PROGRAMMAZIONE AVANZATA

3 Oggetti mutabili

Oggetti immutabili

- Si tratta di oggetti in cui il valore dei campi è impostato una volta per tutte nel costruttore, e poi **non cambia più**
- `rat` e `Complex` sono classi di oggetti immutabili: le operazioni che agiscono su di essi **non modificano** l'oggetto ricevente. Invece, **creano** un nuovo oggetto che rappresenta il risultato dell'operazione

Esempio

```
rat rat::add(const rat& b) const {
  return rat(num * b.den + b.num * den, den * b.den);
}
```

Intuizione

- Sommare $\frac{1}{2}$ e $\frac{3}{4}$ non "distrugge" $\frac{1}{2}$. Il numero $\frac{1}{2}$ continua a esistere anche dopo la somma, che produce come risultato un nuovo numero $\frac{10}{8}$
- Ragionevole in questo contesto, ma non è detto che lo sia sempre

Oggetti mutabili

L'uso di un oggetto mutabile ne **altera lo stato** (non l'identità). Esempi:

- Un conto corrente (i movimenti cambiano il saldo)
- Una penna (l'uso ne consuma l'inchiostro)
- Una lampada (può essere accesa o spenta)

Ci possono essere altri fattori (efficienza) che possono far preferire l'uso di oggetti mutabili laddove si lavora con oggetti logicamente immutabili.

Attenzione

L'uso di oggetti mutabili rende **più difficile** la comprensione del codice e **più facili** gli errori di programmazione:

- Non basta più sapere di che tipo è un oggetto per usarlo "bene"
- Bisogna anche sapere in che stato si trova

Modellazione di una giara

Rappresentazione (campi)

- Una giara ha una **capacità fissa**, stabilita al momento dell'istanziazione
- Una giara ha un **livello variabile**

Operazioni (metodi)

- Deve essere possibile **conoscere lo stato** di una giara (qual è la sua capacità, qual è il suo livello)
- Deve essere possibile **riempire** e **svuotare** completamente una giara
- Deve essere possibile **versare** il contenuto di una giara in un'altra, fino allo svuotamento della prima o al riempimento della seconda, in base a quale dei due eventi avviene per primo

Invarianti

- La capacità di una giara è non negativa
- Il livello di una giara è compreso tra 0 e la sua capacità (estremi inclusi)

Classe Jar

```
class Jar {
private:
  int  capacity; // 0 ⩽ capacity
  int  level;    // 0 ⩽ level ⩽ capacity
public:
  Jar(int);
  int  get_level() const    { return level; }
  int  get_capacity() const { return capacity; }
  void empty();
  void fill();
  void pour_into(Jar&);
};
```

Attenzione

- Ci sono metodi che non hanno il qualificatore `const`
- Questi metodi **possono modificare** lo stato della giara

Metodi di Jar (1/2)

```
Jar::Jar(int c) {
  if (c < 0) throw std::domain_error("invalid capacity");
  capacity = c;
  level = 0;
}

void Jar::empty() {
  level = 0;
}

void Jar::fill() {
  level = capacity;
}
```

Invarianti di classe e oggetti mutabili

- Ogni metodo, all'inizio della sua esecuzione, **può assumere** che l'oggetto sia in uno stato che soddisfa gli invarianti di classe
- Ogni metodo, alla fine della sua esecuzione, **deve garantire** che l'oggetto sia in uno stato che soddisfi gli invarianti di classe

Metodi di `Jar` (2/2)

```
void Jar::pour_into(Jar& b) {
  int q = std::min(level, b.capacity - b.level);
  level -= q;
  b.level += q;
}
```

Note

- L'argomento `b` è un **riferimento non costante** a un oggetto di tipo `Jar`
- Il metodo può **modificare** lo stato di `b` (e dell'oggetto ricevente)
- La funzione di libreria `std::min` calcola il minimo di due valori
- Il metodo deve preservare gli invarianti di **due oggetti distinti**

Curiosità

- Durante l'esecuzione del codice c'è un momento in cui il liquido versato non è né nella prima giara né nella seconda!

Il problema Die Hard

Date due giare di capacità 3 e 5 galloni, riempirne una con 4 galloni (né più né meno) usando esclusivamente le operazioni disponibili in `Jar`.

```
int main() {
  Jar a(5); // a ha capacità 5 galloni
  Jar b(3); // b ha capacità 3 galloni
  b.fill();          // 0 galloni in a, 3 in b
  b.pour_into(a);    // 3 galloni in a, 0 in b
  b.fill();          // 3 galloni in a, 3 in b
  b.pour_into(a);    // 5 galloni in a, 1 in b
  a.empty();         // 0 galloni in a, 1 in b
  b.pour_into(a);    // 1 gallone in a, 0 in b
  b.fill();          // 1 gallone in a, 3 in b
  b.pour_into(a);    // 4 galloni in a, 0 in b
  std::cout << a.get_level() << std::endl;
}
```

Passaggio per valore e per riferimento

Se togliamo & dal tipo del parametro il programma non funziona più!

```
class Jar {
   ...
  void pour_into(Jar);
   ...
};

void Jar::pour_into(Jar b) { ... }
```

Passaggio per **riferimento** `Jar&`

- Al metodo viene trasferito un **riferimento** all'oggetto originale
- Ogni modifica apportata dal metodo agisce direttamente sull'originale

Passaggio per **valore** `Jar`

- Al metodo viene trasferita una **copia** dell'oggetto originale
- Ogni modifica apportata dal metodo agisce sulla copia, non sull'originale

Ritorno per valore o riferimento?

Ritorno “per valore”

```
Jar create_full(int c) {
  Jar a(c);
  a.fill();
  return a; // OK
}
```

Ritorno “per riferimento”

```
Jar& create_full(int c) {
  Jar a(c);
  a.fill();
  return a; // ERRORE
}
```

Qui il ritorno per riferimento è **sbagliato** poiché la giara a viene distrutta alla terminazione della funzione (alcuni compilatori C++ segnalano l’errore)

Il ritorno per riferimento può essere lecito

```
std::ostream& operator<<(std::ostream& os, const rat& a) {
  ...
  return os;
}
```

Qui il ritorno per riferimento è **corretto** poiché `os` è un riferimento a un oggetto creato altrove che sopravvive al ritorno dalla funzione

Esercizi

1. Trovare una soluzione alternativa del problema Die Hard che richieda meno travasi. Scrivere il codice corrispondente.
2. Modificare la classe `Jar` per modellare giare che possono essere aperte o chiuse. In particolare: fare in modo che una giara appena creata sia chiusa; aggiungere metodi `is_open` e `is_closed` per leggere lo stato (aperto/chiuso) di una giara; aggiungere metodi `open` e `close` per aprire/chiudere una giara; modificare i metodi `empty`, `fill` e `pour_into` già visti in modo tale che abbiano un effetto solo su giare aperte.
3. Realizzare una classe `ContoCorrente` dotata delle seguenti operazioni: lettura saldo, versamento, prelievo, lettura del saldo massimo durante la storia del conto. Stabilire la rappresentazione della classe in base alle operazioni che deve supportare. Individuare eventuali invarianti di classe e assicurarsi che siano rispettati.
4. Realizzare una classe `Primi` con un unico metodo `next` (senza argomenti) che, a ogni invocazione, restituisce un elemento diverso della sequenza $2, 3, 5, 7, \ldots$ di numeri primi (alla prima invocazione restituisce 2, alla seconda 3, alla terza 5, e così via). Verificare il funzionamento della classe con una funzione `main` che stampi sul terminale i primi 100 numeri primi usando una istanza di `Primi`.