due caselle, altrimenti indietreggerà di una casella.

Nel main è presente il codice sottostante:

```
  std::ifstream domande("/home/zanko/pf_labs/progetto/domande.txt");
```

```
  std::string Q;
  std::vector<std::string> v;
  while (getline(domande, Q)) {
    v.push_back(Q);
  }
  domande.close();
```

Il quale permette di aprire il file di domande, estrarre una stringa corrispondente alla domanda scelta casualmente e inserirla nel vettore "v" così da essere più accessibile al runtime.

In conclusione, un ciclo di gioco tipico si svolgerà così: prima saranno chiesti all'utente il numero di giocatori e i loro attributi, poi i giocatori, ognuno nel proprio turno, lanceranno il dado per determinare il loro spostamento. Il tabellone si aggiornerà all'inizio di ogni turno, poi dopo che il dado sarà stato tirato, dunque anche alla fine del turno, una volta che si sarà attivato l'effetto della casella sulla quale il giocatore si è fermato. Questo ciclo andrà avanti scorrendo tra i giocatori fino a che uno di questi supererà la casella finale, decretandolo vincitore e terminando il gioco.

In functiontest.cpp sono contenuti i test che controllano il corretto funzionamento delle funzioni all'interno fi giocofunzioni.cpp attraverso il file doctest.h. Al suo interno oltre ai test sono riportate anche le funzioni testate ma private della parte grafica in output, poiché si è preferito concentrarsi sulla parte computazionale di queste, in modo da non riscontrare problemi con la generazione di immagini durante i test. Inoltre, quando eseguito viene mostrato a schermo soltanto il risultato dei test, senza la parte che normalmente verrebbe stampata durante la normale esecuzione del programma.

Nel primo testcase "check functions" dopo aver creato i giocatori P1 e P2, vengono inseriti all'interno del vettore Players, successivamente vengono creati due vettori di stringhe: il primo tabellone, contenente una serie di stringhe corrispondenti alle caselle del tabellone, il secondo dom1 contenente le prime otto righe del documento "domande.txt", corrispondenti alla prima domanda. Il primo subcase "controllare fine pausa" controlla che se si è finiti in precedenza su una casella pausa, la funzione movimento salti il turno del giocatore, andando a modificare l'attributo saltaturno da true a false. Nel codice questo avviene modificando l'attributo saltaturno di P1 da false a true, successivamente si invoca la funzione movimento su P1, infine attraverso un CHECK si controlla che l'attributo saltaturno di P1 sia uguale a false.

Il due successivi testcase, "controllare domande e quiz 1" e "controllare domande e quiz 2" controllano il corretto funzionamento delle funzioni quiz e domanda. Nel codice di entrambi per prima cosa è modificata la posizione di P1 da 0 a 1 successivamente viene invocata la funzione domanda su P1, e come altro attributo viene dato Dom1, oltre a ciò, utilizzando:

```
std::istringstream input("D");
std::cin.rdbuf(input.rdbuf());
```

viene simulato l'input da tastiera sul terminale della risposta, che per "controllare domande e quiz 1" è "D", cioè la risposta corretta, mentre per "controllare domande e quiz 2" è "A" che è sbagliata. Infine, in entrambe i subcase, attraverso un CHECK, si controlla che la posizione di

P1 corrisponda a 3 quando la risposta inserita è corretta, 0 quando è errata. Essendo ci un'unica domanda è semplice controllare se l'output della funzione è corretto.

Nel successivo testcase, si controlla il corretto funzionamento della funzione movimento quando il giocato è finito in precedenza in prigione. Come nel precedente test case, vengono dichiarare le stesse variabili quindi, P1, P2, players e dom1, lunica differenza è tabellone che è:

```
std::vector<std::string> tabellone{"Via","Niente","Pausa","Serpente", "Serpente",
“Pausa", "Niente",  "Prigione", "Scambio", "Bonus","SCF",   "Domanda", "Fine"};
```

Non sono presenti caselle “Prigione”, nelle prime sette posizioni del vettore, in modo tale da non interferire con i test.

Nel testcase sono presenti tre subcase, “controllare termine prigione”, “controllare 3 turni prigione” e “controllare 2 turni prigione”. In tutti e tre i subcase per prima cosa è modificato l'attributo prigione di P1, e successivamente anche l'attributo turniinprigione con il relativo numero di turni per ciascuno subcase(in ordine 1 per il primo, 3 per il secondo 2 per il terzo). In tutti e tre i subcase viene invocata la funzione movimento e simulato l'input da tastiera utilizzando lo stesso codice usato in precedenza nei test sulla funzione domanda, in modo da effettuare l'eventuale tiro del dado. Nel subcase “controllare termine prigione” a differenza degli altri, viene controllato attraverso dei CHECK che: l'attributo prigione sia uguale a false, che la posizione del giocatore sia maggiore di uno, che sia maggiore di sei e che sia diversa da tre. Il primo CHECK controlla che dopo questo turno essendo l'ultimo, il giocatore esca dalla prigione, i CHECK successivi per controllare che abbia tirato il dado e si sia mosso, per cui la sua posizione sarà compresa tra uno e sei, oltre a ciò, controlla anche che non sia sulla casella tre, poiché essendo serpente ed essendocene un'altra della stessa tipologia in quattro, per l'effetto di questa casella, ogni volta che verrà tirato un tre il giocatore concluderà il turno sulla casella successiva.

I subcase successivi sono leggermente diversi, per prima cosa viene creata la variabile bool prova, successivamente attraverso un if si controlla che l'attributo prigione sia true. Se questo è vero, quindi corrispondente al caso in cui non si è riusciti a effettuare cinque o sei, prova assume il valore della proposizione:

```
prova = (Players[0].posizione == 0);
```

Altrimenti se prigione corrisponde a false, quindi il giocatore è riuscito ad uscire, prova assume il valore della proposizione:

```
prova = ((Players[0].posizione >= 1) && (Players[0].posizione <= 6) &&
               (Players[0].posizione != 3));
```

Che corrisponde allo stesso controllo sulla posizione fatto nel test precedente.

Infine, attraverso un check si controlla il valore di prova.

Il testcase dopo controlla il corretto funzionamento della funzione movimento, in particolare che il lancio del dado avvenga in modo corretto e che il numero generato sia compreso tra uno e sei. Esattamente come nei precedenti vengono dichiarare le stesse variabili quindi, P1, P2, players e dom1. In questo caso la variabile tabellone si presenta così:

```
std::vector<std::string> tabellone{"Via", "Niente", "Pausa","Serpente",
"Serpente", "Prigione", "Pausa", "Niente", "Scambio", "Bonus", "SCF", "Domanda",
"Fine"};
```

É presente un unico subcase, in cui viene invocata la funzione movimento su P1 e simulato l'input da tastiera per tirare il dado sempre utilizzando il codice già citato in precedenza. Successivamente si effettuano i medesimi CHECK sulla posizione di P1 che vengono effettuati nel sub case "controllare termine prigione", ovvero che la posizione del giocatore sia maggiore o uguale a uno, minore o uguale a sei, e che sia diversa da tre per gli stessi identici motivi.

Nel testcase "check scambio" viene controllato il corretto funzionamento della funzione movimento quando un giocatore atterra sulla casella scambio. Come nei precedenti testcase vengono dichiarare le stesse variabili quindi, P1, P2, players e dom1. In questo caso la variabile tabellone si presenta così:

```
std::vector<std::string> tabellone{"Via", "Scambio", "Scambio", "Scambio",
                                   "Scambio", "Scambio", "Scambio"};
```

In questa maniera viene forzato l'atterraggio del giocatore su una casella Scambio.

Nel subcase "controllare scambio" viene invocata la funzione movimento su P1, simulando come fatto in precedenza l'input da tastiera per poter lanciare il dado. Dopo di che vengono effettuati due CHECK: si controlla che la posizione di P1 è pari a zero, e che la posizione di P2 sia maggiore o uguale a 1. Siccome P1 è soggetto a scambio l'unico giocatore con cui può interagire è P2 la cui posizione è zero.

Infine, nell'ultimo testcase "check Serpente" viene controllato il corretto funzionamento della funzione movimento quando un giocatore atterra sulla casella serpente. Come nei precedenti testcase vengono dichiarare le stesse variabili quindi, P1, P2, players e dom1. In questo caso la variabile tabellone si presenta così:

```
std::vector<std::string> tabellone{"Via", "Serpente", "Serpente", "Serpente",
                                   "Serpente", "Serpente", "Serpente", "Serpente"};
```

In questa maniera viene forzato l'atterraggio del giocatore su una casella Serpente.

Nel subcase "controllare sepernte", esattamente come nel subcase "controllare scambio", viene invocata la funzione movimento su P1, simulando come fatto in precedenza l'input da tastiera per poter lanciare il dado. I CHECK che vengono effettuati in questo caso sono: che la posizione sia maggiore o uguale a 2, e sia minore o uguale a sette. Per il funzionamento della casella serpente, quando un giocatore vi ci atterra procede fino alla prossima casella della stessa tipologia, per cui in questo caso potrà terminare come minimo nella casella due, se ottiene uno dal tiro del dado, o al al massimo sette, se dovesse ottenere sei.

Passiamo a tabellonetest.cpp. Qui abbiamo testato la creazione del tabellone. Anche qui, come in functiontest.cpp sono state riportate le funzioni necessarie per la creazione del tabellone, senza la parte grafica e senza gli output del terminale per gli stessi motivi precedentemente descritti.

Per prima cosa, attraverso il testcase("Check lunghezza tabellone) è stato verificato che il tabellone abbia la giusta lunghezza.

In seguito è stato controllato se il numero di caselle diverse all'interno del vettore sia quello che ci aspettiamo. Per fare ciò è stato utilizzato un ciclo for, il quale scorrendo controlla tutti gli elementi. Attraverso degli if all'interno viene modificato la variabile int conteggioeffetto(al posto di effetto viene scritto il nome della casella) di 1 a seconda dell'elemento del vettore.

Per ultimo è stato verificato che all'interno del tabellone non ci siano due elementi consecutivi uguali. Questo test è stato realizzando creando una variabile bool "nonconsecutivi". Come nel test precedente viene utilizzato un ciclo for, all'interno del quale è presente un if. Se un elemento è uguale al successivo, nonconsecutivi diventa false, altrimentri true. In questo modo basta controllare che alla fine del processo la variabile bool sia sempre impostata su true.

Per avviare il gioco bisogna compilare il codice ed eseguirlo. Abbiamo deciso di usare CMake.

I comandi usati per CMake sono:

cmake --S . -B build –DCMAKE_BUILD_TYPE=Debug

Per entrare nella modalità debug mentre per compilare con l'output verboso, il quale mostra i comandi che vengono eseguiti durante la compilazione è stato usato:

cmake -- build build --verbose

per eseguire il programma è stato usato il comando:

./build/gioco

Mentre per eseguire i doctest:

./build/functions.t per eseguire i test di functiontest.cpp

./build/tabellonetest.t per eseguire i test di tabellonetest.cpp

cmake --build build --target test per eseguire entrambi i test.