Appello 12-06-2024

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.

```
namespace N {
    struct S {
        S(const char*); // funzione #1
        char* get(); // funzione #2
        const char* get() const; // funzione #3
    }; // struct S
    char* foo(int); // funzione #4
} // namespace N

void foo(N::S& s, const N::S& cs) { // funzione #5
    const char* g1 = s.get(); // chiamata A
    char* cg2 = cs.get(); // chiamata B
}

int main() {
    N::S s("init"); // chiamata C
    foo(s, s); // chiamata D
    foo(12345); // chiamata E
}
```

3. (Gestione risorse) La classe seguente contiene errori inerenti la corretta gestione delle risorse. Individuare almeno un problema, spiegandone brevemente le conseguenze e fornendo un semplice esempio di codice utente che lo causa. Correggere quindi la classe per impedire il verificarsi di tale problema.

```
#include <string>
class S {
    int* pi;
    std::string str;
public:
    explicit S(const std::string& s) : pi(new int(42)), str(s) { }
    ~S() { delete pi; }
};
```

Un problema potrebbe essere all'interno del costruttore, se la costruzione di pi va a buon fine, ma quella della stringa no, allora l'oggetto non avrà concluso la sua costruzione, perciò non sarà possibile invocare il distruttore su di esso causando un memory leak in quanto l'oggetto pi è stato comunque allocato. Inoltre la mancanza della definizione per il costruttore di copia causerà probabilmente una double free su pi.

```
int main() {
    S s("test");
```

```
    S t = s; //quando i due oggetti t e s andranno distrutti verrà fatta una doppia
delete su pi
}
```

Possibile soluzione

```
/* #include <string>
class S {
    int* pi;
    std::string str;
public:
    explicit S(const std::string& s) : pi(nullptr) {
        pi(new int(42));
        try {
            str(s);
        } catch (...) {
            delete pi;
            throw;
        }
    }

    ~S() { delete pi; }
} */
```

con gli smart pointers :

```
#include <string>
#include <memory>

class S {
    std::unique_ptr<int> pi; //non è possibile fare la copia
    std::string str;
public:
    explicit S(const std::string& s) : pi(std::make_unique<int>(42)), str(s){}
};
```

5. Indicare l'output prodotto dal seguente programma:

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

class Base {
public:
    Base() { cout << "Costruttore Base" << endl; } //#1
    virtual void foo(int) { cout << "Base::foo(int)" << endl; } //#2
    virtual void bar(int) { cout << "Base::bar(int)" << endl; } //#3
    virtual void bar(double) { cout << "Base::bar(double)" << endl; } //#4
    virtual ~Base() { cout << "Distruttore Base" << endl; } //#5
};

class Derived : public Base {
public:
```

```
    Derived() { cout << "Costruttore Derived" << endl; } //#6
    void foo(int) { cout << "Derived::foo(int)" << endl; } //#7
    void bar(int) const { cout << "Derived::bar(int) const" << endl; } //#8
    void bar(double) const { cout << "Derived::bar(double) const" << endl; } //#9
    ~Derived() { cout << "Distruttore Derived" << endl; } //#10
};

int main() {
    Derived derived; //Costruttore Base, Costruttore Derived
    Base base; //Costruttore Base
    Base& base_ref = base;
    Base* base_ptr = &derived;
    Derived* derived_ptr = &derived;
    cout << "=== 1 ===" << endl; //= = = 1 = = =
    base_ptr->foo(12.0); //Derived::foo(int) #7
    base_ref.foo(7); //Base::foo(int) #2
    base_ptr->bar(1.0); //Derived::bar(double) const #9 //Base::bar(double)  (forse
nel caso di -> vado a vedere il tipo statico)
    derived_ptr->bar(1.0); //Derived::bar(double) const #9
    derived.bar(2); //Derived::bar(int) const #8
    cout << "=== 2 ===" << endl; // = = = 2 = = =
    base.bar(1); //Base::bar(int) #3
    derived.bar(-1.0); //Derived::bar(double) const #9
    derived.foo(0.3); //Derived::foo(int) #7
    base_ptr->bar('\n'); //Derived::bar(int) const #8 //Base::bar(int)
    return 0;
}
//Distruttore Base
//Distruttore Derived
//Distruttore Base
```