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

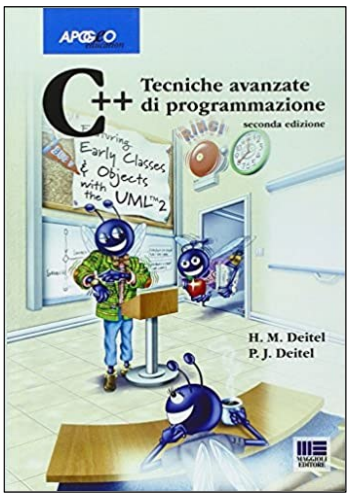
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

Ereditarietà multipla

7

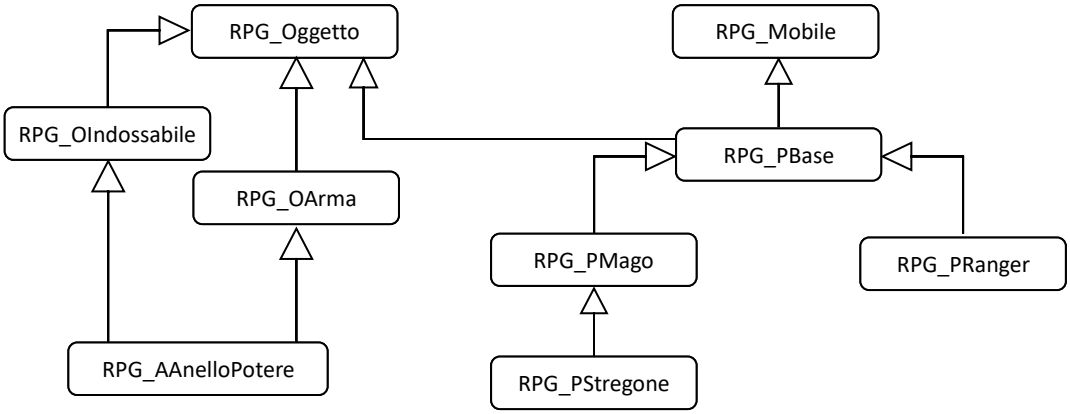
Ereditarietà multipla

- 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».

8

Esempio: *personaggi e oggetti di un RPG*



9

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 {
   …
};
```

10

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.

11

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  };
…  …
```

12

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 };
…  …
```

13

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 };
…  …
```

14

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 <<** …)

15

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  }
```

16

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**

17

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**…

18

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
```

19

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
```

20

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  }
```

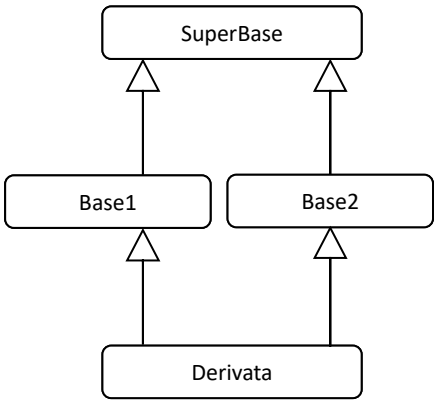
E' naturalmente permesso di associare l'indirizzo di una classe derivata a puntatori a oggetti di ambedue le classi base.

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
$
```

21

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.

22

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 { … };
};
```

23

Ereditarietà *a diamante…*

E la classe **`Derived`** :

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

24

Ereditarietà *a diamante…*

La funzione **`main`** :

```
int main()
{
        Base1 b1;
        Base2 b2;
        Derived d;
        Superbase *pointers[3];

        pointers[0]=&b1;
        pointers[1]=&b2;
        pointers[2]=&d;

        for (int i=0; i<3; i++)
                pointers[i]->print();
}
```

25

Ereditarietà *a diamante…*

La funzione **`main`** :

```
int main()
{
        Base1 b1;
        Base2 b2;
        Derived d;
        Superbase *pointers[3];

        pointers[0]=&b1;
        pointers[1]=&b2;
        pointers[2]=&d;
```

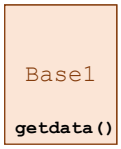
```
$ g++ multiple.cpp -o multiple
multiple.cpp: In function 'int main()':
multiple.cpp:39:22: error: 'Superbase' is an ambiguous base of 'Derived'
   39 |         pointers[2]=&d;
      |                      ^
```

26

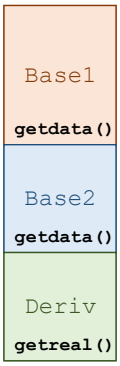
Ereditarietà multipla…

```
class Base1 {
protected:
        int value;
public:
        Base1(int parValue) {
                value=parValue;
        }
        int getData() {
                return value;
        }
};
```

```
class Base2 {
protected:
        char letter;
public:
        Base2(char parValue) {
                letter=parValue;
        }
        char getData() {
                return letter;
        }
};
```




```
class Derived: public Base1,
public Base2 {
        friend ostream &operator…
private:
        double real;
public:
        Derived(int,char,double);
        double getReal() {
                return real;
        }
};
```

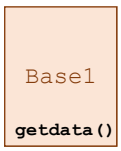


27

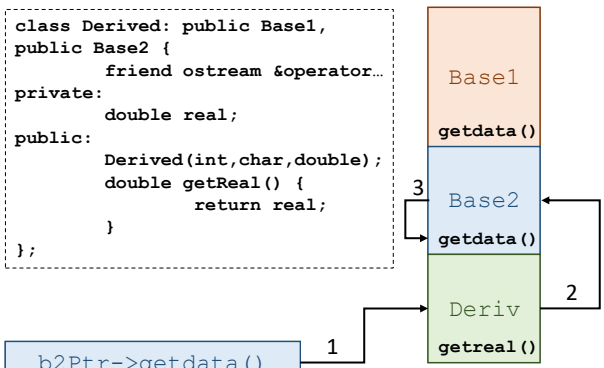
Ereditarietà multipla…

```
class Base1 {
protected:
        int value;
public:
        Base1(int parValue) {
                value=parValue;
        }
        int getData() {
                return value;
        }
};
```

```
class Base2 {
protected:
        char letter;
public:
        Base2(char parValue) {
                letter=parValue;
        }
        char getData() {
                return letter;
        }
};
```

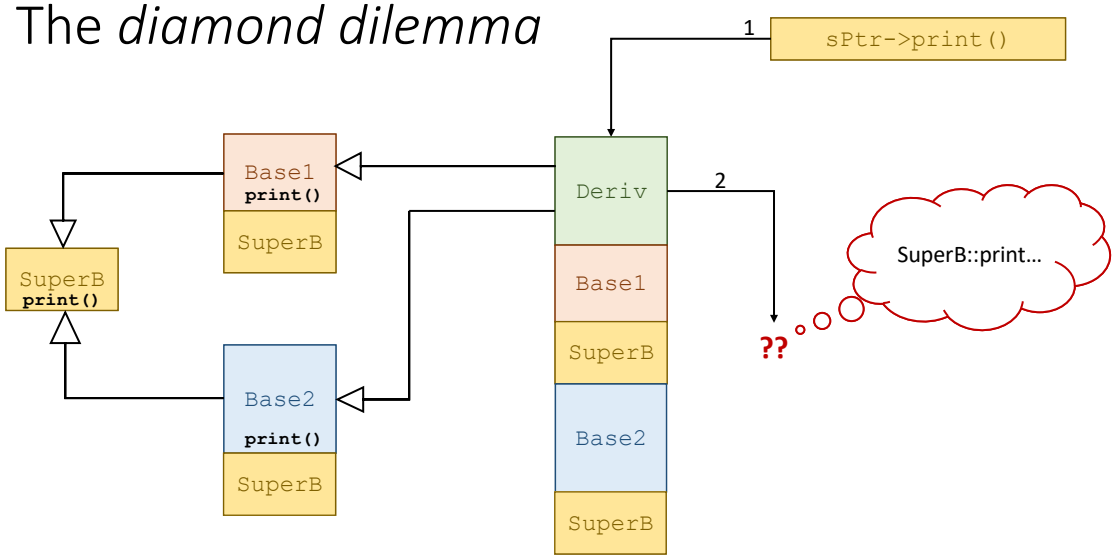



```
class Derived: public Base1,
public Base2 {
        friend ostream &operator…
private:
        double real;
public:
        Derived(int,char,double);
        double getReal() {
                return real;
        }
};
```



28

The *diamond dilemma*



29

Ereditarietà *virtuale*

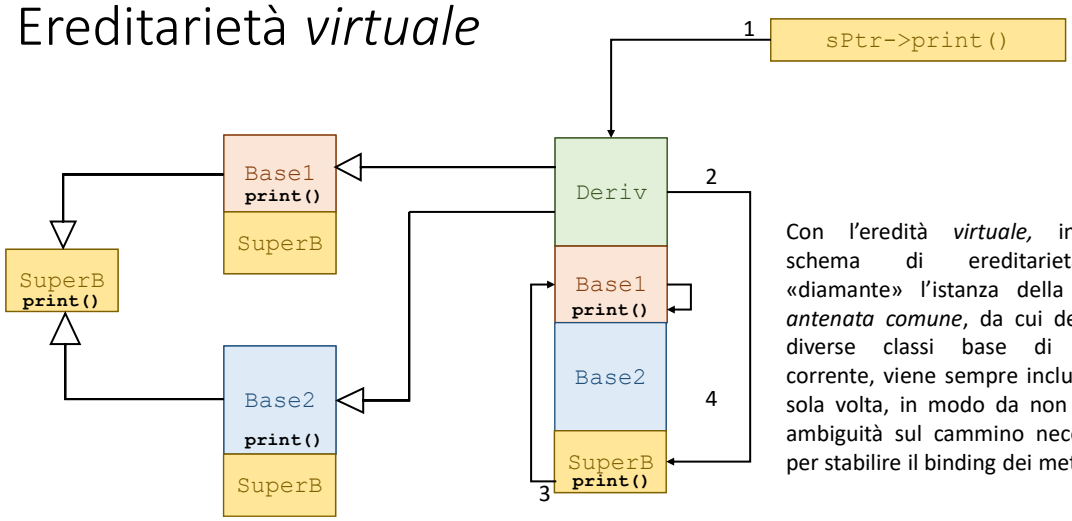
Supponiamo di avere la classe **`SuperBase`** :

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

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

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

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

30

Ereditarietà *virtuale*



Con l'eredità *virtuale,* in uno schema di ereditarietà a «diamante» l'istanza della classe *antenata comune*, da cui derivano diverse classi base di quella corrente, viene sempre inclusa una sola volta, in modo da non creare ambiguità sul cammino necessario per stabilire il binding dei metodi…

31

Ereditarietà *virtuale*

La funzione **`main`** :

```
int main()
{
        Base1 b1;
        Base2 b2;
        Derived d;
        Superbase *pointers[3];

        pointers[0]=&b1;
        pointers[1]=&b2;
        pointers[2]=&d;
```

```
$g++ multiplevirt.cpp -o multiplevirt
$ ./multiplevirt
Base1
Base2
Base1
$
```

32

Distruttori *virtuali*

- Lo standard C++ non definisce il comportamento dei distruttori di una classe derivata, quando invocati implicitamente mediante il puntatore alla classe base.

```
int main()
{
        …
        Base *bPtr;
        Derived d;
        bPtr=&d;
        …
        delete bPtr;
        …
}
```

33

Distruttori *virtuali*

- Per risolvere il problema, è sufficiente dichiarare **`virtual`** il distruttore della classe base

```
class Base {
public:
        Base() {…}
        virtual ~Base() {…}
};
class Derived: public Base {
Public:
        Derived() {…}
        ~Derived() {…}
};
```

34

Distruttori *virtuali*

- Per risolvere il problema, è sufficiente dichiarare **`virtual`** il distruttore della classe base
- Questo rende *virtuali* tutti i distruttori della gerarchia di classe anche se *non hanno tutti lo stesso nome di quello della classe base*
- Quando si applica **`delete`** a un puntatore a **`Base`**, che punta a una istanza di **`Derivata`**, viene invocato sempre il distruttore giusto, e poi a ritroso tutti i precedenti.

35