Appello 09-01-24

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.

```cpp
#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
}
```

	f. candidate	f. utilizzabili	f. migliore	motivazione	note
A	#1, #2	#1	#1		la #2 prende come parametro un oggetto const perciò `cm` dovrebbe essere qualificato, mentre #1 è un match perfetto
B	#3, #4	#3	#3		stesso ragionamento della chiamata A
C	#1, #2	#2	#2		dà errore a tempo di compilazione in quanto stiamo assegnando un valore const a una variabile non const
D	#3, #4	#4	#4		
E					non stiamo dichiarando il namespace perciò le funzioni #5 e #6 non sono visibili, se il namespace fosse dichiarata ci sarebbe comunque ambiguità in quanto sia la #5 che la #6 sono compatibili

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
    // ...
};
```

correzione interfaccia:

```
template <typename T>
struct Set {
    std::list<T> my_set;
    Set();
    Set(T& t);
    unsigned int size() const;
    bool contains(const Set& y) const;
    const T& min() const;
    void erase_min();
    void swap(Set& y);
    std::ostream& operator<<(std::ostream& os) const;
};
```

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.

```
template <typename Iter, typename T>
void replace(Iter first, Iter last, T& old_value, T& new_value) {
    for ( ; first != last; ++first) {
        if (*(first) == old_value)
            *(first) = new_value;
    }
}
```

Requisiti:

- Iter deve supportare la copia perché stiamo facendo un passaggio per valore;
- Iter deve supportare l'operazione di incremento (++first) e il confronto binario (first ≠ last);
- Iter deve supportare la dereferenziazione (* first);

- Iter deve permettere la scrittura sulla sequenza; / deve supportare l'operatore di assegnamento;
- il tipo T deve poter essere confrontabile con il tipo puntato da Iter.

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
}
```

```
void load_and_process(const std::string& conn_params, const std::string& query) {
    Connection conn;
    Result res;

    conn.open(conn_params);
    try {
        res.init();
        try {
            conn.execute(query, res);
            process(res);
            res.finish();
            conn.close();
        } catch(...) {
            res.finish();
            throw;
        }
    } catch(...) {
        conn.close();
        throw;
    }
}
```

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

```
//interfaccia metodo pagamento
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.

Soluzione:

Metodo_Pagamento.hh

```
#ifndef METODO_PAGAMENTO_HH
#define METODO_PAGAMENTO_HH

struct Metodo_Pagamento {
```

```
    virtual const char* nome_metodo() const = 0;
    virtual void addebita_spesa(const Importo& i) = 0;
    virtual void rimborso(const Importo& i) = 0;

    virtual ~Metodo_Pagamento() = default;
};

#endif METODO_PAGAMENTO_HH
```

A_PAY.hh

```
#include "Metodo_Pagamento.hh"
#ifndef A_PAY_HH
#define A_PAY_HH

struct A_PAY : public Metodo_Pagamento {
private:
    void pagamento_A_PAY(const Importo&);
    void rimborso_A_PAY(const Importo&);

public:
    const char* nome_metodo() const override;
    void addebita_spesa(const Importo& i) override;
    void rimborso(const Importo& i) override;

    ~A_PAY() = default;
};

#endif
```

B_PAY.hh

```
#include "Metodo_Pagamento.hh"
#ifndef B_PAY_HH
#define B_PAY_HH

struct B_PAY : public Metodo_Pagamento {
private:
    void addebita_su_B_PAY(const Importo&);
    void accredita_su_B_PAY(const Importo&);

public:
    const char* nome_metodo() const override;
    void addebita_spesa(const Importo& i) override;
    void rimborso(const Importo& i) override;

    ~B_PAY() = default;
};

#endif
```

A_PAY.cc

```cpp
#include "A_PAY.hh"

struct A_PAY : public Metodo_Pagamento {
private:
    void pagamento_A_PAY(const Importo&) {
        //...
    }
    void rimborso_A_PAY(const Importo&) {
        //...
    }

public:
    const char* nome_metodo() const override {
        return "A_PAY Virtual Card";
    }
    void addebita_spesa(const Importo& i) override {
        pagamento_A_PAY(i);
    }
    void rimborso(const Importo& i) override {
        rimborso_A_PAY(i);
    }

    ~A_PAY() = default;
};
```

B_PAY.cc

```cpp
#include "B_PAY.hh"

struct B_PAY : public Metodo_Pagamento {
private:
    void addebita_su_B_PAY(const Importo&) {
        //...
    }
    void accredita_su_B_PAY(const Importo&) {
        //...
    }

public:
    const char* nome_metodo() const override {
        return "B_PAY E-Wallet";
    }
    void addebita_spesa(const Importo& i) override {
        addebita_su_B_PAY(i);
    }
    void rimborso(const Importo& i) override {
        accredita_su_B_PAY(i);
    }

    ~B_PAY() = default;
}
```