Esercizio 1 - Funzione di gestione documenti

Si assumano le seguenti specifiche riguardanti una libreria di gestione documenti.

(a) `Document` è la classe base polimorfa di tutti i documenti. La classe `Document` rende disponibile un metodo `int getSize() const` con il seguente comportamento: `doc.getSize()` ritorna la dimensione in KB del documento `doc`. Inoltre, la classe `Document` rende disponibile un metodo `bool isReadOnly() const` con il seguente comportamento: `doc.isReadOnly()` ritorna `true` se il documento `doc` è in sola lettura, altrimenti ritorna `false`.

(b) `TextDocument` è derivata direttamente da `Document` ed è la classe dei documenti testuali. La classe `TextDocument` rende disponibile un metodo `std::string getFormat() const` con il seguente comportamento: `text.getFormat()` ritorna il formato del documento testuale `text` (ad esempio "txt", "md", "docx"). Inoltre, `TextDocument` rende disponibile un metodo `void encrypt()` con il seguente comportamento: `text.encrypt()` cripta il documento testuale `text`.

(c) `ImageDocument` è derivata direttamente da `Document` ed è la classe dei documenti immagine. La classe `ImageDocument` rende disponibile un metodo `int getResolution() const` con il seguente comportamento: `img.getResolution()` ritorna la risoluzione in DPI dell'immagine `img`. Inoltre, `ImageDocument` rende disponibile un metodo `void resize(double factor)` con il seguente comportamento: `img.resize(factor)` ridimensiona l'immagine `img` di un fattore `factor`.

Definire una funzione `list<Document*> processDocs(vector<Document*>&, int)` tale che in ogni invocazione `processDocs(docs, limit)`:

(1) Per ogni puntatore `p` contenuto nel vector `docs`:

- Se `p` punta ad un oggetto che è un `TextDocument` con formato "docx" e non è in sola lettura, viene criptato.
- Se `p` punta ad un oggetto che è un `ImageDocument` con risoluzione maggiore di 300 DPI e dimensione maggiore di `limit` KB, viene ridimensionato di un fattore 0.5.
- Quindi, se l'oggetto risultante ha una dimensione maggiore di `limit` KB, il puntatore viene rimosso dal vector `docs`.

(2) L'invocazione `processDocs(docs, limit)` deve ritornare una list contenente puntatori a copie di tutti i documenti che sono stati rimossi dal vector `docs`.

```
list<Document*> processDocs(vector<Document*>& docs,
int limit){

        list<Document*> l;

        for(auto p = docs.begin(); p != docs.end(); ++p){
                TextDocument* t =
                dynamic_cast<TextDocument*>(*p);

                if(t && t->getFormat() == "docx" &&
                !t->isReadonly())
                        t->encrypt();
                else{
                        // Se non criptato, rimuovilo
                        l.push_back(t->clone());
                        // Cancellarlo
                        p = docs.erase(p);
                        // Equivalente → docs.erase(p);
                }

                ImageDocument* i =
                dynamic_cast<ImageDocument*>(*p);

                if(i && i->getResolution() > 300
                && i->getSize() > limit)
                        i->resize(0.5);

        }
}
```

```
Puntando a **copie** del puntatore
(se serve clone, "copia" / "copie" sta
in grassetto)

- ->clone();

TextDocument ⟷ clone?

virtual TextDocument* clone() const{
        return new TextDocument(*this);
}
```

Esercizio 2 - Funzione per la gestione di componenti UI

Si assumano le seguenti specifiche riguardanti una libreria per interfacce utente.

(a) `UIComponent` è la classe base polimorfa di tutti i componenti dell'interfaccia utente. La classe `UIComponent` rende disponibile un metodo `bool isVisible() const` con il seguente comportamento: `comp.isVisible()` ritorna `true` se il componente `comp` è visibile, altrimenti ritorna `false`. Inoltre, la classe `UIComponent` rende disponibile un metodo `void setVisible(bool v)` con il seguente comportamento: `comp.setVisible(v)` imposta la visibilità del componente `comp` al valore `v`.

(b) `InteractiveComponent` è derivata direttamente da `UIComponent` ed è la classe base dei componenti interattivi. La classe `InteractiveComponent` rende disponibile un metodo `bool isEnabled() const` con il seguente comportamento: `ic.isEnabled()` ritorna `true` se il componente interattivo `ic` è abilitato, altrimenti ritorna `false`. Inoltre, `InteractiveComponent` rende disponibile un metodo `void setEnabled(bool e)` con il seguente comportamento: `ic.setEnabled(e)` imposta lo stato di abilitazione del componente interattivo `ic` al valore `e`.

(c) `Button` è derivata direttamente da `InteractiveComponent` ed è la classe dei pulsanti. La classe `Button` rende disponibile un metodo `std::string getLabel() const` con il seguente comportamento: `btn.getLabel()` ritorna l'etichetta del pulsante `btn`. Inoltre, `Button` rende disponibile un metodo `void setLabel(const std::string& label)` con il seguente comportamento: `btn.setLabel(label)` imposta l'etichetta del pulsante `btn` al valore `label`.

(d) `TextInput` è derivata direttamente da `InteractiveComponent` ed è la classe dei campi di input testuale. La classe `TextInput` rende disponibile un metodo `bool isReadOnly() const` con il seguente comportamento: `input.isReadOnly()` ritorna `true` se il campo di input `input` è in sola lettura, altrimenti ritorna `false`. Inoltre, `TextInput` rende disponibile un metodo `void setReadOnly(bool ro)` con il seguente comportamento: `input.setReadOnly(ro)` imposta lo stato di sola lettura del campo di input `input` al valore `ro`.

Definire una funzione `std::pair<vector<UIComponent*>, vector<Button*>> processUI(const list<UIComponent*>&, const std::string&)` tale che in ogni invocazione `processUI(components, prefix)`:

(1) Per ogni puntatore `p` contenuto nella lista `components`:

- Se `p` punta ad un oggetto che è un `Button` visibile, modifica l'etichetta del `Button` aggiungendo `prefix` all'inizio dell'etichetta corrente.
- Se `p` punta ad un oggetto che è un `TextInput` visibile e non è in sola lettura, lo imposta come in sola lettura.
- Se `p` punta ad un oggetto che è un `InteractiveComponent` non visibile, lo rende visibile ma disabilitato.

(2) L'invocazione `processUI(components, prefix)` deve ritornare una coppia composta da:

- Un vector contenente puntatori a copie di tutti gli `UIComponent` che erano visibili prima dell'applicazione delle modifiche del punto (1).
- Un vector contenente puntatori a copie di tutti i `Button` le cui etichette sono state modificate.

```cpp
std::pair<vector<UIComponent*>, vector<Button*>>
processUI(const list<UIComponent*>& list, const std::string& prefix) {

        vector<UIComponent*> ret1;
        vector<Button*> ret2;

        // iteriamo su list
        for(auto i: list) // range-based -> C++23
        // identico, ma più corto

        Button* b = dynamic_cast<Button*>(*i);
        if(b && b->isVisible()) {
                // Opzionale
                string originalLabel = btn->getLabel();
                b->setLabel(prefix + originalLabel);

                // Static_cast -> Dopo dynamic_cast "per efficienza" -> Così fa Ranzy!
                ret2.push_back(static_cast<Button*>(b->clone()); // clone = copia del puntatore
        }

        if(dynamic_cast<TextInput*>(*p) && dynamic_cast<TextInput*>(*p)->isVisible() &&
        !dynamic_cast<TextInput*>(*p)->isReadonly(){
                dynamic_cast<TextInput*>(*p)->setReadonly(true);

                // alternativa -> chiamo su const perché non faccio side effects
                const* TextInput* txt = dynamic_cast<const TextInput*>(*p);
                txt->setReadonly(true);
        }

        InteractiveComponent* ic = dynamic_cast<InteractiveComponent*>(*i);
        if(ic && !ic->isVisible(){
                ic->setVisible(true);
                ic->setEnabled(false);

                // pushamo nella coppia
                ret1.push_back(ic->clone());
        }

        return {ret1, ret2};
}
```

```cpp
template<class T>
class SmartPtr {
private:
    T* ptr;
public:
    SmartPtr(T* p = nullptr): ptr(p) { cout << "SmartPtr() "; }
    SmartPtr(const SmartPtr& s): ptr(s.ptr) { cout << "SmartPtrCopy() "; }
    ~SmartPtr() { cout << "~SmartPtr() "; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
};

template<class T>
vector<int> analizza(const vector<T>& v, function<bool(T)> pred) {
    vector<int> r;
    for(int i = 0; i < v.size(); ++i)
        if(pred(v[i])) r.push_back(i);
    return r;
}

template<class T>
int conta(const vector<T>& v, T val) {
    auto risultato = analizza(v, [val](T x) { return x > val; });
    return risultato.size();
}

int somma(const vector<int>& v) {
    int s = 0;
    for(auto x : v) s += x;
    return s;
}

vector<int> v1 = {5, -2, 8, 3, 5, 2, 7};
vector<double> v2 = {2.5, 4.3, -1.2, 0.8, 6.7};
```

Queste definizioni compilano correttamente (con opportuni `#include` e using). Per ognuno dei seguenti statement scrivere nell'apposito spazio:

- NON COMPILA se la compilazione dello statement provoca un errore;
- UNDEFINED se lo statement compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore run-time;
- se lo statement compila ed esegue correttamente (senza undefined behaviour o errori 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.

```cpp
cout << conta(v1, 3);
cout << conta(v2, 1.5);
cout << somma(analizza(v1, [](int x){ return x % 2 == 0; }));
SmartPtr<int> p1(new int(5)); cout << *p1;
SmartPtr<int> p2 = p1; cout << *p2;
```

```cpp
#include<iostream>
#include<string>
using namespace std;

class Numero {
public:
    operator int() const { return 42; }
};

template<class T> class Parser; // dichiarazione incompleta

template<class T1, class T2 = int, int K = 0>
class Dato {
    friend class Parser<T1>;
private:
    T1 val1;
    T2 val2;
    int count;
public:
    Dato(int c = K): count(c) {}
};

template<class T>
class Parser {
public:
    void f() const { Dato<T> d; cout << d.val1 << endl; }
    void g() const { Dato<int, T> d; cout << d.val2 << endl; }
    void h() const { Dato<T, double, 3> d; cout << d.count << endl; }
    void m() const { Dato<char, T> d; cout << d.val2 << endl; }
    void n() const { Dato<string, T, 5> d; cout << d.count << endl; }
    void o() const { Dato<Numero, T, 7> d(10); cout << d.count << endl; }
};
```

Determinare se i seguenti main() compilano correttamente o meno barrando la corrispondente scritta.

`int main() { Parser<int> p1; p1.f(); }`	COMPILA ☒ NON COMPILA ☐
`int main() { Parser<string> p2; p2.f(); }`	COMPILA ☒ NON COMPILA ☐
`int main() { Parser<double> p3; p3.g(); }`	COMPILA ☐ NON COMPILA ☒
`int main() { Parser<char> p4; p4.h(); }`	COMPILA ☒ NON COMPILA ☐
`int main() { Parser<double> p5; p5.h(); }`	COMPILA ☒ NON COMPILA ☐
`int main() { Parser<int> p6; p6.m(); }`	COMPILA ☐ NON COMPILA ☒
`int main() { Parser<char> p7; p7.n(); }`	COMPILA ☐ NON COMPILA ☒
`int main() { Parser<double> p8; p8.n(); }`	COMPILA ☐ NON COMPILA ☒
`int main() { Parser<int> p9; p9.o(); }`	COMPILA ☒ NON COMPILA ☐
`int main() { Parser<Numero> p10; p10.o(); }`	COMPILA ☒ NON COMPILA ☐

Esercizio Cosa Stampa (ATTENZIONE: NUMERARE TUTTE LE RISPOSTE)

```cpp
class A {
public: A() {cout << "A ";}
};

class B: virtual public A {
public: B() {cout << "B ";}
};

class C: virtual public A {
public: C(): A() {cout << "C ";}
};

class D: virtual public B, virtual public C {
public: D(): C(), B() {cout << "D ";}
};
```

Le precedenti classi compilano correttamente (con gli opportuni `include` e `using`). Si supponga che le precedenti classi siano visibili ai seguenti frammenti di codice. Per ognuno di questi frammenti di codice, scrivere nel foglio le **risposte numerate da 1 a 6**:

- **NON COMPILA** se la compilazione del codice provoca un errore;
- **UNDEFINED BEHAVIOUR** se il codice compila correttamente ma l'esecuzione di `main() { F f; }` provoca un undefined behaviour o un errore a run-time;
- se invece il codice compila correttamente e l'esecuzione di `main() { F f; }` non provoca un errore a run-time, la stampa prodotta in output su `cout`; se non provoca alcuna stampa si scriva **NESSUNA STAMPA**.

```cpp
class E: virtual public B {
  public: E() {cout << "E ";}
};

class F: public E,  virtual public C {
  public: F() {cout << "F ";}
};
```

RISPOSTA 1: F f → A B E

```cpp
class E: public B {
  public: E() {cout << "E ";}
};

class F: virtual public E,  virtual public C {
  public: F() {cout << "F ";}
};
```

RISPOSTA 2: F f → A B E C F

```cpp
class E: public D {
  public: E(): B() {cout << "E ";}
};

class F: public E {
  public: F() {cout << "F ";}
};
```

RISPOSTA 3: A B C D B E

```cpp
class E: public D {
  public: E(): B() {cout << "E ";}
};

class F: virtual public E {
  public: F() {cout << "F ";}
};
```

RISPOSTA 4: A B C D B F

```cpp
class F: public B, virtual public C {
  public: F()  {cout << "F ";}
};
```

RISPOSTA 5: A C B F

```cpp
class F: public B {
  public: E() {cout << "E ";}
};

class F: public E,  virtual public C {
  public: F() {cout << "F ";}
};
```

RISPOSTA 6: A C B E F