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

Ereditarietà nella programmazione a oggetti

E' una forma di *riutilizzo del software* basato sullo sviluppo *incrementale* degli oggetti.

Per rispondere a variazioni/estensioni dell'ambito di applicazione del software esistente si procede a:

- Aggiungere le nuove funzionalità affiancando quelle esistenti e ancora necessarie
- Ove necessario alcune funzionalità preesistenti, possono essere ridefinite/estese a seconda delle mutate esigenze
- L'impianto originale e le funzionalità comuni restano inalterate e non necessitano di essere riscritte.

Ereditarietà in C++

Fornisce meccanismi per riutilizzare il codice già scritto in forma di classi, permettendo di definirne di nuove in termini di quelle definite in precedenza.

Più precisamente, data una classe (che diremo *base*) che espone attributi e metodi necessari a determinate applicazioni:

Il C++ permette la definizione di una nuova classe *derivata* (dalla precedente) che *eredita:*

- La sua rappresentazione in memoria
- La sua *interfaccia* (*i.e.* attributi e metodi pubblici)

9

Ereditarietà in C++

il programmatore non deve riscrivere tutto il codice che è in comune tra la classe base e la nuova classe ma può semplicemente:

- ridefinire alcune funzioni membro per estenderle ai nuovi casi di utilizzo (*overload*)
- modificare l'implementazione di alcune funzioni membro (*override*)

…e naturalmente:

- aggiungere nuovi attributi e metodi a quelli ereditati

10

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Supponiamo di avere la classe **prima** :

```
class prima {
        int priv;
public:
        int pub;
        prima() {  priv=pub=0; cout<<"prima()."<<endl;
        }
        ~prima() { cout <<"distruttore prima():"<<endl;
        }
        void setPriv(int i) { priv=i;
                cout<<"prima.setpriv():"<<endl;
        }
        int getPriv() { cout<<"prima.getPriv():"<<endl;
                return priv;}
        int getPub() { cout<<"prima.getPub():"<<endl;
                return pub;
        }
}
```

11

Ereditarietà in C++

La classe **seconda** *deriva* dalla **prima**, (che è la sua *classe base*).

Lo *specificatore di accesso* **public** definisce i criteri di visibilità dei membri della classe base da parte dei metodi della classe derivata.

Definiamo la classe **seconda** :

```
class seconda: public prima {
public:
        int pub2;
        seconda() {
                pub2=1;
                cout<<"seconda():"<<endl;
        }
        ~seconda() { cout <<"distruttore prima():"<<endl; }
        int getPub() {
                cout<<"seconda.getPub():"<<endl;
                return pub+pub2;
        }
        int getPub2() { return pub2; }
}
```

12

Ereditarietà in C++

La classe **seconda** *deriva* dalla **prima**, (che è la sua *classe base*).

Lo *specificatore di accesso* **public** definisce i criteri di visibilità dei membri della classe base da parte dei metodi della classe derivata.

Definiamo la classe **seconda** :

```
class seconda: public prima {
public:
	int pu
	second

	}
	~secon
	int ge
		return pub+pub2;
	}
	int getPub2() { return pub2; }
}
```

Lo schema public è quello più permissivo e prevede che ai metodi della classe derivata:

- *sia concesso* accedere ai membri public della classe base (*i.e.* leggere e modificare il valore degli attributi, invocare i metodi)
- *non sia concesso* accedere ai membri private della classe base

Si ricordi che *una classe derivata non è automaticamente friend della sua classe base*

13

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Definiamo la classe **seconda** :

```
class seconda: public prima {
public:
	int pub2;
	seconda() {
		pub2=1;
		cout<<"seconda():"<<endl;
	}
	~seconda() { cout <<"distruttore prima():"<<endl; }
	int getPub() {
		cout<<"seconda.getPub():"<<endl;
		return pub+pub2;
	}
	int getPub2() { return pub2; }
}
```

Nuovi membri della classe **seconda**: *si aggiungono* a quelli ereditati dalla classe base.

…

14

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Definiamo la classe **seconda** :

```
class seconda: public prima {
public:
        int pub2;
        seconda() {
                pub2=1;
                cout<<"seconda():"<<endl;
        }
        ~seconda() { cout <<"distruttore prima():"<<endl; }
        int getPub() {
                cout<<"seconda.getPub():"<<endl;
                return pub+pub2;
        }
        int getPub2() { return pub2; }
}
```

Costruttore e distruttore della classe **seconda**

Metodo della classe derivata che *ridefinisce* quello *ereditato* dalla classe base (ha la stessa firma)

15

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Utilizziamo le classi:

```
{
        …
        prima a;
        cout << a.getPriv()<<endl;
        cout << a.getPub()<<endl;
        …
```

Invoca il costruttore della classe **prima**

```
prima().
prima.getPriv():
0
Prima.getPub():
0
```

16

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Il costruttore della classe **seconda** invoca anche il costruttore della classe base

Utilizziamo le classi:

```
{
    …
    seconda b;
    …
```

```
prima().
seconda():
```

Il costruttore della classe derivata invoca quello della classe base perché questo allochi e inizializzi come previsto, tutti i membri di questa (inclusi quelli privati) affinché possano essere ereditati.

Il codice restante, può inizializzare i nuovi membri e, nell'ambito delle regole di accessibilità, modificare i membri della classe base.

17

Ereditarietà in C++

```
class prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int getPub() { … }
}
```

```
class seconda: public prima {
    public:
            int pub2;
    seconda() { … }
    ~seconda() { … }
    int getPub() { … }
    int getPub2() { … }
}
```

18

Ereditarietà in C++

```
class prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int getPub() { … }
}
```

```
class seconda: public prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int pub2;
        seconda() { … }
        ~seconda() { … }
        int getPub() { … }
        int getPub2() { … }
}
```

19

Ereditarietà in C++

```
class prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int getPub() { … }
}
```

```
class seconda: public prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int pub2;
        seconda() { … }
        ~seconda() { … }
        int getPub() { … }
        int getPub2() { … }
}
```

Accessibile solo attraverso i metodi ereditati dalla classe **prima**

Invocati implicitamente dal costruttore e dal distruttore della classe **seconda**

20

Ereditarietà in C++

```
class prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int getPub() { … }
}
```

```
class seconda: public prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int pub2;
        seconda() { … }
        ~seconda() { … }
        int getPub() { … }
        int getPub2() { … }
}
```

Attributi ereditati: se non ridefiniti, mantengono stessi tipo, identificatori, etc. Eventualmente inizializzati dal costruttore della classe base.

Metodi ereditati (se non ridefiniti, il codice è quello definito nella classe base…)

21

Ereditarietà in C++

```
class prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int getPub() { … }
}
```

```
class seconda: public prima {
        int priv;
public:
        int pub;
        prima() { … }
        ~prima() { … }
        void setPriv(int i) { … }
        int getPriv() { … }
        int pub2;
        seconda() { … }
        ~seconda() { … }
        int getPub() { … }
        int getPub2() { … }
}
```

Metodo ridefinito. Se invocato come membro della classe b, sostituisce quello ereditato dalla classe base.

22

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Utilizziamo le classi:

```
{
  …
  seconda b;
  …

  b.setPriv(100);
  cout << b.getPriv() <<endl;
```

```
prima().
seconda():
…
prima.setPriv():
prima.getPriv(): 100
…
```

L'invocazione dei metodi ereditati comporta l'esecuzione del codice definito nella classe base (con i rispettivi privilegi).

Si noti che i membri **private** ereditati sono comunque membri della classe derivata (sebbene «invisibili» dai suoi metodi *nativi*). L'istanza **b** ha un membro privato **priv** *«nella sua area di memoria»* e può utilizzarlo con gli appositi metodi di accesso…

23

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Utilizziamo le classi:

```
{
  …
  seconda b;
  …

  b.setPriv(100);
  cout << b.getPriv() <<endl;
  b.pub=b.pub2=10;
  cout << b.getPub() << endl;
```

```
prima().
seconda():
…
prima.setPriv():
prima.getPriv(): 100
…
seconda.getPub(): 11
```

L'accesso agli attributi pubblici ereditati avviene in maniera del tutto analoga ai membri introdotti dalla nuova classe.

Invocando il metodo **b.getPub()**, si invoca il codice introdotto nella definizione della classe **seconda**

Se si invoca un metodo ridefinito, questo *prevale* su quello ereditato (con la stessa firma).

24

Ereditarietà in C++

File: **PrimaESeconda.cpp**

Utilizziamo le classi:

```
{
        …
        seconda b;
        …

        b.setPriv(100);
        cout << b.getPriv() <<endl;
        b.pub=b.pub2=10;
        cout << b.getPub() << endl;
        …
}
```

```
prima().
seconda():
…
prima
prima
…
seconda      Pub(): 11
…
distruttore seconda():
distruttore prima():
distruttore prima():
```

Il distruttore della classe derivata, invoca quello della classe base per le operazioni di deallocazione previste per i membri ereditati.

Il distruttore dell'oggetto a (di classe prima) creato all'inizio.

25

Alcuni esempi…

26

Esempio: *punto*

Scrivere la classe **`punto`** con gli attributi:

```
double x; double y;
```

e con i metodi:

```
punto(); punto(double c1, double c2);
double getX(); double getY();
punto &setX(double nx);
punto &setY(double ny);
double distanza_origine();
```

27

Esercizio: *punto*

File: **`punto.hpp`**

```
1  class punto {
2          double x;
3          double y;
4  public:
5          punto ();
6          punto (double c1, double c2);
7          double getX() const;
8          double getY() const;
9          punto &setX(double);
10         punto &setY(double);
11         double distanza_origine() const;
12         void show();
13 };
```

28

Esercizio: *punto*

File: **punto.cpp**

```
1  #include"punto.hpp"
2
3  punto::punto() {
4                  x=0; y=0;
5  }
6  punto::punto(double c1, double c2){
7                  x=c1; y=c2;
8  }
9
10 double punto::getX() const {
11                 return x;
12 }
13 double punto::getY() const {
14                 return y;
15 }
…  …
```

29

Esercizio: *punto*

File: **punto.cpp**

```
…  …
16 punto& punto::setX(double nx) {
17                 x=nx;
18                 return *this;
19 }
20
21 punto& punto::setY(double ny) {
22                 y=ny;
23                 return *this;
24 }
25
26 double punto::distanza_origine(){
27                 return sqrt(pow(x,2)+pow(y,2));
28 }
29 void punto::show() {
30                 cout<<"["<<x<<";"<<y<<"]";
31 }
```

Restituiscono il riferimento a un oggetto di classe **punto**

30

Esercizio: *punto*

File: **main.cpp**

```
…    …
5    int main()
6    {
7            punto punti[9];
8            int pos=0;
9
10           for(int i=0;i<3;i++)
11                   for(int j=0;j<3;j++)
12                           punti[pos++].setX(i).setY(j);
13
14           for(int i=0;i<9;i++){
15                   punti[i].show();
16                   cout<<":"<<punti[i].distanza_origine()<<endl;
17           }
18   }
```

```
$ g++ punto.cpp main.cpp -o punto1
$
```

31

Esercizio: *punto3D #1*

Scrivere la classe **punto3D** che derivi dalla **punto** e che aggiunga la terza coordinata:

```
double z;
```

i nuovi metodi:

```
double getZ(); punto3D &setZ(double);
```

…i costruttori e ridefinisca i metodi:

```
double distanza_origine(); void show();
```

32

Esercizio: *punto3D #1*

File: **punto3D.hpp**

```
1  #ifndef _PUNTO_3D_HPP_
2  #define _PUNTO_3D_HPP_
3  #include"punto.hpp "
4
5  class punto3D: public punto {
6          double z;
7  public:
8          punto3D();
9          punto3D(double c1,double c2,double c3);
10         double getZ() const;
11         punto3D &setZ(double);
12         void show() const;
13         double distanza_origine() const;
14 };
15 #endif
```

33

Esercizio: *punto3D #1*

File: **punto3D.cpp**

```
1  #include <iostream>
2  #include <cmath>
3  #include "punto3D.hpp"
4  using namespace std;
5
6  punto3D::punto3D(): punto(), z(0) {}
7
8  punto3D::punto3D(double c1,double c2,double c3):punto(c1,c2){
9                  z=c3;
10 }
11 double punto3D::getZ() const {
12                 return z;
13 }
14 punto3D &punto3D::setZ(double nz) {
15                 z=nz;
16                 return *this;
17 }
```

Inizializza i membri ereditati dalla classe punto . Questo invoca il costruttore **`punto::punto(double,double)`**

Restituisce il riferimento a un oggetto di classe **`punto3D`**

34

Esercizio: *punto3D #1*

File: **punto3D.cpp**

```
18 double punto3D::distanza_origine() const{
19         return sqrt(pow(getX(),2)+pow(getY(),2)+pow(z,2));
20 }
21
22 void punto3D::show() const {
23         std::cout << "("<<getX()<<","<<getY()<<","<< z <<")";
24 }
```

35

Esercizio: *punto3D #1*

File: **main3D.cpp**

```
1  #include<iostream>
2  #include<cstdlib>
3  #include"punto3D.hpp"
4  using namespace std;
5  int main() {
6          punto3D punti3[9];
7          int pos=0;
8          int seed;
9
10         cout << "Seed: ";
11         srand(seed);
12         for(int i=0;i<3;i++)
13                 for(int j=0;j<3;j++)
14                         punti3[pos++].setX(i).setY(j).setZ(rand()%3);
15         for(int i=0;i<9;i++){
16                 punti3[i].show();
17                 cout<<":"<<punti3[i].distanza_origine()<<endl;
18         }
20 }
```

```
$ g++ punto.cpp punto3D.cpp main3D.cpp -o punto3D
$
```

36

Esercizio: *punto3D #1*

File: **main3D.cpp**

```
1  #include<iostream>
2  #include<cstdlib>
3  #include"punto3D.hpp"
4  using namespace std;
5  int main() {
6          punto3D punti3[9];
7          int pos=0;
8          int seed;
9
10         cout << "Seed: ";
11         srand(seed);
12         for(int i=0;i<3;i++)
13                 for(int j=0;j<3;j++)
14                         punti3[pos++].setZ(rand()%3).setX(i).setY(j);
15         for(int i=0;i<9;i++){
16                 punti3[i].show();
17                 cout<<":"<<punti3[i].distanza_origine()<<endl;
18         }
20 }
```

Domanda:
Perché scritta nel modo seguente dà errore?
punti3[pos++].setX(i).setY(j).setZ(rand()%3);

Esempio: *personaggi di un RPG*

I giocatori di un RPG sono di tipo diverso però condividono un insieme ristretto di attributi e capacità che definiscono un «personaggio base»

La classe **RPG_PBase** fornisce gli attributi:

```
int forza; int velocita; int salute;
```

E i metodi:

```
RPG_PBase(int f, int v, int s);
int getForza(); int getVelocita(); int getSalute();
void setForza(int f); void setVelocita(int v);
void setSalute(int s);
void show();
```

Esempio: *personaggi di un RPG*

File: **RPG_PBase.hpp**

```
1  #ifndef _PERSONAGGIO_HPP_
2  #define _PERSONAGGIO_HPP_
3
4  class RPG_PBase {
5          int forza;
6          int velocita;
7          int salute;
8  public:
9          RPG_PBase(int f, int v, int s);
10         int getForza();
11         int getVelocita();
12         int getSalute();
13         void setForza(int f);
14         void setVelocita(int v);
15         void setSalute(int s);
16         void show();
17 };
18
19 #endif
```

39

Esempio: *personaggi di un RPG*

File: **RPG_PBase.cpp**

```
1  #include<iostream>
2  using std::cout;
3
4  #include"RPG_PBase.hpp"
5
6  RPG_PBase::RPG_PBase(int f, int v, int s) {
7                  forza=f; velocita=v; salute=s;
8  };
9
10 int RPG_PBase::getForza(){
11                 return forza;
12 }
13 int RPG_PBase::getVelocita(){
14                 return velocita;
15 }
16 int RPG_PBase::getSalute(){
17                 return salute;
18 }
```

40

Esempio: *personaggi di un RPG*

File: **RPG_PBase.cpp**

```
19 void RPG_PBase::setForza(int f){
20         forza=f;
21 }
22 void RPG_PBase::setVelocita(int v){
23         velocita=v;
24 }
25 void RPG_PBase::setSalute(int s){
26         salute=s;
27 }
28 void RPG_PBase::show() {
29         cout << "[F:"<<forza<<", V:"<<velocita<<", S:"<< salute<<"]";
30 }
```


Esercizio: *personaggi di un RPG*

Dal «personaggio base» derivano tutti i personaggi del gioco, ciascuno caratterizzato da attributi e capacità proprie. Ad esempio, i «maghi»…

La classe **RPG_PMago** aggiunge gli attributi:

```
int magia; int incantesimi;
```

i metodi:

```
RPG_PMago(int f, int v, int s, int m, int i);
int getMagia(); int getIncantesimi();
void setMagia(int m); void setIncantesimi(int i);
```

E ridefinisce il metodo **show()**

Esercizio: *personaggi di un RPG*

File: **RPG_PBase.hpp**

```
20  class RPG_PMago: public RPG_PBase
21  {
22          int magia;
23          int incantesimi;
34  public:
25          RPG_PMago(int f, int v, int s, int m, int i);
26          int getMagia();
27          void setMagia(int m);
28          int getIncantesimi();
29          void setIncantesimi(int i);
30          void show();
31  };
```

43

Esercizio: *personaggi di un RPG*

File: **RPG_PBase.cpp**

```
…   …
31  RPG_PMago::RPG_PMago(int f, int v, int s, int m, int i);
32
33  RPG_PMago::getMagia() {
34          return magia;
35  }
36  void RPG_PMago::setMagia(int m) {
37          magia=m;
38  }
39  int RPG_PMago::getIncantesimi() {
40          return incantesimi;
41  }
42  void RPG_PMago::setIncantesimi(int i) {
43          incantesimi=i;
44  }
```

44

Esercizio: *personaggi di un RPG*

File: **RPG_PBase.cpp**

```
44  void RPG_PMago::show(){
45          RPG_PBase::show();
46          cout << " (Mag:"<<magia<<", Inc:"<<incantesimi<<")";
47  }
```

La classe **RPG_PMago** ridefinisce il metodo **show**. Pertanto, questo è il codice che verrà eseguito quando il metodo è invocato su un'istanza di **RPG_PMago**.

Tuttavia, se le funzionalità offerte dal metodo **show** della classe base sono ancora richieste, non è necessario riscriverle. E' possibile invocare in maniera esplicita il metodo **show()** della classe **RPG_PBase**. Questo metodo ha accesso ed opera sui membri che **RPG_PMago** ha ereditato da questa.

45

Esercizio: *personaggi di un RPG*

File: **RPG_main.cpp**

```
19  #include<iostream>
20  #include"RPG_PBase.hpp"
21  using namespace std;
22
23  int main(){
24          RPG_PBase pedone(5,1,5);
25          RPG_PMago merlino(1,5,5,5,5);
26
27          cout << "pedone:  ";
28          pedone.show();
29          cout << endl;
30          cout << "merlino:  ";
31          merlino.show();
32          cout<< endl;
33  }
```

46

Esercizio: *personaggi di un RPG*

Partendo dalle classi **RPG_PBase** e **RPG_PMago** scrivere le seguenti classi:

La classe **RPG_PRanger** aggiunge al personaggio base gli attributi:

```
int ultravista; int superudito;
```

La classe **RPG_PStregone** aggiunge al mago base l'attributo:

```
int magiaNera;
```

Entrambi ridefiniscono il metodo **show()**

47

Esercizio: *personaggi di un RPG*

File: **RPG_PEroi.hpp**

```
34  class RPG_PRanger: public RPG_PBase {
35          int ultravista;
36          int superudito;
37  public:
38          RPG_PRanger(int f, int v, int s, int uv, int su);
39          int getUV();
40          void setSU();
41          int getUV();
42          void setSU();
43          void show();
44  };
45
```

48

Esercizio: *personaggi di un RPG*

File: **RPG_PEroi.hpp**

```
34  class RPG_PStregone: public RPG_PMago {
35          int magiaNera;
36  public:
37          RPG_PStregone(int f, int v, int s, int m, int i, int mn);
38          int getMN();
39          void setMN();
40          void show();
41  };
42
```

49

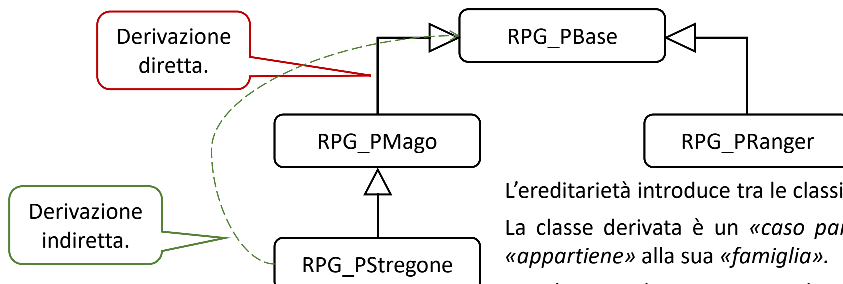
Esercizio: *personaggi di un RPG*

File: **RPG_PEroi.cpp**

```
44  void RPG_PStregone::RPG_PStregone(int f, int v, int s, int m, int i, int mn):
45          RPG_PMago(f,v,s,m,i), magiaNera(mn) {
46  }
47
48  Void RPG_PStregone::show() {
49          RPG_PMago::show();
50          cout << endl << " (MNera:" << magiaNera << ")";
51  }
```

50

Esempio: *personaggi di un RPG*



L'ereditarietà introduce tra le classi la relazione di derivazione: *is-a*

La classe derivata è un «*caso particolare*» della classe base ma «*appartiene*» alla sua «*famiglia*».

Qui diciamo che una istanza di **PMago** *è* una istanza di **PBase**, una istanza di **PStregone** è un'istanza di **PMago**,e anche un'istanza di **PBase**.

PStregone e **PRanger** sono entrambe **PBase**, ma solo il primo è **PMago**

51

Ereditarietà: costruttori e distruttori

Il costruttore di una classe derivata, *prima* di eseguire il suo codice deve eseguire il costruttore della sua classe base. Può farlo in due modi:

- Invocando *implicitamente* il costruttore di default della classe base;
- Invocando *esplicitamente* l'*inizializzatore* di uno i più membri della classe base, o il costruttore stesso della classe base

52

Ereditarietà: costruttori e distruttori

Il costruttore di una classe derivata, *prima* di eseguire il suo codice deve eseguire il costruttore della sua classe base. Può farlo in due modi:

- Invocando *implicitamente* il costruttore di default della classe base;
- Invocando *esplicitamente* l'*inizializzatore* di uno i più membri della classe base, o il costruttore stesso della classe base

```
class seconda: public prima {
public:
        int pub2;
        seconda()  {
                pub2=1;
                cout<<"seconda():"<<endl;
        }
```

53

Ereditarietà: costruttori e distruttori

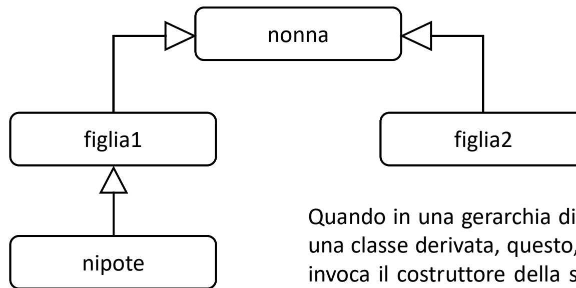
Il costruttore di una classe derivata, *prima* di eseguire il suo codice deve eseguire il costruttore della sua classe base. Può farlo in due modi:

- Invocando *implicitamente* il costruttore di default della classe base;
- Invocando *esplicitamente* l'*inizializzatore* di uno i più membri della classe base, o il costruttore stesso della classe base

```
class punto3D: public punto {
        …
        public:
        punto3D(double c1,double c2,double c3):punto(c1,c2){
                z=c3;
        }
```

54

Ereditarietà: costruttori e distruttori



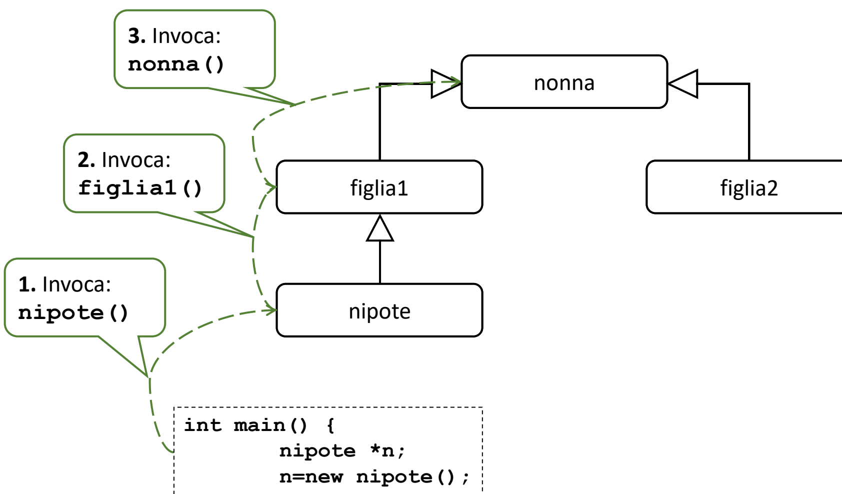

```
int main() {
        nipote *n;
        n=new nipote();
```

Quando in una gerarchia di classi, si invoca il costruttore di una classe derivata, questo, *prima* di eseguire il suo codice, invoca il costruttore della sua classe base (se questa è a sua volta derivata, il processo si ripete fino alla «capostipite»)

Questo perché a ciascun livello della gerarchia, un costruttore inizializza *solo* i membri che introduce la sua classe, *dopo* che tutto quanto ereditato dalle classi precedenti sia già stato allocato.

55

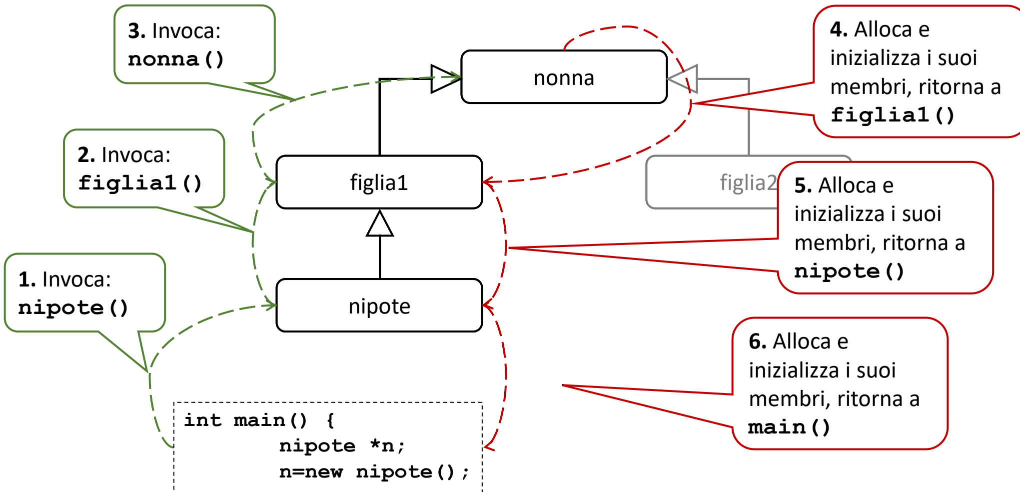
Ereditarietà: costruttori e distruttori




```
int main() {
        nipote *n;
        n=new nipote();
```

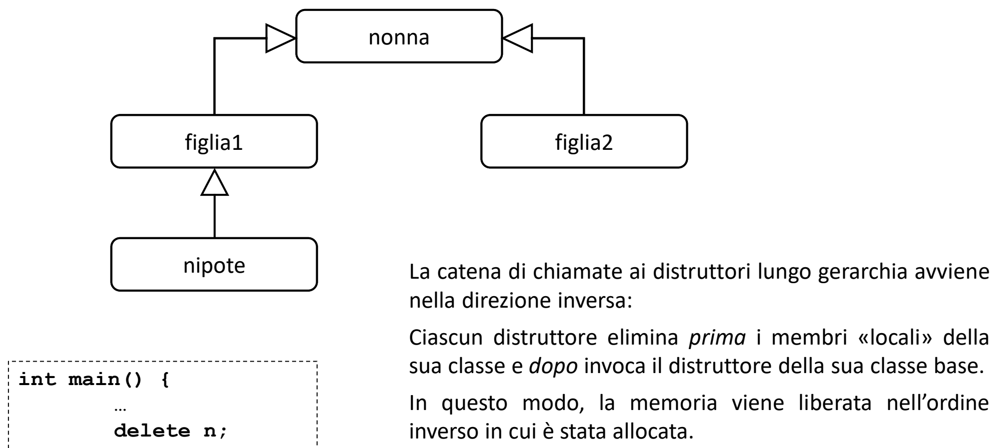
56

Ereditarietà: costruttori e distruttori



57

Ereditarietà: costruttori e distruttori



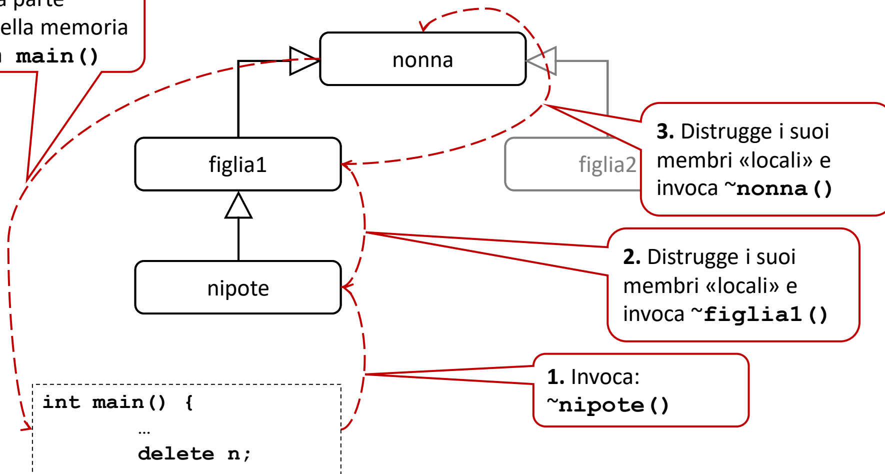
La catena di chiamate ai distruttori lungo gerarchia avviene nella direzione inversa:

Ciascun distruttore elimina *prima* i membri «locali» della sua classe e *dopo* invoca il distruttore della sua classe base.

In questo modo, la memoria viene liberata nell'ordine inverso in cui è stata allocata.

58

Ereditarietà: costruttori e distruttori



59

Classi **C++**: i membri **protected**

- Proteggere i membri privati dalle classi derivate ha senso quando si vogliono consentire *specializzazioni* della classe base purché queste non stravolgano i principi di funzionamento stabiliti.
- Per questa ragione, alle classi derivate è consentito l'uso di tali membri solo attraverso le funzioni di accesso eventualmente fornite allo scopo.
- Tuttavia, in molte circostanze una simile politica risulta eccessivamente restrittiva. A tale scopo il C++ fornisce un qualificatore d'accesso specifico per le relazione di derivazione

60

Classi **C++**: i membri **protected**

I qualificatori di accesso dei membri di una classe C++ sono:

- **`public:`**
  Membri accessibili sia dai metodi della classe, sia dal codice esterno ad essa;
- **`private:`**
  Membri accessibili esclusivamente ai metodi della classe (e dalle funzioni e i metodi delle classi dichiarate come **`friend`**). I membri privati sono inaccessibili anche alle classi derivate.
- **`protected:`**
  Membri accessibili esclusivamente ai metodi della *classe base*, ai metodi delle *classi derivate* e dai funzioni e metodi delle classi **`friend`**.

61

Classi **C++**: i membri **protected**

```
1  class Base {
2          int MyPrivateInt;
3  protected:
4          int MyProtectedInt;
5  public:
6          int MyPublicInt;
7
8  public:
9    int foo1()  { return MyPrivateInt;}
10   int foo2()  { return MyProtectedInt;}
11   int foo3()  { return MyPublicInt;}
12 };
…  …
```

62

Classi **C++**: i membri **protected**

```
1   class Base {
2           int MyPrivateInt;
3   protected:
4           int MyProtectedInt;
5   public:
6           int MyPublicInt;
…   …
12  };
13
14  class Derived : public Base {
15  public:
16     int foo1()  { return MyPrivateInt;}
17     int foo2()  { return MyProtectedInt;}
18     int foo3()  { return MyPublicInt;}
19  };
…   …
```

Errore in fase di compilazione: **MyPrivateInt** è privato in **Base** e quindi non è visibile in questo ambito.

63

Classi **C++**: i membri **protected**

```
1   class Base {
2           int MyPrivateInt;
3   protected:
4           int MyProtectedInt;
5   public:
6           int MyPublicInt;
…   …
12  };
13
14  class Derived : public Base {
15  public:
16     int foo1()  { return MyPrivateInt;}
17     int foo2()  { return MyProtectedInt;}
18     int foo3()  { return MyPublicInt;}
19  };
…   …
```

OK

64

Classi **C++**: i membri **protected**

```
 1  class Base {
 2          int MyPrivateInt;
 3  protected:
 4          int MyProtectedInt;
 5  public:
 6          int MyPublicInt;
 …  …
12  };
 …  …
20  class Unrelated {
18  private:
19    Base B;
20    Derived D;
21  public:
22    int foo1()   { return B.MyPrivateInt;}
23    int foo2()   { return B.MyProtectedInt;}
24    int foo3()   { return D.MyProtectedInt;}
25    int foo4()   { return D.MyPublicInt;}
26  };
```

Errore in fase di compilazione: **MyPrivateInt** è privato in **Base**, pertanto risulta inaccessibile da codice esterno all'istanza **B**

65

Classi **C++**: i membri **protected**

```
 1  class Base {
 2          int MyPrivateInt;
 3  protected:
 4          int MyProtectedInt;
 5  public:
 6          int MyPublicInt;
 …  …
12  };
 …  …
20  class Unrelated {
18  private:
19    Base B;
20    Derived D;
21  public:
22    int foo1()   { return B.MyPrivateInt;}
23    int foo2()   { return B.MyProtectedInt;}
24    int foo3()   { return D.MyProtectedInt;}
25    int foo4()   { return D.MyPublicInt;}
26  };
```

Errore in fase di compilazione: **MyProtectedInt** è protetto in **Base**, pertanto risulta inaccessibile da codice esterno all'istanza **B**

66

Classi **C++**: i membri **protected**

```
1   class Base {
2           int MyPrivateInt;
3   protected:
4           int MyProtectedInt;
5   public:
6           int MyPublicInt;
…   …
12  };
…   …
20  class Unrelated {
18  private:
19    Base B;
20    Derived D;
21  public:
22    int foo1()  { return B.MyPrivateInt;}
23    int foo2()  { return B.MyProtectedInt;}
24    int foo3()  { return D.MyProtectedInt;}
25    int foo4()  { return D.MyPublicInt;}
26  };
```

Errore in fase di compilazione: **MyProtectedInt** è protetto in **Base**, pertanto risulta inaccessibile da codice esterno all'istanza **D**

67

Classi **C++**: i membri **protected**

```
1   class Base {
2           int MyPrivateInt;
3   protected:
4           int MyProtectedInt;
5   public:
6           int MyPublicInt;
…   …
12  };
…   …
20  class Unrelated {
18  private:
19    Base B;
20    Derived D;
21  public:
22    int foo1()  { return B.MyPrivateInt;}
23    int foo2()  { return B.MyProtectedInt;}
24    int foo3()  { return D.MyProtectedInt;}
25    int foo4()  { return D.MyPublicInt;}
26  };
```

OK

68

Tipi di ereditarietà in C++

Tipi di ereditarietà in C++

In C++, la relazione di derivazione tra classi esprime anche uno «schema» di accesso che indica in che modo, i qualificatori di accesso dei membri della classe base si *propagano* nelle classi derivate:

```
class CDerivata: <schema> CBase {
     public:
          void coutProt();
          void coutPub();
};
```

Può essere: **public**, **protected** o **private**.

Tipi di ereditarietà in C++

Definiamo la classe **CBase** …

```
class CBase {
     int iPrivato;
protected:
     int iProtetto;
     void coutPriv();
public:
     int iPubblico;
};
```

71

Tipi di ereditarietà in C++

… e la classe **CDerivata** :

```
class CDerivata: public CBase {
     public:
          void coutProt();
          void coutPub();
};
```

72

Tipi di ereditarietà in C++

Lo schema **`public`** è di gran lunga quello più utilizzato:

In CBase, il membro è:	In CDerivata «diventa»:	Regole d'accesso…
`public`	**`public`**	accessibile direttamente a tutte le funzioni membro, le funzioni **`friend`** e le funzioni non membro
`protected`	**`protected`**	Accessibile direttamente a tutte le funzioni membro e le funzioni **`friend`**
`private`	*nascosti*	Accessibili a funzioni membro e funzioni **`friend`** tramite funzioni membro **`public`** e **`protected`** della classe base

73

Tipi di ereditarietà in C++

Quindi…

```
int main() {
    CDerivata x;
    cout << x.iPubblico << endl;
    x.coutPriv();
    x.coutProt();
    …
```

SI. Membro **`public`** di **`CBase`**.

NO. Membro **`protected`** di **`CBase`**. Accessibile solo dai metodi di **`CDerivata`**

SI. Membro **`public`** di **`CDerivata`**. Il suo codice può visualizzare **`iProtetto`**, che è un membro **`protected`** di **`CBase`**

74

Tipi di ereditarietà in C++

… e la classe **CDerivata2** :

```
class CDerivata2: protected CBase {
     public:
          void coutProt();
          void coutPub();
};
```


Tipi di ereditarietà in C++

Lo schema **protected**:

In CBase, il membro è:	In CDerivata «diventa»:	Regole d'accesso…
public	**protected**	Accessibile direttamente a tutte le funzioni membro e le funzioni **friend**
protected	**protected**	Accessibile direttamente a tutte le funzioni membro e le funzioni **friend**
private	nascosti	Accessibili a funzioni membro e funzioni **friend** tramite funzioni membro **public** e **protected** della classe base

Tipi di ereditarietà in C++

Quindi…

```
int main() {
      CDerivata2 x;
      cout << x.iPubblico << endl;
      x.coutPriv();
      x.coutProt();
      …
```

NO. Membro **`public`** di **`CBase`**, diventa **`protected`** di **`CDerivata2`**.

NO. Membro **`protected`** di **`CBase`**. Accessibile solo dai metodi di **`CDerivata2`**

SI. Membro **`public`** di **`CDerivata2`**. Il suo codice può visualizzare **`iProtetto`**, che è un membro **`protected`** di **`Cbase`**

Tipi di ereditarietà in C++

… e la classe **`CDerivata`** :

```
class CDerivata3: private CBase {
      public:
            void coutProt();
            void coutPub();
};
```

Tipi di ereditarietà in C++

Lo schema **private**:

In CBase, il membro è:	In CDerivata «diventa»:	Regole d'accesso…
public	**private**	Accessibile direttamente a tutte le funzioni membro non **static** e le funzioni **friend**
protected	**private**	Accessibile direttamente a tutte le funzioni membro non **static** e le funzioni **friend**
private	nascosti	Accessibili a funzioni membro e funzioni **friend** tramite funzioni membro **public** e **protected** della classe base

79

Tipi di ereditarietà in C++

Quindi…

```
int main() {
    CDerivata3 x;
    cout << x.iPubblico << endl;
    x.coutPriv();
    x.coutProt();
    …
```

NO. Membro **public** di **CBase**, diventa **private** di **CDerivata3.**

NO. Membro **protected** di **CBase**. Diventa **private** in **CDerivata3**

Si. Membro **public** di **CDerivata3**. Il suo codice può visualizzare **iProtetto**, che è un membro **protected** di **CBase** (ereditato come **private** da **CDerivata3**)

80