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

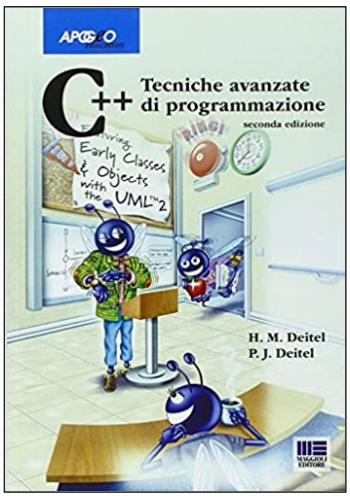
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

Polimorfismo

Il *polimorfismo* è la proprietà dei linguaggi orientati agli oggetti che permette di scrivere codice *generale*, concepito per essere utilizzato nello stesso modo con dati di tipo differente.

- In relazione alle gerarchie di classi, permette di scrivere metodi che trattano tutti gli oggetti delle classi che vi appartengono, come se fossero istanze della classe base (relazione *is-a*)
- In altre parole, i metodi *polimorfici* implementano una funzionalità *comune* a tutte le classi della gerarchia.
- Questo obiettivo è raggiunto mediante l'uso di puntatori e riferimenti alle classi.

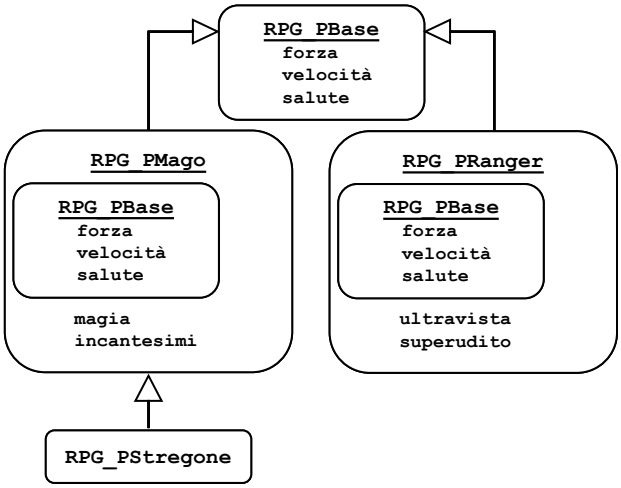
8

Polimorfismo

Ricordiamo che in una gerarchia di classi, vige la relazione *is-a* quindi:

ogni istanza di una delle classi derivate è in definitiva *anche* una istanza della classe base (poiché la contiene);

tutti i metodi della classe base sono ereditati dalle classi derivate, essi costituiscono una *interfaccia* comune a tutti i membri della gerarchia.



9

Esempio: *personaggi di un RPG*

```
1  #include<iostream>
2  #include"RPG_PBase.hpp"
3  using namespace std;
4
5  int main(){
6          RPG_PBase *p;
7          RPG_PBase pedone(5,1,5);
8          RPG_PMago merlino(1,5,5,5,5);
9
10         cout << "pedone: ";
11         p=&pedone;
12         p->show();
13         cout << endl;
14
15         cout << "merlino: ";
16         p=&merlino;
17         p->show();
18         cout<< endl;
19         merlino.show();
20         cout << endl;
21 }
```

Puntatore alla *classe base*.

Istanza della *classe base*.

Assegno il puntatore all'istanza e attraverso di lui invoco il metodo **show()**

È eseguito il metodo **show()** della classe base.

```
pedone: [F:5, V:1, S:5]
...
```

10

Esempio: *personaggi di un RPG*

```
1  #include<iostream>
2  #include"RPG_PBase.hpp"
3  using namespace std;
4
5  int main(){
6          RPG_PBase *p;
7          RPG_PBase pedone(5,1,5);
8          RPG_PMago merlino(1,5,5,5,5);
9
10         cout << "pedone: ";
11         p=&pedone;
12         p->show();
13         cout << endl;
14
15         cout << "merlino: ";
16         p=&merlino;
17         p->show();
18         cout<< endl;
19         merlino.show();
20         cout << endl;
21 }
```

Puntatore alla *classe base*.

Istanza della *classe derivata*.

Assegno il puntatore all'istanza e attraverso di lui invoco il metodo **`show()`**

È eseguito il metodo **`show()`** della classe base.

```
pedone: [F:5, V:1, S:5]
merlino: [F:1, V:5, S:5]
…
```

11

Esempio: *personaggi di un RPG*

```
1  #include<iostream>
2  #include"RPG_PBase.hpp"
3  using namespace std;
4
5  int main(){
6          RPG_PBase *p;
7          RPG_PBase pedone(5,1,5);
8          RPG_PMago merlino(1,5,5,5,5);
9
10         cout << "pedone: ";
11         p=&pedone;
12         p->show();
13         cout << endl;
14
15         cout << "merlino: ";
16         p=&merlino;
17         p->show();
18         cout<< endl;
19         merlino.show();
20         cout << endl;
21 }
```

Istanza della *classe derivata*.

Invoco il metodo **`show()`** attraverso l'istanza della classe derivata

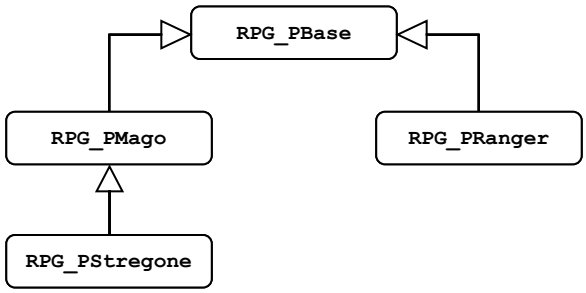
È eseguito il metodo **`show()`** della classe derivata.

```
pedone: [F:5, V:1, S:5]
merlino: [F:1, V:5, S:5]
[F:1, V:5, S:5] (Mag:5, Inc:5)
```

12

Esempio: *personaggi di un RPG*

Componiamo un *team* arbitrario di personaggi e calcoliamo il punteggio *forza* totale del team



13

Esempio: *personaggi di un RPG*

File: **RPG_Team.cpp**

```
 4  #include"RPG_PBase.hpp"
 5  #define COMPNUM  3
 6
 7  int main(){
 8          int seed,f,s,v,i,m,mn,uv,uu,forzatot=0;
 9          char scelta;
10          RPG_PBase *team[COMPNUM];
11          cout << "Seed: ";
12          cin>>seed;
13          srand(seed);
14
15          cout << "*** Componi la tua squadra: ***"<<endl;
16          for(int tmbr=0;tmbr<COMPNUM;tmbr++) {
17                  f=rand()%10; s=rand()%10; v=rand()%10;
18                  i=rand()%10; m=rand()%10; uv=rand()%10;
19                  uu=rand()%10; mn=rand()%10;
20
21                  cout<<"(b)ase, (m)ago, (r)anger, (s)tregone: ";
22                  cin >> scelta;
…   …
```

14

Esempio: *personaggi di un RPG*

File: **RPG_Team.cpp**

```
23          switch (scelta) {
24                          case 'b':
25                                  team[tmbr]=new RPG_PBase(f,v,s);
26                                  break;
27                          case 'm':
28                                  team[tmbr]=new RPG_PMago(f,v,s,m,i);
29                                  break;
30                          case 'r':
31                                  team[tmbr]=new RPG_PRanger(f,v,s,uv,uu);
32                                  break;
33                          case 's':
34                                  team[tmbr]=new RPG_PStregone(f,v,s,m,i,mn);
35                                  break;
36                  }
37          }
…   …
```

15

Esempio: *personaggi di un RPG*

File: **RPG_Team.cpp**

```
…   …
38          cout << "Condizioni fisiche dei personaggi: "<<endl;
39
40          for(int tmbr=0;tmbr<COMPNUM; tmbr++) {
41                  cout <<"["<<tmbr<<"]:";
42                  team[tmbr]->show();
43
44                  forzatot+=team[tmbr]->getForza();
45                  cout<<endl;
46          }
47          cout << "Totale forza: "<<forzatot<<endl;
48  }
```

16

Esempio: *personaggi di un RPG*

L'array di puntatori

```
RPG_PBase *team[COMPNUM];
```

e' popolato dinamicamente con una scelta arbitraria di istanze di classi appartenenti alla gerarchia.

Possiamo gestire «collettivamente» oggetti diversi, quando siamo interessati esclusivamente alle loro caratteristiche comuni

17

…su puntatori e reference

- Si possono assegnare indirizzi (o riferimenti) di una classe derivata a un puntatore (o riferimento) della classe base.

```
RPG_PBase *bPtr, base;
RPG_PMago mago, *mPtr;
bPtr=&mago;
```

- Non si può fare il contrario…

```
mPtr=&base;
```

18

…su puntatori e reference

- Si possono assegnare indirizzi (o riferimenti) di una classe derivata a un puntatore (o riferimento) della classe base.

```
RPG_PBase *bPtr, base;
RPG_PMago mago, *mPtr;
bPtr=&mago;
```

- Attraverso di esso si possono invocare i metodi della classe base…

```
mPtr.setForza(11);
```

- NON si possono invece invocare i metodi della classe derivata …

```
mPtr.setMagia(42);
```

19

Ereditarietà e polimorfismo…

L'ereditarietà consente di costruire una gerarchia di classi in cui ciascuna fornisce strumenti *specializzati* per un determinato ambito di applicazione:

- La *definizione* in maniera *incrementale* di nuovi metodi;
- La *ridefinizione* dei metodi ereditati per «calare» nello specifico caso di applicazione, un metodo già definito a livello più generale.

con puntatori e riferimenti, si può lavorare con diverse classi di una stessa gerarchia, utilizzando solo i metodi ereditati dalla classe base.

20

Funzioni membro **`virtual`**

Le classi della gerarchia ridefiniscono i metodi ereditati per implementare soluzioni *specializzate* a task comuni. (*e.g.* le diverse classi del RPG ridefiniscono il metodo **`show()`**) .

I *metodi virtuali* sono il meccanismo che il C++ offre per :

- Trattare un insieme arbitrario di classi di una gerarchia mediante puntatori e riferimenti a una classe base
- Eseguire di volta in volta l'implementazione *«più specializzata»* dei metodi ereditati dalla classe base. Ad esempio, quella implementata dalla specifica istanza.

21

Funzioni membro **`virtual`**

- La classe base definisce i metodi virtuali mediante il qualificare **`virtual`**

```
class RPG_PBase {
…
    virtual void show();
};
```

- Le classi derivate ereditano il metodo e possono ridefinirlo:

```
class RPG_PMago {
…
    void show() override;
};
```

22

Funzioni membro **`virtual`**

La ridefinizione di un metodo virtuale produce un comportamento detto *override* :

- Il metodo ridefinito dall'istanza corrente *prevale* (*override*) su quello ereditato dalla classe base, anche quando invocato attraverso un puntatore o riferimento alla classe base

23

Funzioni membro **`virtual`**

File: **`BaseEDerivate.cpp`**

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

```
class Base {
public:
        int BasePub;
        Base(): BasePub(0) {}
        Base(int i): BasePub(i) {}
        virtual void getBasePub() {
                cout << "(getBasePub) Base: "<<BasePub<<"."<<endl;
        }
        void showBasePub() {
                cout << "(showBasePub) Base: "<<BasePub<<"."<<endl;
        }
        void setBasePub(int i) {
                BasePub=i;
        }
};
```

Metodo *virtual*: si assume che sarà ridefinito e sostituito nelle classi derivate.

Altri metodi della classe base. Seguono le «solite regole» …

24

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

e la classe **Derived1** :

```
class Derived1 : public Base {
public:
        Derived1(): Base(0) {};
        Derived1(int i): Base(i) {};
        void getBasePub() override {
                cout << "(getBasePub) Der1: "<<BasePub<<"."<<endl;
        }
        void showBasePub() {
                cout << "(showBasePub) Der1: "<<BasePub<<"."<<endl;
        }
        void myshow() {
                cout << "(myshow) Der1: "<<BasePub<<"."<<endl;
        }
```

Derived1 *sostituisce* il metodo con la nuova implementazione. E' la ridefinizione di un metodo virtual che determina l'override

Derived1 *ridefinisce* un metodo non virtuale ereditato dalla classe base. No override.

25

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Base***

```
        Base *bPtr;
        Derived1 d1(17), *dPtr;

        bPtr=dPtr=&d1;

        cout<<"bPtr->getBasePub: ";
        bPtr->getBasePub();
        cout<<"bPtr->showBasePub: ";
        bPtr->showBasePub();
        cout<<"bPtr->setBasePub: ";
        bPtr->setBasePub(17);
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Base** e invochiamo il metodo virtuale

```
bPtr->getBasePub: (getBasePub) Der1: 17.
…
```

26

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Base***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;

cout<<"bPtr->getBasePub: ";
bPtr->getBasePub();
cout<<"bPtr->showBasePub: ";
bPtr->showBasePub();
cout<<"bPtr->setBasePub: ";
bPtr->setBasePub(17);
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Base** e invochiamo il metodo non virtuale ma ridefinito

```
bPtr->getBasePub: (getBasePub) Der1: 17.
bPtr->showBasePub: (showBasePub) Base: 17.
…
```

27

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Base***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;

cout<<"bPtr->getBasePub: ";
bPtr->getBasePub();
cout<<"bPtr->showBasePub: ";
bPtr->showBasePub();
cout<<"bPtr->setBasePub: ";
bPtr->setBasePub(17);
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Base** e invochiamo un metodo non ridefinito (solo ereditato)

```
bPtr->getBasePub: (getBasePub) Der1: 17.
bPtr->showBasePub: (showBasePub) Base: 17.
bPtr->setBasePub: (setBasePub) Base: i=17.
```

28

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived1***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(17);
cout<<"d1.showBasePub: ";
d1.showBasePub();
…
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Derived1** e invochiamo il metodo virtuale

```
dPtr->getBasePub: (getBasePub) Der1: 17.
…
```

29

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived1***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(17);
cout<<"d1.showBasePub: ";
d1.showBasePub();
…
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Derived1** e invochiamo il metodo non virtuale ma ridefinito

```
dPtr->getBasePub: (getBasePub) Der1: 17.
dPtr->showBasePub: (showBasePub) Der1: 17.
…
```

30

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived1***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(17);
cout<<"d1.showBasePub: ";
d1.showBasePub();
…
```

Accediamo all'oggetto **d1** mediante il puntatore alla classe **Derived1** e invochiamo un metodo non ridefinito (solo ereditato)

```
dPtr->getBasePub: (getBasePub) Der1: 17.
dPtr->showBasePub: (showBasePub) Der1: 17.
dPtr->setBasePub: (setBasePub) Base: i=17.
…
```

31

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

Accediamo direttamente all'oggetto **d1** e invochiamo un metodo non virtuale ma ridefinito

*classe **Derived1***

```
Base *bPtr;
Derived1 d1(17), *dPtr;

bPtr=dPtr=&d1;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(17);
cout<<"d1.showBasePub: ";
d1.showBasePub();
…
```

```
dPtr->getBasePub: (getBasePub) Der1: 17.
dPtr->showBasePub: (showBasePub) Der1: 17.
dPtr->setBasePub: (setBasePub) Base: i=17.
d1.showBasePub: (showBasePub) Der1: 17.
```

32

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

E ora la classe **Derived2** :

```
class Derived2 : public Derived1 {
public:
        Derived2(): Derived1(0) {};
        Derived2(int i): Derived1(i) {};
        void getBasePub() {
                cout << "(getBasePub) Der2: "<<BasePub<<"."<<endl;
        }
        void showBasePub() {
                cout << "(showBasePub) Der2: "<<BasePub<<"."<<endl;
        }
```

Derived2 *sostituisce* il metodo con la nuova implementazione. E' la ridefinizione di un metodo virtual che determina l'override

Derived2 *ridefinisce* un metodo non virtuale ereditato dalla classe base. No override.

33

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Base***

```
Base *bPtr;
Derived1 *dPtr;
Derived2 d2(18);
…
bPtr=dPtr=&d2;

cout<<"bPtr->getBasePub: ";
bPtr->getBasePub();
cout<<"bPtr->showBasePub: ";
bPtr->showBasePub();
cout<<"bPtr->setBasePub: ";
bPtr->setBasePub(18);
```

Accediamo all'oggetto **d2** mediante il puntatore alla classe **Base** e invochiamo il metodo virtual (override di d1), il metodo non virtuale ma ridefinito in **d2**, e il metodo solo ereditato da **d1**

```
bPtr->getBasePub: (getBasePub) Der2: 18.
bPtr->showBasePub: (showBasePub) Base: 18.
bPtr->setBasePub: (setBasePub) Base: i=18.
```

34

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived1***

```
Base *bPtr;
Derived1 *dPtr;
Derived2 d2(18);
…
bPtr=dPtr=&d2;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(18);
```

Accediamo all'oggetto **d2** mediante il puntatore alla classe **Base** e invochiamo il metodo virtual (override di d1), il metodo non virtuale ma ridefinito in **d2**, e il metodo solo ereditato da **d1**

```
?
?
?
```


Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived1***

```
Base *bPtr;
Derived1 *dPtr;
Derived2 d2(18);
…
bPtr=dPtr=&d2;
…
cout<<"dPtr->getBasePub: ";
dPtr->getBasePub();
cout<<"dPtr->showBasePub: ";
dPtr->showBasePub();
cout<<"dPtr->setBasePub: ";
dPtr->setBasePub(18);
```

Accediamo all'oggetto **d2** mediante il puntatore alla classe **Derived1** e invochiamo il metodo virtual (override in **d1**), il metodo non virtuale ma ridefinito in **d2**, e il metodo solo ereditato da **d1**

```
dPtr->getBasePub: (getBasePub) Der2: 18.
dPtr->showBasePub: (showBasePub) Der1: 18.
dPtr->setBasePub: (setBasePub) Base: i=18.
```

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived2***

```
Base *bPtr;
Derived1 *dPtr;
Derived2 d2(18), *d2Ptr;
…
d2Ptr=&d2;
…
cout<<"d2Ptr->getBasePub: ";
d2Ptr->getBasePub();
cout<<"d2Ptr->showBasePub: ";
d2Ptr->showBasePub();
cout<<"d2Ptr->setBasePub: ";
d2Ptr->setBasePub(18);
```

Accediamo all'oggetto **d2** mediante il puntatore alla classe **Derived2** e invochiamo il metodo virtual (override in **d1**), il metodo non virtuale ma ridefinito in **d2**, e il metodo solo ereditato da **d1**

```
?
?
?
```


Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso mediante puntatore alla classe **Derived2***

```
Base *bPtr;
Derived1 *dPtr;
Derived2 d2(18), *d2Ptr;
…
d2Ptr=&d2;
…
cout<<"d2Ptr->getBasePub: ";
d2Ptr->getBasePub();
cout<<"d2Ptr->showBasePub: ";
d2Ptr->showBasePub();
cout<<"d2Ptr->setBasePub: ";
d2Ptr->setBasePub(18);
```

Accediamo all'oggetto **d2** mediante il puntatore alla classe **Derived2** e invochiamo il metodo virtual (override in **d1**), il metodo non virtuale ma ridefinito in **d2**, e il metodo solo ereditato da **d1**

```
d2Ptr->getBasePub: (getBasePub) Der2: 18.
d2Ptr->showBasePub: (showBasePub) Der2: 18.
d2Ptr->setBasePub: (setBasePub) Base: i=18.
```

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso all'oggetto **b1** mediante puntatore alla classe **Base***

```
Base b1(13),*bPtr;
…
bPtr=&b1;
…
cout<<"bPtr->getBasePub: ";
bPtr->getBasePub();
cout<<"bPtr->showBasePub: ";
bPtr->showBasePub();
```

Accediamo all'oggetto **b1** mediante il puntatore alla classe **Base** e invochiamo il metodo virtual (override in **d1**) e il metodo non virtuale ma ridefinito in **d2**

```
?
?
```

39

Funzioni membro **virtual**

File: **BaseEDerivate.cpp**

… cosa succede nella **main**:

*Accesso all'oggetto **b1** mediante puntatore alla classe **Base***

```
Base b1(13),*bPtr;
…
bPtr=&b1;
…
cout<<"bPtr->getBasePub: ";
bPtr->getBasePub();
cout<<"bPtr->showBasePub: ";
bPtr->showBasePub();
```

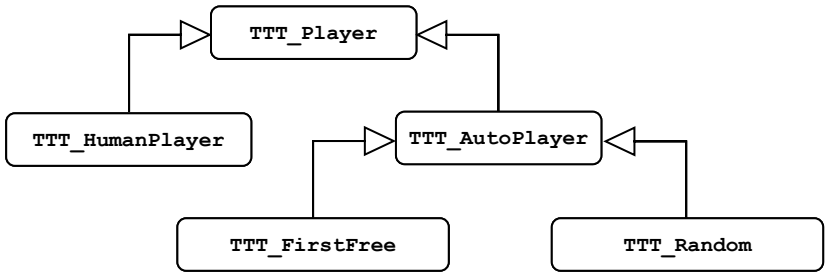
Accediamo all'oggetto **b1** mediante il puntatore alla classe **Base** e invochiamo il metodo virtual (override in **d1**) e il metodo non virtuale ma ridefinito in **d2**

```
bPtr->getBasePub: (getBasePub) Base: 13.
bPtr->showBasePub: (showBasePub) Base: 13.
```

40

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 mia_mossa(int &riga, int &colonna)`**

41