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

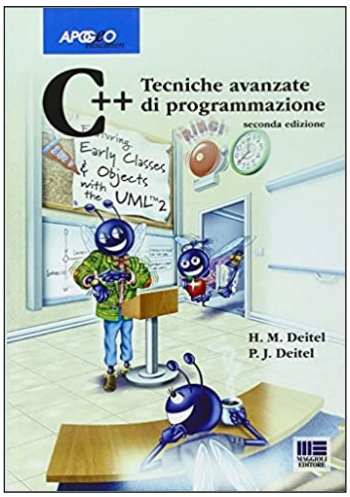
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

Funzioni virtuali «pure»

- Una funzione virtuale «pura» è una funzione membro virtuale dichiarata nella definizione della sua classe, ma per la quale non sarà fornita alcuna implementazione
- L'implementazione di una funzione virtuale pura è demandata alle classi derivate per *override*.

```
class antenata {
    …
    virtual void member_foo(int, …, int) = 0;
    virtual void member_bar(float, …, int) = 0;
};
```

8

Classi *astratte*

- Una classe che contiene almeno una funzione virtuale pura è detta *astratta* e…
  - Non può essere istanziata, né allocata

```
antenata gea;
antenata *rea;

rea=new antenata();
```

NO (antenata gea;)

SI (antenata *rea;)

NO (rea=new antenata();)

Delle classi astratte è possibile dichiarare puntatori e riferimenti, per implementare il polimorfismo.

9

Classi *astratte*

```
class pronipote : public antenata {
   void member_foo(int, …, int) override {
   …;
   }
   void member_bar(float, …, int) override {
   …;
   }
};
```

SI

```
pronipote era, afrodite
antenata *demetra;
demetra= new pronipote();
```

10

Classi *astratte*

- Una classe che derivi da una classe astratta e non ne implementi tutte le funzioni virtuali pure è anch'essa astratta
  - Non può essere istanziata, né allocata

11

Classi *astratte*

- In pratica, una classe astratta non può essere usata direttamente, ma può essere usata come base per definire classi derivate, che hanno il compito di implementare le funzioni pure ereditate.
- Ogni classe derivata implementerà le diverse operazioni, specializzandosi in relazione al tipo di dati impiegato e dalle condizioni di applicazione dei metodi

12

Ereditarietà multipla

13

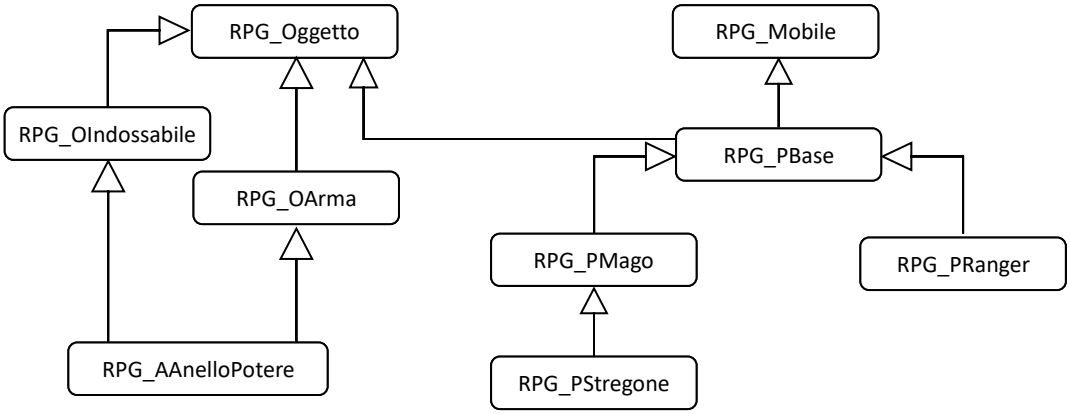
Ereditarietà multipla

- Il meccanismo che permette a una classe di derivare da più classi base, ereditando i membri di entrambe;
- La classe derivata ha una relazione *is-a* con entrambe le classi basi
- Amplia moltissimo le possibilità «espressive» delle gerarchie di classi

  richiede tuttavia una certa disciplina nella progettazione, poiché possono determinarsi ambiguità e comportamenti non facilmente «interpretabili».

14

Esempio: *personaggi e oggetti di un RPG*



15

Ereditarietà multipla

- Date due classi base Base1 e Base2, una classe Derived che derivi da entrambe è dichiarata come segue:

```
Class Base1;
Class Base2;

Class Derivata: public Base1, public Base2 {
   …
};
```

16

Ereditarietà multipla

- Più in generale:

```
class Base1;
class Base2;
…
class BaseN;

class Derivata: <schema1> Base1, <schema2> Base2, …,
<schema N> BaseN {
   …
};
```

Gli schemi di ereditarietà: **public**, **protected** e **private**. Anche diversi tra loro.

17

Esempio: *ereditarietà multipla*

File: **multipla.hpp**

```
…  …
 4  class Base1 {
 5  protected:
 6          int value;
 7  public:
 8          Base1(int parValue) {
 9                  value=parValue;
10          }
11          int getData() {
12                  return value;
13          }
14  };
…  …
```

18

Esempio: *ereditarietà multipla*

File: **multipla.hpp**

```
…  …
15 class Base2 {
16 protected:
17         char letter;
18 public:
19         Base2(char parValue) {
20                 letter=parValue;
21         }
22         char getData() {
23                 return letter;
24         }
25 };
…  …
```

19

Esempio: *ereditarietà multipla*

File: **multipla.hpp**

```
…  …
26 class Derived: public Base1, public Base2 {
27         friend ostream &operator<<(ostream&, const Derived &);
28 private:
29         double real;
30 public:
31         Derived(int,char,double);
32         double getReal() {
33                 return real;
34         }
35 };
…  …
```

20

Esempio: *ereditarietà multipla*

File: **multipla.hpp**

```
…
26 class Derived: public Base1, public Base2 {
27         friend ostream &operator<<(ostream&, const Derived &);
28 private:
29         double real;
30 public:
31         Derived(int,char,double);
32         double getReal() {
33                 return real;
34         }
35 };
…
```

Operatore ridefinito (come funzione globale) per consentire di visualizzare i dati della classe sullo schermo (**cout <<** …)

21

Esempio: *ereditarietà multipla*

File: **multipla.cpp**

```
1  #include<iostream>
2  #include"multipla.hpp"
3  using namespace std;
4
5  Derived::Derived(int i, char c, double d): Base1(i),Base2(c) {
6          real=d;
7  }
8
9  ostream &operator<<(ostream &output, const Derived &derived) {
10         output<<"int:"<<derived.value<<", char:"<< derived.letter;
11         output<<", real:"<<derived.real<<endl;
12         return output;
13 }
```

22

Esempio: *ereditarietà multipla*

File: **multipla.cpp**

```
1  #include<iostream>
2  #include"multipla.hpp"
3  using namespace std;
4
5  Derived::Derived(int i, char c, double d): Base1(i),Base2(c) {
6          real=d;
7  }
8
9  ostream &operator<<(ostream &output, const Derived &derived) {
10         output<<"int:"<<derived.value<<", char:"<< derived.letter;
11         output<<", real:"<<derived.real<<endl;
12         return output;
13 }
```

Invocazione esplicita dei costruttori di entrambe le classi base. **Derived** è una **Base1** ed è anche una **Base2**

23

Esempio: *ereditarietà multipla*

File: **multipla.cpp**

```
1  #include<iostream>
2  #include"multipla.hpp"
3  using namespace std;
4
5  Derived::Derived(int i, char c, double d): Base1(i),Base2(c) {
6          real=d;
7  }
8
9  ostream &operator<<(ostream &output, const Derived &derived) {
10         output<<"int:"<<derived.value<<", char:"<< derived.letter;
11         output<<", real:"<<derived.real<<endl;
12         return output;
13 }
```

Derived eredita sia gli attributi di **Base1** sia quelli di **Base2**. Questi sono stati dichiarati protected (nelle rispettive classi base e lo sono ancora in quella derivata… ma la funzione **<<** è **friend** di **Derived**…

24

Esempio: *ereditarietà multipla*

File: **multimain.cpp**

```
1  #include<iostream>
2  #include"multipla.hpp"
3  using namespace std;
4
5  int main()
6  {
7          Base1 b1(10),*b1Ptr;
8          Base2 b2('Z'), *b2Ptr;
9          Derived d(7,'A',3.5),*dPtr;
10
11         cout << "Base1:"<<b1.getData()<<endl;
12         cout << "Base2:"<<b2.getData()<<endl;
13         cout << "Der. :"<<d<<endl<<endl;
```

```
$ g++ multimain.cpp multipla.cpp -o multip
```

```
$ ./multip
Base1:10
Base2:90
Der. :int:7, char:A, real:3.5
```

25

Esempio: *ereditarietà multipla*

File: **multimain.cpp**

```
14 cout << "Accesso individuale membri di d:"<<endl;
15         cout << "Value: "<<d.Base1::getData()<<endl;
16         cout << "Letter: "<<d.Base2::getData()<<endl;
17         cout << "Real: "<<d.getReal()<<endl<<endl;
18
```

Entrambe le classi base espongono un metodo denominato **getData()**. La classe **Derived** li eredita entrambi.

In questi casi, è obbligatorio rimuovere l'ambiguità indicando classe base di provenienza del membro cui si riferisce nelle espressioni.

Si utilizza l'operatore di scope `::` applicato al nome dell'attributo/metodo ambiguo;

```
Accesso individuale membri di d:
Value: 7
Letter: A
Real: 3.5
```

26

Esempio: *ereditarietà multipla*

File: **multimain.cpp**

```
19  cout << "Accesso mediante puntatori alle basi:"<<end
20          b1Ptr=&d;
21          b2Ptr=&d;
22          dPtr=&d;
23          cout << "b1Ptr->getData(): "<<b1Ptr->getData()<<endl;
24          cout << "b2Ptr->getData(): "<<b2Ptr->getData()<<endl;
25         //cout << "dPtr->getData(): "<<dPtr->getData()<<endl;
26  }
```

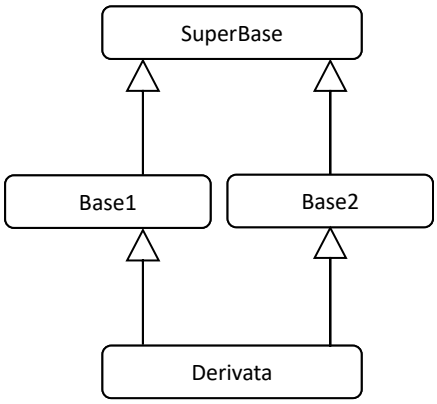
E' naturalmente permesso di associare l'indirizzo di una classe derivata a puntatori a oggetti di ambedue le classi base.

Quando si accede ai membri della derivata mediante puntatori a una delle classi base, è il tipo del puntatore a risolvere l'ambiguità

```
Accesso mediante puntatori alle basi:
Value: 7
Letter: A
$
```

27

Ereditarietà *a diamante…*



- Si crea un potenziale problema per **Derivata**: potrebbe contenere due copie degli stessi membri di **SuperBase**, ereditati rispettivamente tramite **Base1** e **Base2**
- Una situazione di questo tipo può creare errori in fase di compilazione, quando non è possibile risolvere l'ambiguità tra le chiamate.

28

Ereditarietà *a diamante…*

Supponiamo di avere la classe **`SuperBase`** :

```
class SuperBase {
public:
        virtual void print()=0;
};

class Base1: public SuperBase {
public:
       void print() override { … };
};

class Base2: public SuperBase {
public:
        void print() override { … };
};
```

29

Ereditarietà *a diamante…*

E la classe **`Derived`** :

```
class Deriver: public Base1, public Base2 {
public:
        void print() {
               Base1::print();
        }
};
```

30

Ereditarietà *a diamante…*

…

31

Torniamo al Tic-tac-toe

32

Esercizio: *tic-tac-toe #2*

Nella definizione della classe **tttPlayGround** aggiungiamo l'attributo che contiene i puntatori agli oggetti che implementano i giocatori selezionati.

```
TTT_Player *player[2];
```

Questi sono assegnati mediante un nuovo costruttore e da un apposito metodo:

```
tttPlayGround(TTT_Player*, TTT_Player*);
void setPlayer(int, TTT_Player*);
```

Ulteriori nuovi metodi:

```
bool muovi(); // non prende argomenti
bool getPlayground(char[3][3]) const;
```

33

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();
…  …
```

34

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

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

35

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.hpp**

```
32  bool finita()
33                  {
34                          return (check() || counter==9);
35                  };
36
37                  char prossimo()
38                  {
39                                  return icons[prossimoG];
40                  };
41                  char vince()
42                  {
43                          return icons[vincitore];
44                  }
45                  bool muovi();
46                  void show();
47  };
48  #endif
```

36

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.cpp**

```
 8  tttPlayGround::tttPlayGround(TTT_Player *one
 9          player[0]=one;
10          player[1]=two;
11          reset();
12  }
13  void tttPlayGround::setPlayer(int no, TTT_Player *p) {
14          if((no==0)||(no==1))
15                  player[no]=p;
16  }
17
 …  …
```

37

Esercizio: *tic-tac-toe #2*

File: **TTT_PlayGround.cpp**

```
 …  …
18  bool tttPlayGround::getPlayground(char playersCopy[3][3]) const
19  {
20          if(playersCopy==nullptr)
21                  return false;
22          for(int i=0;i<3;i++)
23                  for(int j=0;j<3;j++)
24                          playersCopy[i][j]=playground[i][j];
25          return true;
26  }
 …  …
```

38

Esercizio: *tic-tac-toe #2*

Schema generale del metodo **`tttPlayGround::muovi()`**:

a) Istanza un oggetto **`TTT_Move`**, che utilizza per ricevere la mossa dai giocatori.

b) Il *giocatore corrente* è indicato dalla posizione del suo oggetto nell'array **`player`** ed è dato dall'espressione **`prossimoG-1`**

c) Invoca il metodo **`move`** del giocatore corrente, se questo restituisce **`true`**, ne attua la mossa e procede.

d) Aggiorna il giocatore di turno, incrementa il contatore delle mosse, etc. come nella versione precedente.

39

Esercizio: *tic-tac-toe #2*

File: **`TTT_PlayGround.cpp`**

```
…
…
bool tttPlayGround::muovi() {
        int numplayer;
        TTT_Move m(0,0);
        bool res;
        numplayer=prossimoG-1;
        if(!player[numplayer]->move(m))
                return false;

        if (playground[m.row][m.col]!=0||counter==9)
                return false;
        playground[m.row][m.col]=prossimoG;
        counter++;
        prossimoG=3-prossimoG;
        check();
        return true;
}
```

40

Esercizio: *tic-tac-toe #2*

Schema generale del lfunzione **`main()`**:

a) Il programma chiede all'utente di stabilire la tipologia i giocatori 1 e 2. Istanzia la classe del giocatore richiesto (in caso di **`RandPlayer`** chiede il seed) assegnandone i puntatori ai rispettivi elementi dell'array **`tttPlayGround::player[]`**;

b) la partita è divisa in *round*, composti da una mossa ciascuno dei giocatori. Prima di ciascuna mossa, ai giocatori viene mostrato lo schema di gioco (metodo **`tttPlayGround::show()`**)

c) La partita procede fino a quando non sussistono: vittoria di uno dei giocatori, pareggio o abbandono.

41

Esercizio: *tic-tac-toe #2*

File: **`TTT_main.cpp`**

```
 1  #include<iostream>
 2  #include"TTT_PlayGround.hpp"
 3  #include"TTT_Players.hpp"
 4  using namespace std;
 5
 6  int main()
 7  {
 8          tttPlayGround match;
 9          char scelta;
10          int seed,round=0;
11          bool mossaOk;
12
13          cout << "*** Tic-tac-toe v2 ***"<<endl;
14          for(int i=0;i<2;i++) {
15                  do {
16                          cout<<" player("<<i<<"): ";
17                          cout<<" (h)uman, (f)irstfree, (r)andom, e(x)it "<<endl;
18                          cin>> scelta;
```

42

Esercizio: *tic-tac-toe #2*

File: **TTT_main.cpp**

```
19      switch(scelta) {
20              case 'h':
21                      match.setPlayer(i,new TTT_HumanPlayer(i,&match));
22                      scelta=0;
23                      break;
24              case 'f':
25                      match.setPlayer(i,new TTT_FirstFree(i,&match));
26                      scelta=0;
27                      break;
28              case 'r':
29                      cout << " seed:"; cin >>seed;
30                      match.setPlayer(i,new TTT_RandomPlayer(i,&match,seed));
31                      scelta=0;
32                      break;
33              case 'x':
34                      exit(0);
35      }
36    } while (scelta!=0);
37  } // end for(int i=0)
```

43

Esercizio: *tic-tac-toe #2*

File: **TTT_main.cpp**

```
 …  …
38        do {
39                  cout << "Round:"<<round<<endl;
40                  match.show();
41                  cout<<"Player "<<round%2<<":"<<endl;
42                  mossaOk=match.muovi();
43                  if(!mossaOk){
44                          cout << "Richiesta terminazione"<<endl;
45                          break;
46                  }
47                  if(match.finita())
48                          break;
 …  …
```

44

Esercizio: *tic-tac-toe #2*

File: **`TTT_main.cpp`**

```
 …  …
49                  match.show();
50                  cout<<"Player "<<(round%2+1)<<":"<<endl;
51                  mossaOk=match.muovi();
52                  if (!mossaOk){
53                          cout << "Richiesta terminazione"<<endl;
54                          break;
55                  }
56
57                  round++;
58          } while(!match.finita());
59
60          match.show();
61          cout <<"Vince:"<<match.vince()<<endl;
62  }
```

45

Esercizio: *tic-tac-toe #2 (il sequel…)*

Implementare la versione *«arena»* del gioco in cui:

- sono giocate un numero arbitrario di partite tra due giocatori automatici a scelta (*e.g.* 1000 partite tra **`FirstFree`** e **`Random`**);
- al termine, il programma visualizza il numero di partite vinte dal primo, quelle vinte dal secondo e quelle finite in parità.

46