Tutorato 7

20/12/2023

Programmazione ad Oggetti – 2023-2024

Gabriel Rovesti

2103389 – LM Computer Science

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

Esercizio 1: Modellazione

Si consideri il seguente modello di realtà concernente l'app InForma per archiviare allenamenti sportivi.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `Workout` i cui oggetti rappresentano un allenamento (workout) archiviabile in InForma. Ogni `Workout` è caratterizzato dalla durata temporale espressa in minuti. La classe è astratta in quanto prevede i seguenti **metodi virtuali puri**:
   - un metodo di "clonazione": `Workout* clone()`.
   - un metodo `int calorie()` con il seguente contratto puro: `w->calorie()` ritorna il numero di calorie consumate durante l'allenamento `*w`.
2. Definire una classe concreta `Corsa` derivata da `Workout` i cui oggetti rappresentano un allenamento di corsa. Ogni oggetto `Corsa` è caratterizzato dalla distanza percorsa espressa in Km. La classe `Corsa` implementa i metodi virtuali puri di `Workout` come segue:
   - implementazione della clonazione standard per la classe `Corsa`.
   - per ogni puntatore `p` a `Corsa`, `p->calorie()` ritorna il numero di calorie dato dalla formula $500K^2/D$, dove K è la distanza percorsa in Km nell'allenamento `*p` e D è la durata in minuti dell'allenamento `*p`.
3. Definire una classe astratta `Nuoto` derivata da `Workout` i cui oggetti rappresentano un generico allenamento di nuoto che non specifica lo stile di nuoto. Ogni oggetto `Nuoto` è caratterizzato dal numero di vasche nuotate.
4. Definire una classe concreta `StileLibero` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile libero. La classe `StileLibero` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `StileLibero`.
   - per ogni puntatore `p` a `StileLibero`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 10$ le calorie sono $35V$, mentre se $D \geq 10$ le calorie sono $40V$.

Esercizio 1: Modellazione

5. Definire una classe concreta `Dorso` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile dorso. La classe `Dorso` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Dorso`.
   - per ogni puntatore `p` a `Dorso`, `p->calorie()` ritorna il seguente numero di calorie: se D è la durata in minuti dell'allenamento `*p` e V è il numero di vasche nuotate nell'allenamento `*p` allora quando $D < 15$ le calorie sono $30V$, mentre se $D \geq 15$ le calorie sono $35V$.
6. Definire una classe concreta `Rana` derivata da `Nuoto` i cui oggetti rappresentano un allenamento di nuoto a stile rana. La classe `Rana` implementa i metodi virtuali puri di `Nuoto` come segue:
   - implementazione della clonazione standard per la classe `Rana`.
   - per ogni puntatore `p` a `Rana`, `p->calorie()` ritorna $25V$ calorie dove V è il numero di vasche nuotate nell'allenamento `*p`.

(B) Definire una classe `InForma` i cui oggetti rappresentano una installazione dell'app. Un oggetto di `InForma` è quindi caratterizzato da un contenitore di elementi di tipo `const Workout*` che contiene tutti gli allenamenti archiviati dall'app. La classe `InForma` rende disponibili i seguenti metodi:

1. Un metodo `vector<Nuoto*> vasche(int)` con il seguente comportamento: una invocazione `app.vasche(v)` ritorna un STL vector di puntatori a **copie** di tutti e soli gli allenamenti a nuoto memorizzati in `app` con un numero di vasche percorse $>$ `v`.
2. Un metodo `vector<Workout*> calorie(int)` con il seguente comportamento: una invocazione `app.calorie(x)` ritorna un vector contenente dei puntatori a **copie** di tutti e soli gli allenamenti memorizzati in `app` che : (i) hanno comportato un consumo di calorie $>$ `x`; e (ii) non sono allenamenti di nuoto a rana.
3. Un metodo `void removeNuoto()` con il seguente comportamento: una invocazione `app.removeNuoto()` rimuove dagli allenamenti archiviati in `app` **tutti** gli allenamenti a nuoto che abbiano il massimo numero di calorie tra tutti gli allenamenti a nuoto; se `app` non ha archiviato alcun allenamento a nuoto allora viene sollevata l'eccezione "`NoRemove`" di tipo `std::string`.

Esercizio 2: Ridefinizioni

Esercizio Definizioni

```
class Z {
private:
  int x;
};

class B {
private:
  Z bz;
};

class C: virtual public B {
private:
  Z cz;
};

class D: public C {
};

class E: virtual public B {
public:
  Z ez;
  // ridefinizione assegnazione
  // standard di E
};

class F: public D, public E {
private:
  Z* fz;
public:
  // ridefinizione del costruttore di copia profonda di F
  // ridefinizione del distruttore profondo di F
  // definizione del metodo di clonazione di F
};
```

Si considerino le definizioni sopra.
(1) Ridefinire l'assegnazione della classe `E` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `E`. Naturalmente non è permesso l'uso della keyword `default`.
(2) Ridefinire il costruttore di copia profonda della classe `F`.
(3) Ridefinire il distruttore profondo della classe `F`.
(4) Definire il metodo di clonazione della classe `F`.

Esercizio 2: Soluzione


```
// SOLUZIONE
class E: virtual public B {
public:
  Z ez;
  E& operator(const E& e) {
    B::operator=(e);
    ez=e.ez;
    return *this;
  }
};

class F: public D, public E {
private:
  Z* fz;
public:
  F(const F& f): B(f), D(f), E(f), fz(f.fz!=nullptr ? new Z(*f.fz) : nullptr) {}
  ~F() {delete fz;}
  virtual F* clone() const {return new F(*this);}
};
```

Esercizio 3: Funzione

Si considerino le seguenti dichiarazioni di classi di qualche libreria grafica, dove gli oggetti delle classi `Container`, `Component`, `Button` e `MenuItem` sono chiamati, rispettivamente, contenitori, componenti, pulsanti ed entrate di menu.

```
class Component;

class Container {
public:
  virtual ~Container();
  vector<Component*> getComponents() const;
};

class Component: public Container {};

class Button: public Component {
public:
  vector<Container*> getContainers() const;
};

class MenuItem: public Button {
public:
  void setEnabled(bool b = true);
};

class NoButton {};
```

Esercizio 3: Funzione

Assumiamo i seguenti fatti.

1. Il comportamento del metodo `getComponents()` della classe `Container` è il seguente: `c.getComponents()` ritorna un vector di puntatori a tutte le componenti inserite nel contenitore `c`; se `c` non ha alcuna componente allora ritorna un vector vuoto.
2. Il comportamento del metodo `getContainers()` della classe `Button` è il seguente: `b.getContainers()` ritorna un vector di puntatori a tutti i contenitori che contengono il pulsante `b`; se `b` non appartiene ad alcun contenitore allora ritorna un vector vuoto.
3. Il comportamento del metodo `setEnabled()` della classe `MenuItem` è il seguente: `mi.setEnabled(b)` abilita (con `b==true`) o disabilita (con `b==false`) l'entrata di menu `mi`.

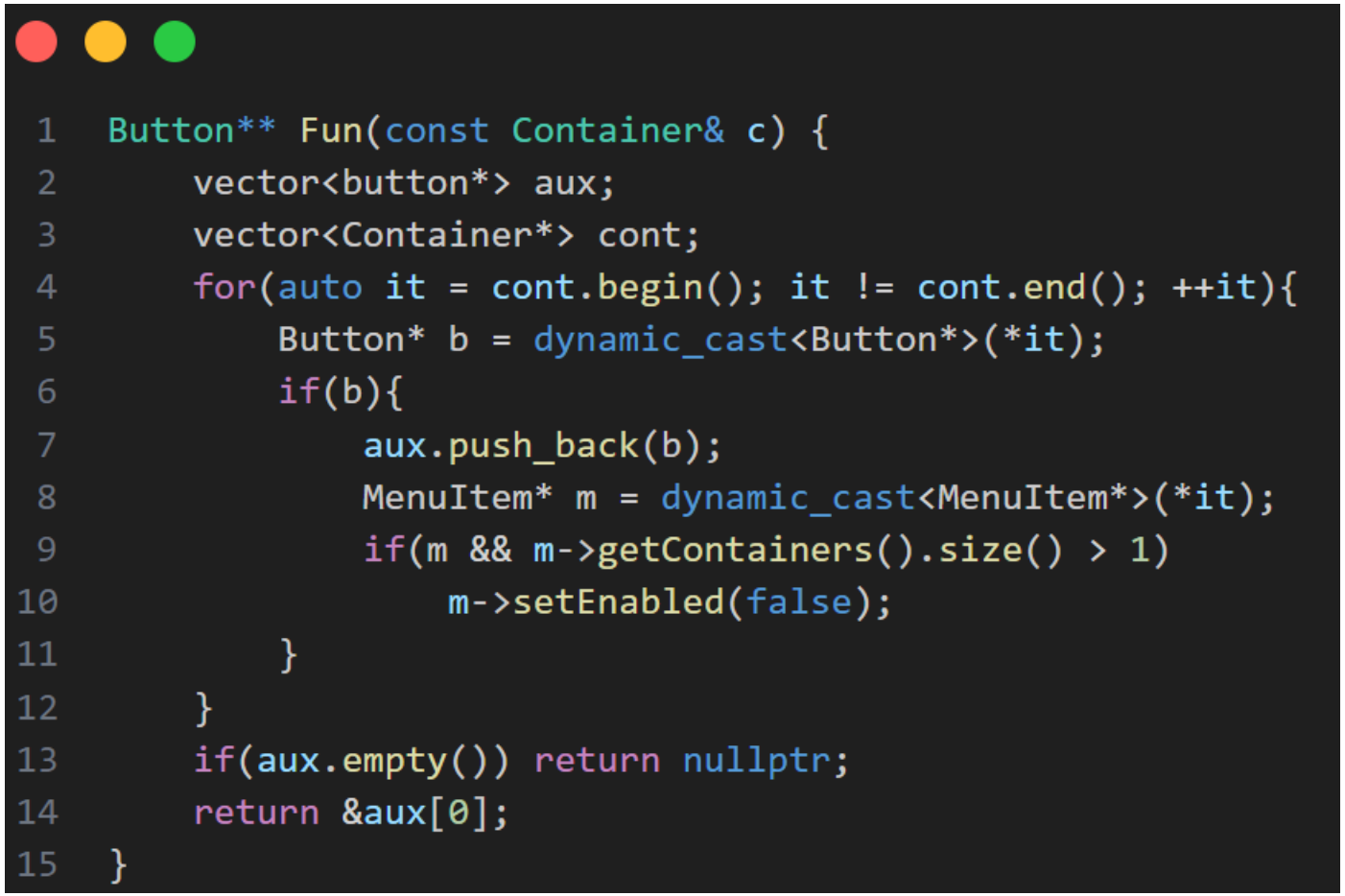
Definire una funzione `Button** Fun(const Container&)` con il seguente comportamento: in ogni invocazione `Fun(c)`

1. Se `c` contiene almeno una componente `Button` allora

   ritorna un puntatore alla prima cella di un array dinamico di puntatori a pulsanti contenente tutti e soli i puntatori ai pulsanti che sono componenti del contenitore `c` ed in cui tutte le componenti che sono una entrata di menu e sono contenute in almeno 2 contenitori vengono disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora solleva una eccezione di tipo `NoButton`.

Esercizio 3: Soluzione



```
1  Button** Fun(const Container& c) {
2      vector<button*> aux;
3      vector<Container*> cont;
4      for(auto it = cont.begin(); it != cont.end(); ++it){
5          Button* b = dynamic_cast<Button*>(*it);
6          if(b){
7              aux.push_back(b);
8              MenuItem* m = dynamic_cast<MenuItem*>(*it);
9              if(m && m->getContainers().size() > 1)
10                 m->setEnabled(false);
11         }
12     }
13     if(aux.empty()) return nullptr;
14     return &aux[0];
15 }
```

Esercizio 4: Cosa Stampa


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

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

class D: virtual public B {
public:
  D() {cout<< " D() ";}
  ~D() {cout<< " ~D() ";}
  virtual void g() {cout <<" D::g ";}
  const B* j() {cout <<" D::j "; return this;}
  void k() const {cout <<" D::k "; k();}
  void m() {cout <<" D::m "; g(); j();}
};

class E: public C, public D  {
public:
  E() {cout<< " E() ";}
  ~E() {cout<< " ~E() ";}
  virtual void g() const {cout <<" E::g ";}
  const E* j() {cout <<" E::j "; return this;}
  void m() {cout <<" E::m "; g(); j();}
  D& n() final {cout <<" E::n "; return *this;}
};

class F: virtual public E {
public:
  F() {cout<< " F() ";}
  ~F() {cout<< " ~F() ";}
  F(const F& x): B(x) {cout<< " Fc ";}
  void k() {cout <<" F::k "; g();}
  void m() {cout <<" F::m "; j();}
};

B* p1 = new E(); B* p2 = new C(); B* p3 = new D();
C* p4 = new E(); const B* p5 = new E(); const B* p6 = new F();
```

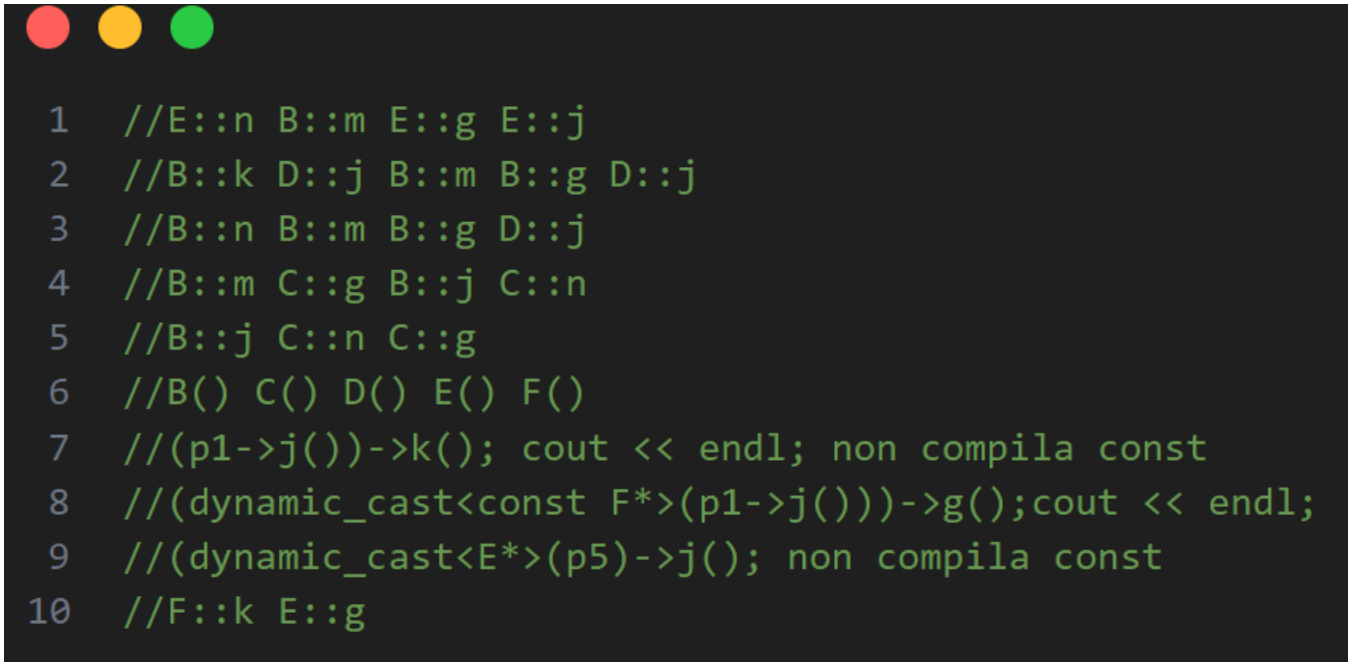
Queste definizioni compilano correttamente (con opportuni `#include` e `using`). Per ognuno dei seguenti statement scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dello statement provoca un errore;
- **UNDEFINED** se lo statement compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore run-time;
- se lo statement compila ed esegue correttamente (senza undefined behaviour o errori run-time) allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

Esercizio 4: Cosa Stampa

```
(p4->n()).m(); ..............................

p3->k(); ....................................

(p3->n()).m(); ..............................

p2->m(); ....................................

(p2->j())->g(); .............................

C* p = new F(F()); ..........................

(p1->j())->k(); .............................

(dynamic_cast<const F*>(p1->j()))->g(); .....

(dynamic_cast<E*>(p5))->j(); ................

(dynamic_cast<C*>(const_cast<B*>(p6)))->k(); ..
```

Esercizio 4: Soluzione



```
1  //E::n B::m E::g E::j
2  //B::k D::j B::m B::g D::j
3  //B::n B::m B::g D::j
4  //B::m C::g B::j C::n
5  //B::j C::n C::g
6  //B() C() D() E() F()
7  //(p1->j())->k(); cout << endl; non compila const
8  //(dynamic_cast<const F*>(p1->j()))->g();cout << endl;
9  //(dynamic_cast<E*>(p5)->j(); non compila const
10 //F::k E::g
```

Esercizio 5: Sottotipi

Esercizio Tipi

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

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

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};
```

```
int main() {
B b; C c; D d; E e;


cout << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...)


     << F(...,...) << F(...,...) << F(...,...) << F(...,...) << F(...,...);
}
```

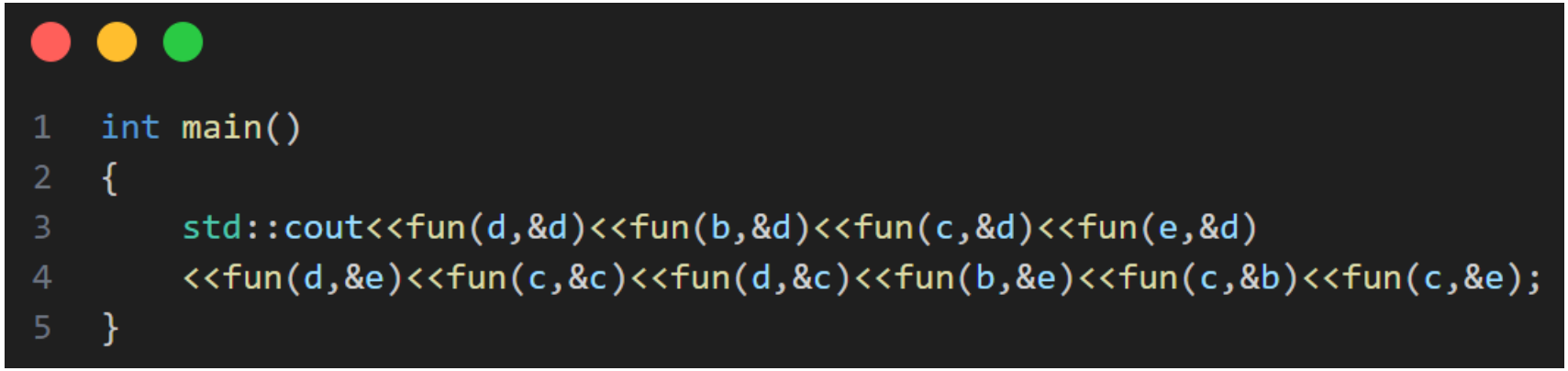
Si considerino le precedenti definizioni ed il `main()` incompleto. Definire opportunamente negli appositi spazi `...,...` le chiamate alla funzione `F` di questo `main()` usando gli oggetti locali `b`, `c`, `d`, `e`, `f` in modo tale che: (1) non vi siano errori in compilazione o a run-time; (2) le chiamate di `F` siano **tutte diverse** tra loro; (3) l'esecuzione produca in output **esattamente** la stampa **`6544233241`**.

Esercizio 5: Sottotipi


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

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

class C: virtual public B {};

class D: virtual public B {};

class E: public C, public D {};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
int main() {
B b; C c; D d; E e;


cout << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....)


     << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....);
}
```

Esercizio 5: Soluzione (possibile)



```
1  int main()
2  {
3      std::cout<<fun(d,&d)<<fun(b,&d)<<fun(c,&d)<<fun(e,&d)
4      <<fun(d,&e)<<fun(c,&c)<<fun(d,&c)<<fun(b,&e)<<fun(c,&b)<<fun(c,&e);
5  }
```

Esercizio 6: Errore runtime

Scrivere un programma consistente di esattamente tre classi `A`, `B` e `C` e della sola funzione `main()` che soddisfi le seguenti condizioni:

1. la classe `A` è definita come:

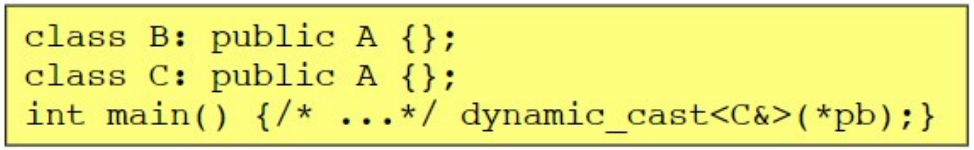
   ```
   class A { public: virtual ~A(){} };
   ```

2. le classi `B` e `C` devono essere definite per ereditarietà e non contengono alcun membro
3. la funzione `main()` definisce le tre variabili:

   ```
   A* pa = new A; B* pb = new B; C* pc = new C;
   ```

   e nessuna altra variabile (di alcun tipo)
4. la funzione `main()` può **utilizzare solamente** espressioni di tipo `A*`, `B*` e `C*`, **non può** sollevare eccezioni mediante una `throw` e **non può** invocare l'operatore `new`
5. il programma deve compilare correttamente
6. l'esecuzione di `main()` **deve provocare un errore run-time**.

Esercizio 6: Soluzione



```
class B: public A {};
class C: public A {};
int main() {/* ...*/ dynamic_cast<C&>(*pb);}
```

Soluzione

Dereferencing a NULL pointer is undefined behavior.

In fact the standard calls this exact situation out in a note (8.3.2/4 "References"):

> Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior.