Programmazione 2
e Laboratorio di Programmazione

Corso di Laurea in

Informatica

Università degli Studi di Napoli “Parthenope”

Anno Accademico 2023-2024

Prof. Luigi Catuogno

1

Informazioni sul corso

Docente	Luigi Catuogno `luigi.catuogno@uniparthenope.it`
Orario	Lun: 9:00-11:00 Mer: 11:00-13:00
Sede	Centro Direzionale Napoli **Aula Magna**
Ricevimento	Mer: 14:00-16:00 (previo appuntamento) Ufficio docente oppure Team: **cxxa3bo**

2

Libri di testo

Introduzione al linguaggio – costrutti e tecniche di base

[FdP] H. M. Deitel, P. J. Deitel

C++ Fondamenti di programmazione

II ed. (2014) Maggioli Editore (Apogeo Education)

ISBN: 978-88-387-8571-9

3

Libri di testo

Tecniche avanzate e strutture dati elementari

[TAP] H. M. Deitel, P. J. Deitel

C++ Tecniche avanzate di programmazione

II ed. (2011) Maggioli Editore (Apogeo Education)

ISBN: 978-88-387-8572-6

4

Risorse on-line

Team del corso

Programmazione 2 AA 2023-24 - Prof. Catuogno
Comunicazioni, incontri e avvisi per il corso
Codice: **ftomzjx**

Piattaforma e-learning

Programmazione II e Laboratorio di Programmazione II - A.A. 2023-24
Materiale didattico, manualistica, esercitazioni.
URL: https://elearning.uniparthenope.it/course/view.php?id=2386

5

Le `class` in C++

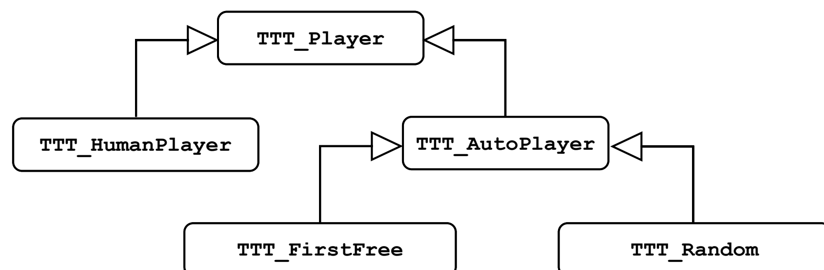
6

Polimorfismo

7

Esercizio: *Tic-Tac-Toe #2 (il trailer...)*

In vista della prossima versione del Tic-Tac-Toe visto in precedenza, si abbozzi la seguente gerarchia di classi che implementano diverse tipologie di giocatore...



Si assuma che i metodi delle classi accedano alla matrice **`playground`** e implementino il metodo **`bool move(…)`**

8

Esercizio: *Tic-Tac-Toe #2 (il trailer…)*

Suggerimento: è più comodo rappresentare le mosse dei giocatori con una classe a se stante: **TTT_move**…

File: **TTT_Players.hpp**

```
 6  class TTT_Move {
 7  public:
 8          int row;
 9          int col;
10          void TTT_Move(int i, int j): row(i), col(j) {}
11  };
```

9

Esercizio: *tic-tac-toe #2*

Scrivere una classe **TTT_Player**: classe base della gerarchia.

E' necessario che ciascuna istanza di giocatore abbia, tra gli attributi privati, un riferimento/puntatore al playground su cui sta giocando:

```
int playerID;
tttPlayGround *match;
```

Tra i metodi: il costruttore, eventuali metodi d'accesso e il metodo che calcola la mossa da giocare (restituisce **false** se non può/vuole muovere)…

```
TTT_Player(ID, match);
int getID();
bool move(TTT_move&);
```

10

Esercizio: *tic-tac-toe #2*

Scrivere una classe **`TTT_Player`**: classe base della gerarchia.

E' necessario che ciascuna istanza di giocatore abbia, tra gli attributi privati, un riferimento/puntatore al playground su cui sta giocando:

```
int playerID;
tttPlayGround *match;
```

Tra i metodi: il costruttore, eventuali metodi d'accesso e il metodo che calcola la mossa da giocare (restituisce **`false`** se non può

```
TTT_Player(ID, match);
int getID();
bool move(TTT_move&);
```

Ricordiamo che questa classe non fornisce alcuna reale strategia di gioco. La utilizziamo solo per definire una interfaccia comune a tutti i giocatori.

In altre parole *«qui, non sappiamo ancora cosa mettere»* nel metodo move() che dovrà essere reimplementato in override da tutte le altre classi derivate.

11

Esercizio: *Tic-Tac-Toe #2*

File: **`TTT_Players.hpp`**

```
…   …
13  class TTT_Player{
14  protected:
15          int playerID;
16          tttPlayGround *match;
17  public:
18          TTT_Player(int ID, tttPlayGround *m): playerID(ID), match(m) {};
19
20          int getID() const {
21                  return playerID;
22          }
23          virtual bool move(TTT_Move&) = 0;
24          // restituisce false se non vuole/puo' muovere.
25  };
…   …
```

12

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
…   …
13  class TTT_Player{
14  protected:
15          int playerID;
16          tttPlayGround *match;
17  public:
18          TTT_Player(int ID, tttPlayGround *m): playerID(ID), match(m) {};
19
20          int getID() const {
21                  return playerID;
22          }
23          virtual bool move(TTT_Move&) = 0;
24          // restituisce false se non vuole/[...]overe.
25  };
…   …
```

Un metodo virtuale dichiarato in questo modo è detto *metodo virtuale puro* e non deve essere implementato nella classe base, ma solo dalle classi derivate in override.

La presenza di questo metodo fa della classe **TTT_Player** una *classe astratta*.

13

Esercizio: *tic-tac-toe #2*

Scrivere una classe **TTT_HumanPlayer**: derivata dalla classe **TTT_Player**.

Assumiamo che il giocatore decida la sua mossa valutando lo schema visualizzato sullo schermo.

E' necessario implementare il metodo **move** in modo che chieda all'utente l'immissione di riga e colonna della sua mossa, ed eventualmente gestisca il suo desiderio di abbandonare la partita.

```
TTT_HumanPlayer(ID, match);
bool move(TTT_move&) override;
```

14

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
 …  …
27  class TTT_HumanPlayer: public TTT_Player {
28  public:
29          TTT_HumanPlayer(int ID, tttPlayGround *m): TTT_Player(ID,m) {}
30          bool move(TTT_Move&) override;
31  };
 …  …
```

15

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
 1  #include<iostream>
 2  #include<cstdlib>
 3  #include"TTT_Players.hpp"
 4  #include"TTT_PlayGround.hpp"
 5
 6  using namespace std;
 7
 8  bool TTT_HumanPlayer::move(TTT_Move &m) {
 9          int i, j;
10          cout << endl << "Inserisci riga e colonna (-1 -1 per terminare):";
11          cin >> i >> j;
12          if(i==-1)
13                  return false;
14          m.row=i;
15          m.col=j;
16          return true;
17  }
```

16

Esercizio: *tic-tac-toe #2*

Scrivere la classe **TTT_AutoPlayer**, derivata dalla classe **TTT_Player**, che è la classe base per tutte le future classi di giocatori automatici.
Una caratteristica comune a tutti i giocatori automatici è che devono avere, tra i propri attributi, una copia locale della matrice di gioco.

```
char myPlayground[3][3];
```

Assumiamo che questa classe e le derivate, prima di muovere, ottengano la copia aggiornata dello schema mediante il metodo : **bool tttPlayGround::getPlayground(char m[3][3])**

Ancora a questo livello, non è specificata lo modalità in cui la classe effettua la sua mossa. Il metodo **move**, deve restare ancora non implementato…

17

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
…   …
13  class TTT_AutoPlayer: public TTT_Player {
14  protected:
15          char myPlayground[3][3];
16  public:
17          TTT_AutoPlayer(int ID, tttPlayGround *m): TTT_Player(ID,m) {}
18          // virtual bool move(TTT_Move&) override;
19  };
…   …
```

Le classi derivate da una classe astratta, devono obbligatoriamente fornire una implementazione per i metodi virtuali puri ereditati. In caso contrario, sono anch'esse *astratte*.

18

Esercizio: *tic-tac-toe #2*

La classe **`TTT_FirstFree`**, derivata dalla classe **`TTT_AutoPlayer`**, fornisce una implementazione del metodo **`muovi`** ereditato dalla classe base.

```
TTT_FirstFree(int, tttPlayGround*);
bool move(TTT_Move&) override;
```

Il metodo **`move`** di questa classe implementa la strategia *«first free»* per la quale, il giocatore sceglie la prima casella libera nello schema. Se non trova caselle vuote restituisce **`false`**.

19

Esercizio: *tic-tac-toe #2*

```
TTT_FirstFree(int, tttPlayGround*);
bool move(TTT_Move&) override;
```

Il metodo **`move`**:

1. invoca il metodo **`getPlayground(char [3][3])`** della partita puntata dall'attributo **`match`** e gli passa l'indirizzo della copia locale del playground (attributo **`myPlayground`** ereditato dalla classe **`AutoPlayer`**).
2. scorre tale matrice da sinistra a destra e dall'alto in basso, identifica la prima casella libera (il cui contenuto è zero), e sceglie riga e colonna corrispondenti per comporre la mossa.
3. Se non trova caselle vuote restituisce **`false`**.

20

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
43  class TTT_FirstFree: public TTT_AutoPlayer {
44  public:
45          TTT_FirstFree(int ID, tttPlayGround *m): TTT_AutoPlayer(ID,m) {}
46          bool move(TTT_Move&) override;
47  };
```

21

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.cpp**

```
18  bool TTT_FirstFree::move(TTT_Move &m)
19  {
20          if (!match->getPlayground(myPlayground))
21                  return false; // qualcosa è andata storta
22          for(int i=0;i<3;i++)
23                  for(int j=0;j<3;j++)
24                          if(myPlayground[i][j]==0) {
25                                  m.row=i;
26                                  m.col=j;
27                                  return true;
28                          }
29          return false; // nessuna cella libera
30  }
```

22

Esercizio: *tic-tac-toe #2*

La classe **TTT_RandomPlayer**, derivata dalla classe **TTT_AutoPlayer**, fornisce una implementazione del metodo **muovi** ereditato dalla classe base e alcune novità. L'attributo **seed** è un intero che serve a inizializzare il PRNG usato per generare mosse «a caso». Il metodo virtuale **reset(int s)** è impiegato per settare questo attributo e invocare la funzione **srand()**.

```
TTT_RandomPlayer(int, tttPlayGround*);
TTT_RandomPlayer(int, tttPlayGround*, int s);
bool move(TTT_Move&) override;
virtual void reset(int s);
```

Il metodo **reset(int s)** è virtuale poiché da questa classe potremmo derivare ulteriori strategie su base casuale (*e.g.* diversi PRNG)…

23

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.hpp**

```
49  class TTT_RandomPlayer: public TTT_AutoPlayer {
50          int seed;
51          const int maxiter=100000;
52  public:
53          TTT_RandomPlayer(int ID, tttPlayGround *m);
54          TTT_RandomPlayer(int ID, tttPlayGround *m, int s);
55          virtual void reset(int);
```

24

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.cpp**

```
…   …
18  TTT_RandomPlayer::TTT_RandomPlayer(int ID, tttPlayGround *m): TTT_AutoPlayer(ID,m), seed(0)
19  {
20  }
21
22  TTT_RandomPlayer::TTT_RandomPlayer(int ID, tttPlayGround *m, int s): TTT_AutoPlayer(ID,m) {
23          reset(s);
24  }
25
26  void TTT_RandomPlayer::reset(int s) {
27          seed=s;
28          srand(s);
29  }
…   …
```

25

Esercizio: *Tic-Tac-Toe #2*

File: **TTT_Players.cpp**

```
…   …
44  bool TTT_RandomPlayer::move(TTT_Move &m) {
45          int i,j,count=0;
46          if (!match->getPlayground(myPlayground))
47                  return false; // qualcosa è andata storta
48          do {
49                  i=rand()%3;
50                  j=rand()%3;
51                  if(myPlayground[i][j]==0) {
52                          m.row=i;
53                          m.col=j;
54                          return true;
55                  }
56                  count++;
57          } while(count<maxiter);
58          return false;
59  }
…   …
```

26

Esercizio: *tic-tac-toe #2*

La classe **tttPlayGround**, proveniente dalla prima versione del gioco,deve essere leggermente modificata. E' necessario:

a) inserire, tra i nuovi attributi l'array **player** che contiene i puntatori alle istanze delle classi scelte per rappresentare i due giocatori;

b) definire il metodo **getPlayground(char[3][3]);**

c) riscrivere il metodo **muovi()** che non prende più le coordinate della mossa dagli argomenti, ma invoca il metodo **move()** dell'oggetto del giocatore di turno;

27

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

```
1  #ifndef _TTT_PLAYGROUND_HPP_
2  #define _TTT_PLAYGROUND_HPP_
3
4  class TTT_Player;
5  class tttPlayGround {
6  private:
7                  int playground[3][3];
8                  int prossimoG;
9                  int vincitore;
10                 int counter;
11                 const char icons[3]={' ','O','X'};
12                 TTT_Player *player[2];
13
15                 bool check();
…  …
```

28

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

```
16  public:
17              tttPlayGround()
18              {
19                      reset();
20              }
21              void reset()
22              {
23                      for(int i=0;i<3;i++)
24                              for(int j=0;j<3;j++)
25                                      playground[i][j]=0;
26                      prossimoG=1;
27                      vincitore=0;
28                      counter=0;
29              };
30              bool getPlayground(char[3][3]) const;
 …  …
```

29

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

```
16              bool finita()
17              {
18                      return (check() || counter==9);
19              };
20
21              char prossimo()
22              {
23                              return icons[prossimoG];
24              };
25              char vince()
26              {
27                      return icons[vincitore];
28              }
29              // bool muovi(int i,int j);
30              bool muovi(int player)
31              void show();
32  };
33  #endif
```

30

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

```
16                              bool finita()
17                              {
18                                      return (check() || counter==9);
19                              };
20
21                              char prossimo()
22                              {
23                                              return icons[prossimoG];
24                              };
25                              char vince()
26                              {
27                                      return icons[vincitore];
28                              }
29                              //bool muovi(int i,int j);
30                              bool muovi(int player)
31                              void show();
32  };
33  #endif
```

TO BE CONTINUED

31

Ereditarietà e Friendship

- L'*ereditarietà* è una relazione che intercorre tra la classe base e le sue derivate. In essa sono stabilite:
  - I criteri di accesso ai membri della classe base rispetto alle classi derivate;
  - Le regole con cui tali regole si propagano nella *gerarchia*

32

Ereditarietà e Friendship

- La *«friendship»* è una relazione che intercorre tra due specifiche classi. Se una classe **B** è dichiarata *friend* dalla classe **A**:
  - Il codice della seconda (*i.e.* dei suoi metodi), hanno accesso a tutti i membri della prima;
  - Tuttavia…

33

Ereditarietà e Friendship

- La «*friendship*» non è simmetrica:
- La «*friendship*» non è ereditaria:
  - Il fatto che **B** sia friend di **A** non implica che **B** sia friend delle classi **C**,**D**, … derivate di **A**
  - Il fatto che **B** sia friend di **A** non implica che le classi **C**,**D**, … derivate di **B** siano friend di **A**
- La «*friendship*» non è transitiva:
  - Il fatto che **A** sia *friend* di **B** e **B** sia *friend* di **C** non implica che **A** sia *friend* di **C**

34