Tutorato 5

29/11/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

virtual vs override

- **`virtual`**
  - Permette la ridefinizione di un metodo da parte di una classe in modo polimorfo, tendenzialmente da una classe base a una serie di classi derivate)

- **`override`**
  - Permette di precisare che una classe derivata sta esplicitamente facendo l'overriding di un metodo (= virtuale/stessa firma e parametri)

```
struct Base { virtual void foo() {} };
struct Derived: Base { void foo() override {} };
```

- Use `virtual` for the base class function declaration.
  This is technically necessary.
- Use `override` (only) for a derived class' override.
  This helps maintenance.

RTTI – Runtime Type Information

La Run-time type information (RTTI) è un meccanismo che consente di determinare il tipo di un oggetto durante l'esecuzione del programma.

Esistono tre elementi principali del linguaggio C++ per ottenere informazioni sul tipo in tempo reale:

1. L'operatore **`dynamic_cast`**.
   - Utilizzato per la conversione di tipi polimorfi.

2. L'operatore **`typeid`**.
   - Utilizzato per identificare il tipo esatto di un oggetto.

typeid

Permette di determinare il tipo di un'espressione qualsiasi in un tempo di esecuzione costante. Ritornano un oggetto della classe **`type_info`**, che va incluso nell'header. Normalmente si usa «per controllare l'uguaglianza tra tipi dinamici»

```cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class Persona {
public:
    virtual ~Persona();
};

class Impiegato : public Persona {};

int main() {
    Persona person;
    Impiegato employee;
    Persona* ptr = &employee;
    Persona& ref = employee;
```

```cpp
    cout << typeid(person).name() << endl;
    // Persona (noto a compile-time).
    cout << typeid(employee).name() << endl;
    // Impiegato (noto a compile-time).
    cout << typeid(ptr).name()
    // Persona (noto a compile-time).
    cout << typeid(*ptr).name() << endl;
    // Impiegato (riferimento polimorfo, si trova a compile time
    cout << typeid(ref).name() << std::endl;
    // Impiegato (i riferimenti possono essere polimorfi).
}
```

typeid

- Se l'espressione operando di **`typeid`** è un riferimento ad una classe *che contiene almeno un metodo virtuale*, cioè una classe polimorfa allora restituisce un oggetto di tipo **`type_info`** che rappresenta il tipo dinamico di **`ref`**.

- Se l'espressione operando di **`typeid`** è un puntatore dereferenziato ***`punt`**, dove punt è un puntatore ad un tipo polimorfo, allora **`typeid`** restituisce un oggetto di tipo **`type_info`** che rappresenta il tipo **`T`**, dove **`T*`** è il tipo dinamico di **`punt`**

Attenzione:

- Se la classe non contiene metodi virtuali, **`typeid`** ritorna il tipo statico del riferimento o del puntatore dereferenziato.

- **`typeid`** su un puntatore (non dereferenziato) restituisce sempre il tipo statico del puntatore.

typeid


```cpp
class A {public: virtual ~A() {}};
class B : public A {};
class D: public B {};

#include <iostream>
#include <typeinfo>

int main(){
    B b; D d;
    B& rb = d;
    A* pa = &b;
    if(typeid(rb) == typeid(B)) std::cout << "1\n";
    if(typeid(rb) == typeid(D)) std::cout << "2\n";
    if(typeid(*pa) == typeid(A)) std::cout << "3\n";
    if(typeid(*pa) == typeid(B)) std::cout << "4\n";
    if(typeid(*pa) == typeid(D)) std::cout << "5\n";
    // stampa: 2 4
}
```

typeid


```
1  class A {public: virtual ~A() {}};
2  class B : public A {
3  public:
4      virtual void f() { cout << "B::f" << endl; }
5  };
6  class D: public B {
7  public:
8      void f() { cout << "D::f" << endl; }
9      virtual void g() { cout << "D::g" << endl; }
10 };
11 class E: public D {
12 public:
13     void g() { cout << "E::g" << endl; }
14     void h() { cout << "E::h" << endl; }
15 };
```

```
1  int main()
2  {
3      B b; D d; E e;
4      B* p = &d;
5      p->f(); // ok
6      p->g(); // non compila
7      D* r = &e;
8      r->g(); // ok
9      r->h(); // non compila
10 }
```

dynamic_cast

Utilizzato per effettuare il downcasting (o per convertire allo stesso livello della gerarchia) di un riferimento o di un puntatore a un tipo più specifico nella gerarchia delle classi.

- La corretta conversione viene controllata in fase di esecuzione (runtime).
  - Se i tipi non sono compatibili, verrà lanciata un'eccezione (quando si tratta di riferimenti) o verrà restituito un puntatore nullo (quando si tratta di puntatori).

- Sotto la tipica sintassi (normalmente, è sempre possibile andare dalla base alla derivata)

```
Base *b = dynamic_cast<base*>(derivata)
```

- In generale, si converte la classe nelle parentesi tonde in quella nelle angolate

Attenzione: si può usare anche per fare upcasting (ma tendenzialmente, si usa come scritto)

dynamic_cast

- B tipo polimorfo, D <= B
- B^* p puntatore polimorfo
- Downcast: $B^* \Rightarrow D^*$, $B\ \& \Rightarrow D\&$

`dynamic_cast<D*>(p) != 0`

se e solo se

`TD(p) <= D*`

(tipo dinamico di **`P`** compatibile con il tipo target **`D*`**)

dynamic_cast

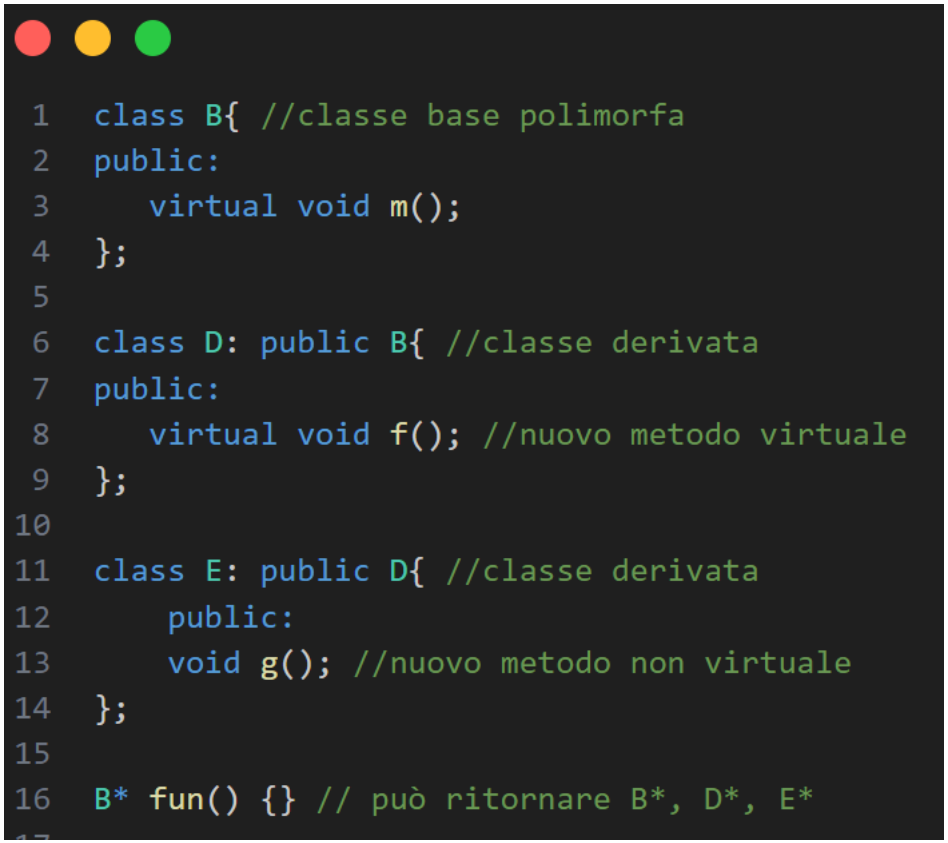
- Conversioni safe (safe cast)
  - *Downcasting* (dall'alto verso il basso = dalla base alla derivata)
  - *Upcasting* (dal basso verso l'alto = dalla derivata alla base)
- Tendenzialmente viene eseguito per safe downcasting e per convertire classi «allo stesso livello» nella gerarchia
- Viene eseguito a runtime, dato che il successo della conversione dipende dal tipo dinamico da convertire

dynamic_cast

- Nel caso dei riferimenti, se **`dynamic_cast`** fallisce, viene sollevata un'eccezione di tipo **`std::bad_cast`**, definito nel file header **`typeinfo`**.

```
1  class X {public: virtual ~X() {}};
2  class B {public: virtual ~B() {}};
3  class D: public B{};
4
5  #include <typeinfo>
6  #include <iostream>
7  using namespace std;
8
9  int main(){
10     D d; B& b = d; //upcast
11     try{X& x = dynamic_cast<X&>(b);}
12     catch(bad_cast& e){cout << "Fallimento" << endl;}
13 }
```

dynamic_cast



```
1  class B{ //classe base polimorfa
2  public:
3     virtual void m();
4  };
5
6  class D: public B{ //classe derivata
7  public:
8     virtual void f(); //nuovo metodo virtuale
9  };
10
11 class E: public D{ //classe derivata
12     public:
13     void g(); //nuovo metodo non virtuale
14 };
15
16 B* fun() {} // può ritornare B*, D*, E*
```

```
int main(){
    B* p = fun();
    if(dynamic_cast<D*>(p)) //downcast possibile
        static_cast<D*>(p)->f(); //ok, come downcast
    E* q = dynamic_cast<E*>(p); //downcast possibile
    if(q) //ok, come downcast
        q->g();
}
```

Downcasting e check dinamico

- In generale il safe downcasting va usato solamente in caso di necessità.
  - Usarne tanti è lento in memoria e fa dipende l'implementazione di un oggetto dalla serie di classi derivate (accoppiamento nel codice)

- Tipicamente, si ha la necessità di fare un downcasting di un puntatore ad una classe base B per ottenere la disponibilità dei membri di una classe derivata da B che non sono stati ereditati da B.

- Non si deve cadere nella tentazione di rimpiazzare le chiamate polimorfe automatiche ai metodi virtuali con delle istruzioni condizionali (ad esempio degli switch) che usano il dynamic cast per effettuare type checking dinamico inutile.
  - Nella maggior parte dei casi una tale pratica va contro l'estensibilità del codice.

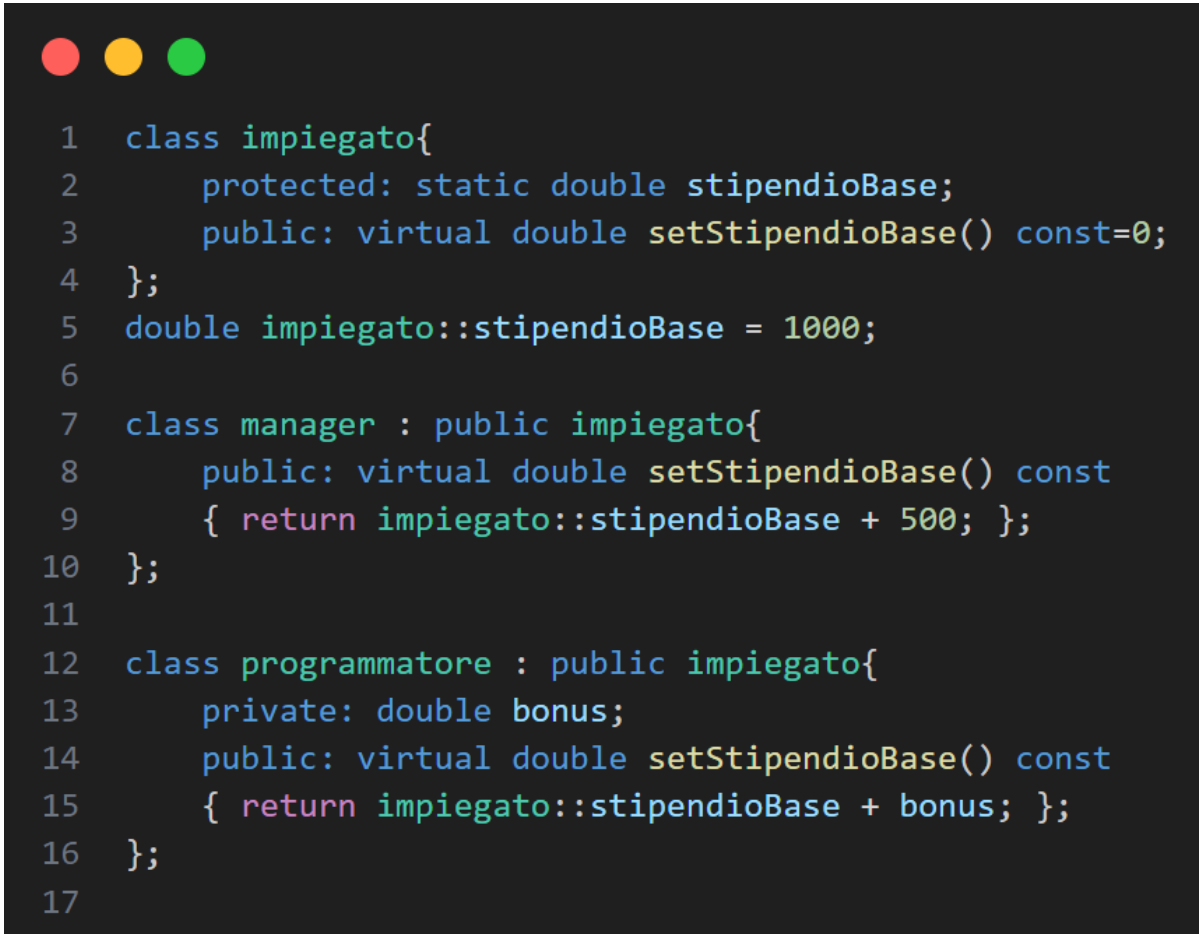
- Quando possibile, usare metodi virtuali nelle classi base piuttosto che fare type checking dinamico.

Downcasting e check dinamico


```cpp
class Base {
public:
    virtual ~Base() {}
};

class D1: public Base{
public:
    virtual void doSomethingD1() {};
};

class D2: public Base{
public:
    virtual void doSomethingD2() {};
};

class D3: public Base{
public:
    virtual void doSomethingD3() {};
};
```

```cpp
int main() {
    Base* b = new D1();
    D1* d1 = dynamic_cast<D1*>(b);
    if (d1 != nullptr) {
        cout << "D1" << endl;
    }
    D2* d2 = dynamic_cast<D2*>(b);
    if (d2 != nullptr) {
        cout << "D2" << endl;
    }
    D3* d3 = dynamic_cast<D3*>(b);
    if (d3 != nullptr) {
        cout << "D3" << endl;
    }
    return 0;
}
```

Downcasting e check dinamico

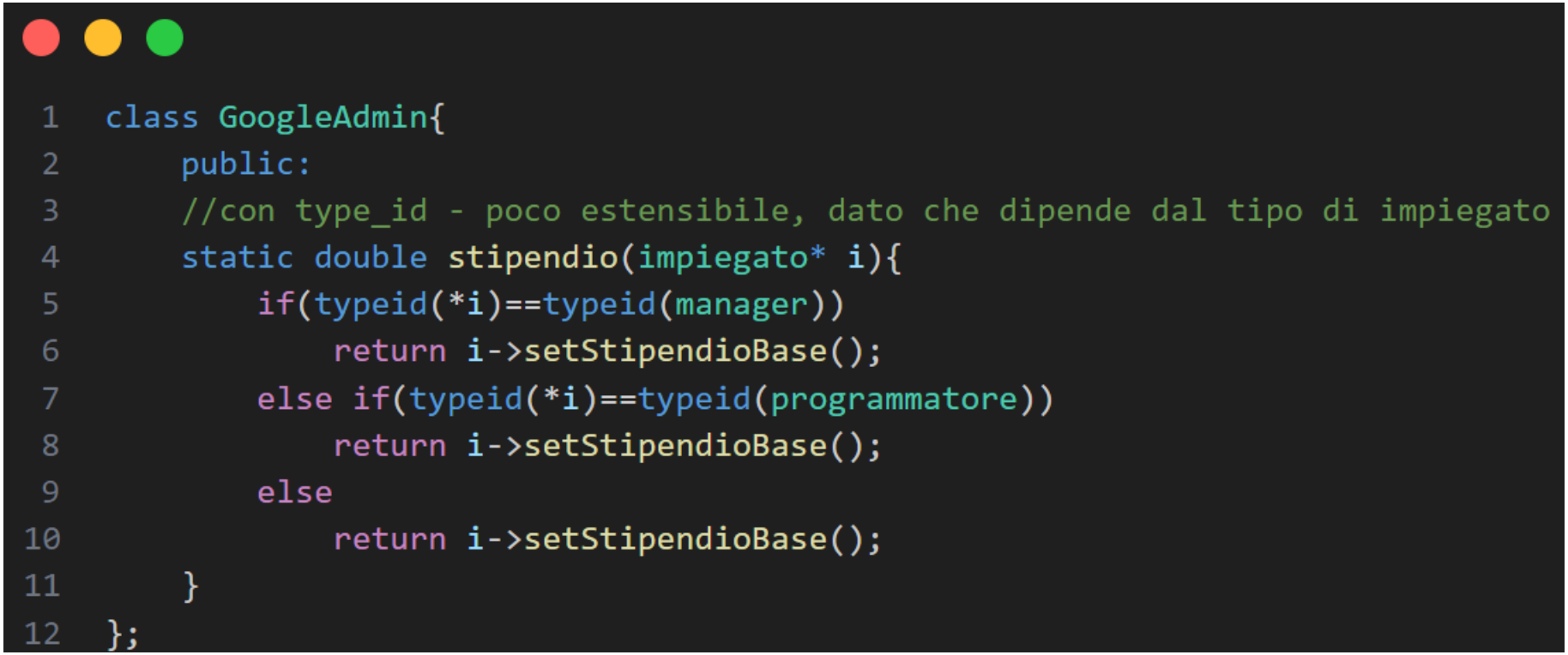


```
1  class impiegato{
2      protected: static double stipendioBase;
3      public: virtual double setStipendioBase() const=0;
4  };
5  double impiegato::stipendioBase = 1000;
6
7  class manager : public impiegato{
8      public: virtual double setStipendioBase() const
9      { return impiegato::stipendioBase + 500; };
10 };
11
12 class programmatore : public impiegato{
13     private: double bonus;
14     public: virtual double setStipendioBase() const
15     { return impiegato::stipendioBase + bonus; };
16 };
17
```

Downcasting e check dinamico

```
1  class GoogleAdmin{
2      public:
3      //con dynamic_cast
4      static double stipendio(impiegato* i){
5          manager* m = dynamic_cast<manager*>(i);
6          if(m) return m->setStipendioBase();
7          programmatore* p = dynamic_cast<programmatore*>(i);
8          if(p) return p->setStipendioBase();
9          return i->setStipendioBase();
10     }
11 };
```

Downcasting e check dinamico



```
1  class GoogleAdmin{
2      public:
3      //con type_id - poco estensibile, dato che dipende dal tipo di impiegato
4      static double stipendio(impiegato* i){
5          if(typeid(*i)==typeid(manager))
6              return i->setStipendioBase();
7          else if(typeid(*i)==typeid(programmatore))
8              return i->setStipendioBase();
9          else
10             return i->setStipendioBase();
11     }
12 };
```

In the real world: Principi SOLID

1. Single Responsibility Principle
- Ogni classe ha una sola responsabilità logicamente definita

2. Open-Closed Principle
- Classi aperte all'estensione, ma chiuse alla modifica

3. Liskov Substitution Principle
- Gli oggetti dovrebbero rimpiazzare dinamicamente i tipi in base al loro contratto di invocazione

4. Interface Segregation Principle
- Meglio creare interfacce ad-hoc per situazione

5. Dependency Inversion Principle
- Usare il polimorfismo in maniera tale da non dipendere da sottotipi, ma rendere classi astratte e usare metodi alla bisogna

Esercizio 1: Funzione

Si assuma che `Abs` sia una classe base astratta. Definire un template di funzione `Fun(T1*, T2&)` che ritorna un booleano con il seguente comportamento. Consideriamo una istanziazione implicita `Fun(p,r)` dove supponiamo che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi (cioè che contengono almeno un metodo virtuale). Allora `Fun(p,r)` ritorna `true` se e soltanto se valgono le seguenti condizioni:

1. i parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. siano `D1*` il tipo dinamico di `p` e `D2&` il tipo dinamico di `r`. Allora (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `Abs`.

Esercizio 1: Soluzione


```
1    template <class T1,class T2>
2    bool FUN(T1* t1, T2& t2) const {
3        return typeid(T1) == typeid(T2) && typeid(*t1) == typeid(t2) && dynamic_cast<Abs*>(t1);
4    }
```

Esercizio 2: Gerarchia

Definire una unica gerarchia di classi che includa:

(1) una classe base polimorfa `A` alla radice della gerarchia;

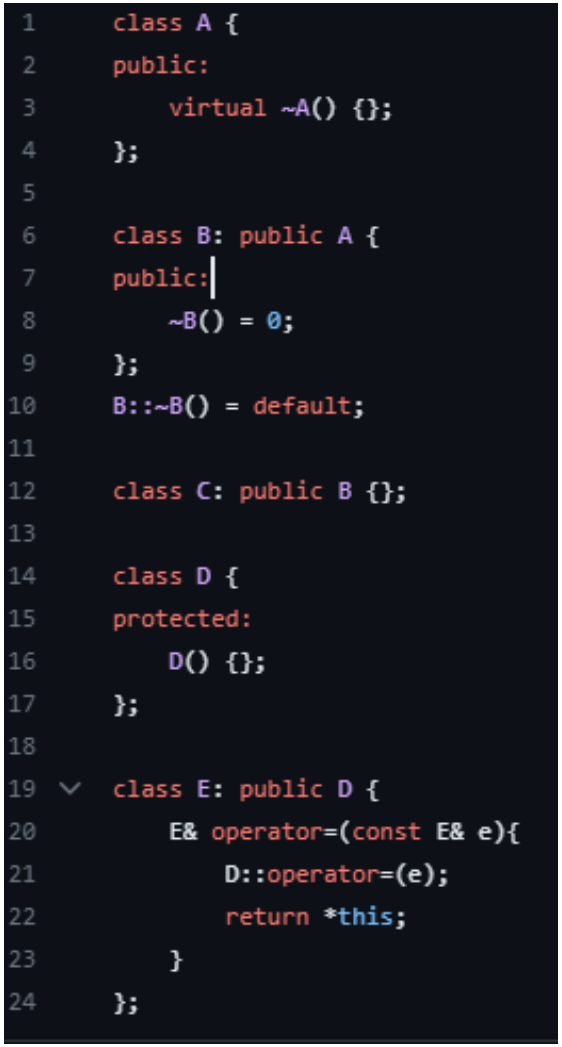
(2) una classe derivata astratta `B`;

(3) una sottoclasse `C` di `B` che sia concreta;

(4) una classe `D` che non permetta la costruzione pubblica dei suoi oggetti, ma solamente la costruzione di oggetti di `D` che siano sottooggetti;

(5) una classe `E` derivata direttamente da `D` e con l'assegnazione ridefinita pubblicamente con comportamento identico a quello dell'assegnazione standard di `E`.

Esercizio 2: Soluzione



```cpp
class A {
public:
    virtual ~A() {};
};

class B: public A {
public:
    ~B() = 0;
};
B::~B() = default;

class C: public B {};

class D {
protected:
    D() {};
};

class E: public D {
    E& operator=(const E& e){
        D::operator=(e);
        return *this;
    }
};
```

Esercizio 3: Cosa Stampa

Esercizio Cosa Stampa

```
class B {
protected:
  virtual void h() {cout<<"B::h ";}
public:
  virtual void f() {cout <<"B::f "; g(); h();}
  virtual void g() const {cout <<"B::g ";}
  virtual void k() {cout <<"B::k "; h(); m(); }
  void m() {cout <<"B::m "; g(); h();}
  virtual B* n() {cout <<"B::n "; return this;}
};

class D: public B {
protected:
  void h() {cout <<"D::h ";}
public:
  virtual void g() {cout <<"D::g ";}
  void k() const {cout <<"D::k "; k();}
  void m() {cout <<"D::m "; g(); h();}
};

class C: public B {
public:
  virtual void g() const {cout <<"C::g ";}
  void k() {cout <<"C::k "; B::n();}
  virtual void m() {cout <<"C::m "; g(); h();}
  B* n() {cout <<"C::n "; return this;}
};

class E: public C {
protected:
  void h() {cout <<"E::h ";}
public:
  void m() {cout <<"E::m "; g(); h();}
  C* n() {cout <<"E::n "; return this;}
};

const B* p1 = new D(); B* p2 = new C(); B* p3 = new D(); C* p4 = new E(); B* p5 = new E();
```

Le precedenti definizioni compilano correttamente. Per ognuno dei seguenti 14 statement in tabella con **numerazione da 01 a 14**, scrivere **chiaramente nel foglio 14 risposte con numerazione da 01 a 14** e per ciascuna risposta:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **UNDEFINED BEHAVIOUR** se l'istruzione compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva **chiaramente** la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

Esercizio 3: Cosa Stampa


```
class B {
protected:
  virtual void h() {cout<<"B::h ";}
public:
  virtual void f() {cout <<"B::f "; g(); h();}
  virtual void g() const {cout <<"B::g ";}
  virtual void k() {cout <<"B::k "; h(); m(); }
  void m() {cout <<"B::m "; g(); h();}
  virtual B* n() {cout <<"B::n "; return this;}
};
```

```
class C: public B {
public:
  virtual void g() const {cout <<"C::g ";}
  void k() {cout <<"C::k "; B::n();}
  virtual void m() {cout <<"C::m "; g(); h();}
  B* n() {cout <<"C::n "; return this;}
};
```

```
class D: public B {
protected:
  void h() {cout <<"D::h ";}
public:
  virtual void g() {cout <<"D::g ";}
  void k() const {cout <<"D::k "; k();}
  void m() {cout <<"D::m "; g(); h();}
};
```

```
class E: public C {
protected:
  void h() {cout <<"E::h ";}
public:
  void m() {cout <<"E::m "; g(); h();}
  C* n() {cout <<"E::n "; return this;}
};
```

```
const B* p1 = new D(); B* p2 = new C(); B* p3 = new D(); C* p4 = new E(); B* p5 = new E();
```

```
01: p1->g();
02: (p1->n())->g();
03: p2->f();
04: p2->m();
05: (static_cast<D*>(p2))->k();
06: p3->k();
07: p3->f();
```

```
08: (p3->n())->m();
09: (p3->n())->n()->g();
10: (static_cast<C*>(p3->n()))->g();
11: (p4->n())->m();
12: p5->g();
13: p5->k();
14: (dynamic_cast<C*>(p5))->m();
```

Esercizio 3: Soluzione


```
// SOLUZIONE
01: B::g
02: NON COMPILA
03: B::f  C::g  B::h
04: B::m  C::g  B::h
05: UNDEFINED BEHAVIOUR
06: B::k  D::h  B::m  B::g  D::h
07: B::f  B::g  D::h
08: B::n  B::m  B::g  D::h
09: B::n  B::n  B::g
10: B::n  B::g
11: E::n  B::m  C::g  E::h
12: C::g
13: C::k  B::n
14: E::m  C::g  E::h
```

Esercizio 4: Sottotipi

Siano `A`, `B`, `C` e `D` distinte classi polimorfe. Si considerino le seguenti definizioni.

```
template<class X>
X& fun(X& ref) { return ref; };

main() {
  B b;
  fun<A>(b);
  B* p = new D();
  C c;
  try{
    dynamic_cast<B&>(fun<A>(c));
    cout << "topolino";
  }
  catch(bad_cast) { cout << "pippo "; }
  if( !(dynamic_cast<D*>(new B())) ) cout << "pluto ";
}
```

Si supponga che:

1. il `main()` compili correttamente ed esegua senza provocare errori a run-time;
2. l'esecuzione del `main()` provochi in output su `cout` la stampa `pippo pluto`.

In tali ipotesi, per ognuna delle relazioni di sottotipo $X \leq Y$ nelle seguenti tabelle segnare con una croce l'entrata

(a) "Vero" per indicare che `X` **sicuramente è** sottotipo di `Y`;
(b) "Falso" per indicare che `X` **sicuramente non è** sottotipo di `Y`;
(c) "Possibile" **altrimenti**, ovvero se non valgono nè (a) nè (b).

	Vero	Falso	Possibile
$A \leq B$			
$A \leq C$			
$A \leq D$			
$B \leq A$			
$B \leq C$			
$B \leq D$			

	Vero	Falso	Possibile
$C \leq A$			
$C \leq B$			
$C \leq D$			
$D \leq A$			
$D \leq B$			
$D \leq C$			

Esercizio 4: Soluzione

Siano A, B, C e D distinte classi polimorfe. Si considerino le seguenti definizioni.

```
template<class X>
X& fun(X& ref) { return ref; };

main() {
  B b;
  fun<A>(b);
  B* p = new D();
  C c;
  try{
    dynamic_cast<B&>(fun<A>(c));
    cout << "topolino";
  }
  catch(bad_cast) { cout << "pippo "; }
  if( !(dynamic_cast<D*>(new B())) ) cout << "pluto ";
}
```

Soluzione

vincoli:
$B \leq A$, $C \leq A$, $D \leq B$, $C \not\leq B$

	Vero	Falso	Possibile
A≤B		X	
A≤C		X	
A≤D		X	
B≤A	X		
B≤C			X
B≤D		X	

	Vero	Falso	Possibile
C≤A	X		
C≤B		X	
C≤D		X	
D≤A	X		
D≤B	X		
D≤C			X

Esercizio 5: Stampe con lettere

Si considerino le seguenti definizioni di classe e funzione:

```
class A {
public:
  virtual ~A() {};
};
class B: public A {};
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};

char F(A* p, C& r) {
  B* punt1 = dynamic_cast<B*> (p);
  try{
    E& s = dynamic_cast<E&> (r);
  }
  catch(bad_cast) {
    if(punt1) return 'O';
    else return 'M';
  }
  if(punt1) return 'R';
  return 'A';
}
```

Si consideri inoltre il seguente `main()` incompleto, dove `?` è semplicemente un simbolo per una incognita:

```
main(){
  A a; B b; C c; D d; E e;
  cout << F(?,?) << F(?,?) << F(?,?) << F(?,?);
}
```

Definire opportunamente le chiamate in tale `main()` usando gli oggetti `a`, `b`, `c`, `d`, `e` locali al `main()` in modo tale che la sua esecuzione provochi la stampa `ROMA`.

Esercizio 5: Soluzione (possibile)


```
int main() {
  A a; B b; C c; D d; E e;
  cout << F(&b,e) << F(&b,c) << F(&a,c) << F(&a,e);
}
```