Metodologie di Programmazione

Corso di Laurea in "Informatica"

09 gennaio 2024

1. (Risoluzione overloading) Mostrare il processo di risoluzione dell'overloading per le seguenti chiamate di funzione. Per ogni chiamata, indicare: l'insieme delle funzioni candidate; l'insieme delle funzioni utilizzabili; la migliore funzione utilizzabile (se esiste); il motivo di eventuali errori di compilazione.

```
#include <string>

namespace N {
  struct C {
    std::string& first();              // funzione #1
    const std::string& first() const; // funzione #2
    std::string& last();               // funzione #3
    const std::string& last() const;  // funzione #4
  }; // class C

  void bar(double);                    // funzione #5
  std::string& bar(int);               // funzione #6
} // namespace N

void foo(N::C& cm, const N::C& cc) {
  std::string& s1 = cm.first();      // chiamata A
  const std::string& s2 = cm.last(); // chiamata B
  std::string& s3 = cc.first();      // chiamata C
  const std::string& s4 = cc.last(); // chiamata D
  bar(s4.size());                    // chiamata E
}
```

2. (Progettazione tipo concreto) La classe templatica `Set` è intesa rappresentare un insieme di elementi di tipo `T`. L'implementazione della classe si basa sulla manipolazione di liste ordinate (senza duplicati). L'interfaccia della classe presenta numerosi problemi; cercare di individuarne il maggior numero e indicare come possono essere risolti (riscrivendo l'interfaccia).

```
template <typename T>
struct Set {
  std::list<T> my_set;   // la lista ordinata di elementi
  Set();                 // costruisce insieme vuoto.
  Set(T t);              // costruisce singoletto {t}
  unsigned int size();   // numero elementi
  bool contains(Set y);  // test di contenimento
  T& min();              // accesso a elemento minimo (primo)
  void erase_min();      // elimina elemento minimo
  void swap(Set y);      // scambia *this con y
  std::ostream operator<<(std::ostream os); // output
  // ...
};
```

3. (Funzione generica) Definire la funzione generica `replace` che, presi in input una sequenza e due valori `old_value` e `new_value` di tipo generico `T`, rimpiazza nella sequenza ogni elemento equivalente a `old_value` con una copia di `new_value` (nota bene: l'algoritmo non produce una nuova sequenza, ma modifica direttamente la sequenza fornita in input.) Elencare i requisiti imposti dall'implementazione sui parametri della funzione.

4. (Gestione risorse) Nell'ipotesi che eventuali errori siano segnalati tramite eccezioni, il seguente codice non ha un comportamento corretto. Individuare almeno un problema, indicando la sequenza di operazioni che porta alla sua occorrenza. Fornire quindi una soluzione basata sull'utilizzo dei blocchi try/catch.

```
void load_and_process(const std::string& conn_params, const std::string& query) {
  Connection conn;        // costr. default: non lancia eccezioni
  conn.open(conn_params); // acquisizione connessione

  Results res; // costr. default: non lancia eccezioni
  res.init();  // acquisizione buffer per risultati

  conn.execute(query, res); // caricamento dati
  process(res);             // elaborazione dati

  res.finish(); // rilascio buffer: non lancia eccezioni
  conn.close(); // rilascio connessione: non lancia eccezioni
}
```

5. (Domande a risposta aperta)

   (a) Fornire un esempio di violazione della ODR (One Definiton Rule) che non può essere rilevato nella fase di compilazione in senso stretto.

   (b) Spiegare cosa si intende quando si dice che una porzione di codice è neutrale rispetto alle eccezioni. Quali sono i casi in cui non si deve essere neutrali?

   (c) Spiegare brevemente perché, nelle interfacce basate su polimorfismo dinamico, è opportuno definire i distruttori virtuali non puri.

   (d) Fornire un semplice esempio in cui l'uso dello specificatore `override` causa un errore a tempo di compilazione.

   (e) Aiutandosi con un semplice esempio, spiegare in cosa consiste il principio di separazione delle interfacce (ISP).

6. Un sito per il commercio elettronico gestisce acquisti (e rimborsi) mediante alcuni metodi di pagamento usando codice come il seguente:

```
struct Metodo_Pagamento {
  const char* nome_metodo() const;
  void addebita_spesa(const Importo& i);
  void rimborso(const Importo& i);
  // ...
private:
  enum Metodo { A_PAY, B_PAY };
  Metodo metodo;
  void pagamento_A_PAY(const Importo&);
  void rimborso_A_PAY(const Importo&);
  void addebita_su_B_PAY(const Importo&);
  void accredita_su_B_PAY(const Importo&);
  // ...
};

const char* Metodo_Pagamento::nome_metodo() const {
  switch (metodo) {
  case A_PAY:
    return "A_PAY Virtual Card";
  case B_PAY:
    return "B_PAY E-Wallet";
  }
}

void Metodo_Pagamento::addebita_spesa(const Importo& i) {
  switch (metodo) {
  case A_PAY:
    pagamento_A_PAY(i);
    break;
  case B_PAY:
    addebita_su_B_PAY(i);
    break;
  }
}

void Metodo_Pagamento::rimborso(const Importo& spesa) {
  switch (metodo) {
  case A_PAY:
    rimborso_A_PAY(i);
    break;
  case B_PAY:
    accredita_su_B_PAY(i);
    break;
  }
}
```

Nell'ipotesi che l'insieme dei metodi di pagamento supportati sarà in futuro esteso, impostare una soluzione alternativa più aderente ai principi della progettazione orientata agli oggetti. Mostrare le dipendenze tra le classi e la corrispondente implementazione dei metodi mostrati sopra.