Esercizio

La classe `telefonata` presenta l'inconveniente di non poter rappresentare numeri telefonici che iniziano con lo 0.

Scegliere una **rappresentazione alternativa** per il numero telefonico che risolva l'inconveniente e riscrivere le definizioni dei metodi per tale rappresentazione.

```
class telefonata {
.....
private:
   orario inizio, fine;
   const int numero;
};
```

assegnazione

```
telefonata::telefonata() {numero = 0;}
```

Invocazione **esplicita** del "costruttore di copia" per il campo dati costante `numero`.

```
telefonata::telefonata() : numero(0) {}
```

È possibile richiamare esplicitamente un costruttore per qualsiasi campo dati:

```
telefonata::telefonata(orario i, orario f, int n)
                       : inizio(i), fine(f), numero(n) {}
```

Come vengono eseguite le seguenti istruzioni:

```
orario t1(12,25,33), t2(12,47,12);
telefonata telefonata_a_carlo(t1,t2,333333);
```

usando il precedente costruttore

```
telefonata::telefonata(orario i, orario f, int n)
  {inizio = i; fine = f; numero = n; }
```

e usando il nuovo costruttore

```
telefonata::telefonata(orario i, orario f, int n)
            : inizio(i), fine(f), numero(n) {}
```

nell'ipotesi che non ci siano campi costanti e quindi siano permesse entrambe le definizioni.

```
orario t1(12,25,33), t2(12,47,12);
telefonata telefonata_a_carlo(t1,t2,333333);
```

```
telefonata::telefonata(orario i, orario f, int n)
  {inizio = i; fine = f; numero = n; }
```

2 costruttori di copia di orario, 1 costruttore di copia di int;
2 costruttori di default di orario, 1 costruttore di default per int;
2 assegnazione di orario, 1 assegnazione tra int.

```
telefonata::telefonata(orario i, orario f, int n)
            : inizio(i), fine(f), numero(n) {}
```

2 costruttori di copia di orario, 1 costruttore di copia di int;
2 costruttori di copia di orario, 1 costruttore di copia per int.


WHAT A
WASTE OF
SPACE


WHAT A
WASTE
OF TIME

Passiamo i parametri come riferimenti costanti.

```
telefonata::telefonata(const orario& i,
                       const orario& f, const int& n)
                       : inizio(i), fine(f), numero(n) {}
```

Ciò corrisponde quindi alle seguenti istruzioni:

```
orario inizio(t1); orario fine(t2); int numero(333333);
```

Lista di inizializzazione del costruttore.

In una classe **C** con lista ordinata di campi dati **x1,...,xk**, un costruttore con lista di inizializzazione per i campi dati $x_{i1},...,x_{ij}$ è definito tramite la seguente sintassi:

```
C(T1,...,Tn): xi1(...),..., xij(...) {\\ codice };
```

Il comportamento del costruttore è il seguente:

(A) Ordinatamente per ogni campo dati **xi** ($1 \le i \le k$) viene richiamato un costruttore:
- o esplicitamente tramite una chiamata ad un costruttore **xi(...)** definita nella lista di inizializzazione
- oppure implicitamente (non appare nella lista di inizializzazione) tramite una chiamata al costruttore di default **xi()**

(B) Quindi viene eseguito il codice del costruttore.

[1] Naturalmente, parliamo di costruttori e costruttori di copia anche per campi dati di tipo non classe (primitivo o derivato): la lista di inizializzazione può includere inizializzazioni anche per questi campi dati.

[2] La chiamata implicita al costruttore di default standard per un campo dati di tipo non classe, come al solito, alloca lo spazio in memoria ma lascia indefinito il valore.

[3] L'ordine in cui vengono invocati i costruttori, esplicitamente o implicitamente, è sempre determinato dalla lista ordinata dei campi dati, qualsiasi sia l'ordine delle chiamate nella lista di inizializzazione.

```
class D {
public:
  D() {cout << "D0 ";}
  D(int a) {cout << "D1 ";}
};
class E {
private:
  D d;
public:
  E(): d(3) {cout << "E0 ";}
  E(double a, int b) {cout << "E2 ";}
  E(const E& a): d(a.d) {cout << "Ec ";}
};
class C {
private:
  int z;
  E e;
  D d;
  E* p;
public:
  C(): p(0), e(), z(4) {cout << "C0 ";}
  C(int a, E b): e(3.7,2), p(&b), z(1), d(3) {cout << "C1 ";}
  C(char a, int b): e(), d(2), p(&e)  {cout << "C2 ";}
};
int main() {
  E e; cout << "UNO\n";
  C c; cout << "DUE\n";
  C c1(1,e); cout << "TRE\n";
  C c2('b',2); cout << "QUATTRO";
}
```

Cosa stampa?

È possibile avere dei campi dati di **tipo riferimento** `T&`.

Che succede?

Esercizio: definire un costruttore di default legale per **C**.

```
class E {
private:
  int x;
public:
  E(int z=0): x(z) {}
};

class C {
private:
  int z;
  E x;
  const E e;
  E& r;
  int* const p;
  public:
  C();
};
```

```
C::C(): e(1), r(x), p(new int(0)) {}
```

Esercizio: confrontare gli output dei seguenti due programmi.

```
class C {
public:
  C() {cout << "C0 "; }
  C(const C&) {cout <<"Cc ";}
};

class D {
public:
  C c;
  D() {cout << "D0 ";}
};

int main(){
  D x; cout << endl;
  // stampa ...
  D y(x); cout << endl;
  // stampa ...
}
```

```
class C {
public:
  C() {cout << "C0 "; }
  C(const C&) {cout <<"Cc ";}
};

class D {
public:
  C c;
  D() {cout << "D0 ";}
  D(const D&) {cout <<"Dc ";}
};

int main(){
  D x; cout << endl;
  // stampa ...
  D y(x); cout << endl;
  // stampa ...
}
```

Spiegazione: il costruttore di copia standard di una classe **D** invoca ordinatamente per ogni campo dati **x** di **D** il corrispondente costruttore di copia del tipo di **x**.

Costruttori standard

Possiamo quindi specificare precisamente il comportamento dei costruttori standard di default e di copia per una classe **C** con lista ordinata di campi dati **x1,...,xk**

Costruttore di default standard:

```
C(): x1(),...,xk() {}
```

Costruttore di copia standard:

```
C(const C& obj): x1(obj.x1),...,xk(obj.xk) {}
```

Esercizio

Definire, separando interfaccia ed implementazione, una classe `Raz` i cui oggetti rappresentano un numero razionale $\frac{num}{den}$ (naturalmente, i numeri razionali hanno sempre un denominatore diverso da 0). La classe deve includere:

1. opportuni costruttori;
2. un metodo `Raz inverso()` con il seguente comportamento: se l'oggetto di invocazione rappresenta $\frac{n}{m}$ allora `inverso` ritorna un oggetto che rappresenta $\frac{m}{n}$;
3. un operatore esplicito di conversione al tipo primitivo `double`;
4. l'overloading degli operatori di somma e moltiplicazione;
5. l'overloading dell'operatore di incremento postfisso che, naturalmente, dovrà incrementare di 1 il razionale di invocazione;
6. l'overloading dell'operatore di uguaglianza;
7. l'overloading dell'operatore di output su `ostream`;
8. un metodo `Raz unTerzo()` che ritorna il razionale 0.3333...

```
1  class Raz {
2  private:
3  int num, den; // INV globale: den!=0, numero razione num/den
4
5  public:
6  // convertitore da int => Raz
7  Raz(int n=0, int d=1): num(d==0 ? 0 : n), den(d==0 ? 1 : d) {}
8
9  operator double() const {
10   return static_cast<double> (num) / static_cast<double> (den); // divisione intera
11 }
12
13 };
14
```