Progetto C++

Appiani Andrea 1057683

1 Introduzione

L'applicazione è un semplice gioco basato sull'estrazione di un numero casuale che diversi giocatori devono indovinare annunciando al termine di ogni partita, come vincitori, i giocatori la cui predizione si è avvicinata maggiormente.

2 Codice

2.1 Librerie usate

- `<iostream>` per definire lo standard input/output di C++;
- `<cmath>` per l'estrazione randomica del numero;
- `<ctime>` per variare il seed di randomizzazione in base all'orario;
- `<string>` per utilizzare le stringhe di caratteri;
- `<vector>` per implementare le liste indicizzate di STL;
- `<map>` per implementare le mappe chiave-valore di STL;
- `<algorithm>` per l'ordinamento nelle strutture STL.

2.2 Classi create

- **Persona**
- **GiocatoreAbstract**
- **Giocatore**
- **Casinò**
- **ComparatorByID**
- **ComparatorByWinrate**

2.3 Classe Persona

Contiene tre campi pubblici: nome, cognome, età con il metodo `print()` per stampare queste informazioni a schermo.

2.4 Classe GiocatoreAbstract

Contiene cinque campi privati: vittorie, sconfitte, counter e ID; counter è **statico**, è inizializzato a 1 e viene incrementato alla creazione di ogni nuova istanza. I campi vittorie e sconfitte vengono usati nel metodo `float getWinrate()` ritornando la percentuale di vittorie del giocatore.

2.5 Classe Giocatore

Sotto-classe sia di **Persona** che di **GiocatoreAbstract** eredita in modo pubblico tutti i loro campi e metodi (ridefinendoli tramite *Override*).

2.6 Classe Casinò

Possiede come campi privati due strutture STL: un *vector* di giocatori e una *map* con associazione "giocatore-previsione". Il riempimento del *vector* è affidato al metodo `addCiclo()` che, ricevendo da tastiera i campi di ogni giocatore, aggiunge una nuova istanza effettuando però prima un controllo sull'età inserita.
Il riempimento della *map* è effettuato dal metodo `inserisciPrevisioni()`, il quale si incarica di controllare che i valori inseriti da tastiera siano numeri interi compresi tra 0 e 100.

Il metodo pubblico `gioca()` controlla prima che il *vector* di Giocatori sia vuoto ed in caso contrario procede all'estrazione di un numero casuale, fissando il seed di randomizzazione tramite l'orario attuale: `srand(time(0))`; in questo modo si garantisce un'estrazione diversa ad ogni esecuzione dell'applicazione.
Al termine dell'estrazione dei vincitori verrà chiesto se si desidera giocare ancora mantenendo gli stessi partecipanti, in caso contrario il gioco terminerà eseguendo un *clear()* del vector di giocatori e mostrando una classifica ordinata in base al winrate di ciascuno.

2.7 Classi ComparatorByID e ComparatorByWinrate

Classi che offrono ciascuna un singolo metodo necessario all'ordinamento dei giocatori all'interno delle strutture STL utilizzate.

3 Costrutti C++ Utilizzati

3.1 ADT

L'applicazione è stata divisa in due file (oltre al main): un file header che contiene la definizione di tutte le classi utilizzate in termini di metodi, campi e relazioni di ereditarietà tra esse; un file cpp che importa l'header volto a implementare separatamente tutti i metodi dichiarati nel file hpp.
In questo modo in caso di modifiche all'implementazione non sarà necessario modificare anche l'interfaccia importata dall'utente, la quale allo stesso tempo definisce cosa l'utente può vedere/modificare e cosa rimane nascosto (*Information Hiding*).

3.2 Ereditarietà Multipla

La classe **Giocatore** estende pubblicamente sia la classe **Persona** che **GiocatoreAbstract**. In questo modo ogni istanza giocatore è provvista sia dei campi/metodi generici di una persona che di quelli specifici di un giocatore. Per costruire un oggetto **Giocatore** è prima necessario creare una persona, questo è reso attraverso l'inizializzazione del costruttore:

```
Giocatore(string s1, string s2, int i): Persona(s1,s2,i) { ... }
```

che andrà quindi prima a inizializzare i campi di **Persona** e successivamente quelli di **Giocatore**.

3.3 Classe Astratta

La classe **GiocatoreAbstract** è così chiamata in quanto è caratterizzata da un metodo senza implementazione dichiarato nel seguente modo:

```
virtual float getWinrate() const = 0;
```

Attraverso *=0* si afferma che tale metodo non sarà implementato nel file cpp. Similmente, sia per questa classe che per tutte le altre, è stato dichiarato il distruttore senza implementazione:

```
virtual ~GiocatoreAbstract() = default;
```

Questo perchè nessuna di esse è provvista di strutture dati allocate dinamicamente tramite puntatori che richiedono la liberazione prima che l'oggetto venga distrutto.

3.4 Classi "friend"

GiocatoreAbstract contiene diversi campi privati (come ad esempio i contatori *Vittorie* e *Sconfitte*) che però devono essere utilizzati da altre classi esterne. Per questo motivo si è scelto di mantenere i campi privati, quindi nascosti ad accessi esterni, dichiarando invece esplicitamente quali classi ne potranno liberamente fare uso nel seguente modo:

```
friend class ComparatorByID;
friend class ComparatorByWinrate;
friend class Casino;
friend class Giocatore;
```

3.5 Overload dei metodi

Le classi **Persona** e **Giocatore** sono provviste di tre costruttori in Overload tra loro; il primo riceve tutti i parametri necessari, il secondo è un *Copy-Constructor* che riceve il riferimento di un altra istanza della stessa classe, mentre il terzo non riceve alcun parametro ed inizializza i campi a valori di default. Ad esempio nella classe **Persona** si ha:

```
Persona(string, string, int);
Persona(const Persona&);
Persona();
```

Nonostante i *Copy-Constructor* non siano esplicitamente chiamati all'interno dell'applicazione, essi vengono automaticamente invocati quando, all'interno di una funzione che riceve un'istanza di **Persona** o **Giocatore**, i parametri attuali vengono copiati in quelli formali.

3.6 Override dei metodi virtuali

I metodi virtuali si hanno solamente nelle classi **Persona** e **GiocatoreAbstract** in quanto dovranno essere ereditati e sovrascritti nella sottoclasse **Giocatore**. Un metodo che richiede l'essere dichiarato *virtual* è il distruttore di entrambe in quanto, nonostante questo non sia il caso, **Giocatore** deve essere in grado di deallocare eventuali strutture dinamiche ereditate dalle super classi. Si ha un esempio di Override per il metodo *getWinrate()*:

```
GiocatoreAbstract:
    virtual float getWinrate() const = 0;

Giocatore:
    float getWinrate() const {
        return (vittorie==0 && sconfitte==0)? 0 : (float)vittorie/(vittorie+sconfitte);}
```

3.7 Metodi e Parametri "const"

Tutti i metodi che non modificano i campi della propria classe sono stati dichiarati *const*; ad esempio `getNome()`, `getCognome()` e `getEta()` nella classe **Persona** devono solamente ritornare il valori di questi campi. Similmente per quanto riguarda i parametri usati dai metodi sono stati dichiarati *const* quelli che, all'interno del metodo, vengono letti ma non modificati; è stato il caso per i *copy-constructor* e per i metodi che fanno da "comparatore" tra due oggetti.

3.8 Facade e Singleton

La classe **Casinò** è stata realizzata tramite il paradigma *Facade* in quanto permette, attraverso una semplice interfaccia, accesso ad altre classi con interfacce più varie ed eventualmente più complesse; in

questo caso si occupa di gestire nell'ordine corretto i metodi delle classi **Giocatore, ComparatorByID e ComparatorByWinrate** in base al procedimento del gioco.
Allo stesso tempo **Casinò** sfrutta il paradigma *Singleton* realizzato con la seguente implementazione:

```
private:
    Casino(){}
    static Casino* casino
public:
    static Casino* getCasino()
```

Implementazione nella quale il metodo *getCasino()* inizializza, se necessario, l'istanza di **Casinò** puntata dal campo statico "casino" ritornandone poi l'indirizzo, così da essere utilizzato nel Main per invocare i metodi della classe.

3.9 Strutture STL e Algoritmi

All'interno della classe **Casinò** sono state implementate due strutture STL:

- `Vector<Giocatore>` per salvare i partecipanti della sessione corrente;
- `Map<Giocatore,int,ComparatorByID>` per salvare le coppie "giocatore-previsione" della sessione corrente.

Il motivo per cui l'uso della sola struttura *map* non sarebbe stato sufficiente è perchè avendo l'oggetto Giocatore come campo chiave esso sarebbe stato utilizzabile solamente in lettura.

In questa applicazione è stato necessario avere due metodi di ordinamento dei giocatori: uno è tramite ID mentre un altro si basa sul *Winrate* (dal maggiore al minore). Per questo motivo non sarebbe stato sufficiente fare overload dell'operatore `"<"` della classe Giocatore ma si è scelto di creare le due classi a sé stanti **ComparatorByID** e **ComparatorByWinrate**. Le classi implementano, rispettivamente:

```
bool operator()(const Giocatore& g1, const Giocatore& g2) const
    return g1.myID < g2.myID;

bool operator()(const Giocatore& g1, const Giocatore& g2) const
    return g1.getWinrate() > g2.getWinrate();
```

Il primo comparatore è necessario per mantenere la *map* ordinata e per permettere l'assegnamento `map[oggetto_chiave]=valore`; in particolare nella prima dichiarazione della *map* in **Casinò** è stato specificato il campo **ComparatorByID** perchè avendo come chiave un oggetto personalizzato serve specificare il suo metodo di ordinamento all'interno della struttura dati. L'altro metodo di sorting dichiarato è implementato tramite la funzione `std::sort()`:

```
sort(giocatori.begin(), giocatori.end(), ComparatorByWinrate());
```

ed è utilizzato per stampare i giocatori in ordine di winrate al termine della sessione.

E' stato poi implementato un algoritmo di ricerca:

```
Giocatore* trovaGiocatore(int ID)
```

il quale esegue una ricerca tramite ID all'interno del *Vector*; tale metodo è necessario in quanto non potendo modificare i campi (vittorie e sconfitte in particolare) nella *map*, l'uso di quest'ultima si limita a trovare l'ID dei giocatori interessati e, attraverso questo metodo, si otterrà il riferimento al giocatore modificabile nel *Vector*.
Infine per le iterazioni sulle STL sono stati usati sia normali *vector::iterator* e *map::iterator* che le loro varianti *const_iterator*.