

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro `p2` tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce una vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro `p2` con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro `p2` mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

[ESERCIZI - RIVALUTAZIONE] → OCCASIONE

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

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

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

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

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

p1->g();
 p1->k();
 p2->f();
 p2->m();
 p3->k();
 p3->f();
 p4->m();
 p4->k();
 p5->g();
 (p3->n())->m();
 (p3->n())->n()->g();
 (p4->n())->m();
 (p5->n())->g();
 (dynamic_cast<B*>(p1))->m();
 (static_cast<C*>(p2))->k();
 (static_cast<B*>(p3->n()))->g();

STATIC_CAST = CAST FATTO PRIMA DI ESEGUIRE
 ERRORI SU DYNAMIC_CAST PERCHÉ È FATTO A RUN-TIME
 (A) → STAMPANDO P3 → N C
 → B C RETURN THIS
 CONST A* CONST A* STA B STA C

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro p2 tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce un vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro p2 con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro p2 mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

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

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

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

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

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

→ `(static_cast<C*>(p2))->k();`

$A * P2 = \text{new } B$

$C * P2 = \text{new } C$

ERRORS RUNTIME!

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

STACK

OVERFLOW

```
p1->g(); .....
p1->k(); .....
p2->f(); .....
p2->m(); .....
p3->k(); .....
p3->f(); .....
p4->m(); .....
p4->k(); .....
p5->g(); .....
(p3->n())->m(); .....
(p3->n())->n()->g(); .....
(p4->n())->m(); .....
(p5->n())->g(); .....
(dynamic_cast<B*>(p1))->m(); .....
(static_cast<C*>(p2))->k(); .....
(static_cast<B*>(p3->n()))->g(); .....
```

A
B C B = B / CONST A *

STATIC-CAST → CAMBIO PRIMA DI ESSERE GUARDATO

DYNAMIC-CAST → CAMBIO DURANTE L'ESecuzione

$A * P2 = \text{new } C();$

PUNTERO = CONVERSIONE SUBITO = STATIC CASTING
 $A * C$

* THIS → PRIMA * THIS, poi CASTING

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro p2 tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce una vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro p2 con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro p2 mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

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

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

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

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

C++ 17 IN RO
(ANNOTAZIONE)

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

NO RUN TIME OVERRIDING

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

```
p1->g(); .....
p1->k(); .....
p2->f(); .....
p2->m(); .....
p3->k(); .....
p3->f(); .....
p4->m(); .....
p4->k(); .....
p5->g(); .....
(p3->n())->m(); .....
(p3->n())->n()->g(); .....
(p4->n())->m(); .....
(p5->n())->g(); .....
(dynamic_cast<B*>(p1))->m(); .....
(static_cast<C*>(p2))->k(); .....
(static_cast<B*>(p3->n()))->g(); .....
```

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro `p2` tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce una vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro `p2` con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro `p2` mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

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

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

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

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

$A \cdot p1 = \text{new } D \rightarrow B \cdot p1 = \text{new } D$
 $\cdot (\text{dynamic_cast}<B*>(p1)) \rightarrow m();$

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

```
p1->g(); .....
p1->k(); .....
p2->f(); .....
p2->m(); .....
p3->k(); .....
p3->f(); .....
p4->m(); .....
p4->k(); .....
p5->g(); .....
(p3->n())->m(); .....
(p3->n())->n()->g(); .....
(p4->n())->m(); .....
(p5->n())->g(); .....
(dynamic_cast<B*>(p1))->m(); .....
(static_cast<C*>(p2))->k(); .....
(static_cast<B*>(p3->n()))->g(); .....
```

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro p2 tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce una vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro p2 con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro p2 mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

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

```
class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};
```

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

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

```
class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};
```

(p5->n())->g();

NON COMPILA
↓

CONST-CAST (A*)
(p5->n())
→ GC
COMPILA!

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

```
p1->g(); .....
p1->k(); .....
p2->f(); .....
p2->m(); .....
p3->k(); .....
p3->f(); .....
p4->m(); .....
p4->k(); .....
p5->g(); .....
(p3->n())->m(); .....
(p3->n())->n()->g(); .....
(p4->n())->m(); .....
(p5->n())->g(); .....
(dynamic_cast<B*>(p1))->m(); .....
(static_cast<C*>(p2))->k(); .....
(static_cast<B*>(p3->n()))->g(); .....
```

- Un metodo `vector<Consumo> rimuoviConsumoZero()` con il seguente comportamento: una invocazione `p2.rimuoviConsumoZero()` rimuove dalle schede SIM gestite dal centro `p2` tutte le schede con piano di tariffazione a consumo che hanno un credito residuo pari a 0 €, e restituisce una vettore contenente una copia di tutte le schede con piano di tariffazione a consumo rimosse.
- Un metodo `double contabilizza()` con il seguente comportamento: una invocazione `p2.contabilizza()` provoca la contabilizzazione in tutte le schede SIM gestite dal centro `p2` con credito residuo positivo di una telefonata di 1 secondo, di una connessione di 1 MB e dell'invio di 1 sms, e restituisce il guadagno ottenuto dal centro `p2` mediante questa contabilizzazione (cioè la differenza del totale dei crediti residui di tutte le schede prima e dopo questa contabilizzazione).

Esercizio 2

```

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

class C: public A {
private:
    void j() { cout <<" C::j "; }
public:
    virtual void g() { cout <<" C::g "; }
    void m() { cout <<" C::m "; g(); j(); }
    void k() const { cout <<" C::k "; k(); }
};

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

class D: public B {
protected:
    void j() { cout <<" D::j "; }
public:
    B* n() final { cout <<" D::n "; return this; }
    void m() { cout <<" D::m "; g(); j(); }
};

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

```

Handwritten notes:

- ② *no virtual* (pointing to `void m()` in class A)
- ③ (pointing to `virtual void g()` in class B)
- THIS* (pointing to `n()` in class B)
- (p4->n())->m();* (pointing to `n()` in class D)
- ④ (pointing to `n()` in class D)
- ① (pointing to `n()` in class D)

Le precedenti definizioni compilano correttamente. Per ognuna delle seguenti istruzioni scrivere nell'apposito spazio:

- NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- ERRORE RUN-TIME** se l'istruzione compila correttamente ma la sua esecuzione provoca un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a 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**.

```

p1->g(); .....
p1->k(); .....
p2->f(); .....
p2->m(); .....
p3->k(); .....
p3->f(); .....
p4->m(); .....
p4->k(); .....
p5->g(); .....
(p3->n())->m(); .....
(p3->n())->n()->g(); .....
(p4->n())->m(); .....
(p5->n())->g(); .....
(dynamic_cast<B*>(p1))->m(); .....
(static_cast<C*>(p2))->k(); .....
(static_cast<B*>(p3->n()))->g(); .....

```