Esercizi Lambda

Esercizio 1: Cattura e modifiche

```cpp
int main() {
    int x = 10;
    int y = 20;

    auto l1 = [=]() mutable { x = 30; y = 40; return x + y; };
    auto l2 = [&]() { x = 50; y = 60; return x + y; };

    cout << l1() << endl;
    cout << "x: " << x << ", y: " << y << endl;

    cout << l2() << endl;
    cout << "x: " << x << ", y: " << y << endl;

    return 0;
}
```

Cosa stampa?

Esercizio 2: Ricorsione con lambda

```cpp
int main() {
    std::function<int(int)> factorial = [&factorial](int n) {
        return n <= 1 ? 1 : n * factorial(n - 1);
    };

    cout << factorial(5) << endl;

    auto fibonacci = [](int n) {
        std::function<int(int)> fib = [&fib](int n) {
            return n <= 1 ? n : fib(n-1) + fib(n-2);
        };
        return fib(n);
    };

    cout << fibonacci(7) << endl;

    return 0;
}
```

Cosa stampa?

Vero o Falso

Esercizio 3: Vero o Falso su ereditarietà e polimorfismo

1. `dynamic_cast<D*>(nullptr)` ritorna sempre `nullptr` anche se D non è una classe polimorfa.
2. Se `A` e `B` sono classi e `B` deriva da `A`, allora un puntatore a `B` può essere assegnato a un puntatore a `A` senza cast esplicito.
3. Se un distruttore virtuale è definito nella classe base, allora tutte le classi derivate devono ridefinire il distruttore.
4. L'operatore `typeid` può essere usato su tipi non polimorfi, ma in questo caso controlla solo il tipo statico.
5. In un'ereditarietà a diamante con base virtuale, il costruttore della classe base viene chiamato una sola volta, direttamente dalla classe più derivata.
6. L'operatore `sizeof` applicato a una classe ritorna la somma delle dimensioni di tutti i membri più eventuali padding bytes.
7. Un metodo virtuale può essere final e pure allo stesso tempo.
8. Se `B` deriva pubblicamente da `A` e `C` deriva pubblicamente da `B`, allora `dynamic_cast<A*>(new C())` è sempre valido.
9. Se un metodo è dichiarato `const`, può chiamare metodi non-const della stessa classe.
10. L'operatore `static_cast` può convertire un puntatore a void in un puntatore a qualsiasi tipo.

Gerarchie Problematiche

Esercizio 5: Gerarchia con ambiguità

```
class Base {
public:
    virtual void foo() { cout << "Base::foo" << endl; }
    int x = 10;
};

class Derived1 : public Base {
public:
    void foo() override { cout << "Derived1::foo" << endl; }
    int x = 20;
};

class Derived2 : public Base {
public:
    void foo() override { cout << "Derived2::foo" << endl; }
    int x = 30;
};
```

```
class Diamond : public Derived1, public Derived2 {
public:
    void foo() override { cout << "Diamond::foo" << endl; }
};

int main() {
    Diamond d;
    d.foo();  // OK
    cout << d.x << endl;  // Ambiguo

    Base* b = &d;  // Quale Base?
    return 0;
}
```

Individua e correggi i problemi in questa gerarchia.

Esercizio 6: Ereditarietà privata e nascondimento di membri

```
class Base {
public:
    virtual void foo() { cout << "Base::foo" << endl; }
    void bar() { cout << "Base::bar" << endl; }
};

class Derived : private Base {
public:
    void foo() override { cout << "Derived::foo" << endl; }
    void bar() { cout << "Derived::bar" << endl; }
    using Base::foo;  // Cosa fa questa riga?
};

int main() {
    Derived d;
    d.foo();
    d.bar();

    Base* b = &d;  // Compila?
    Base& r = d;   // Compila?

    return 0;
}
```

Quali linee compilano e quali no? Perché?

Concetti Fuori dagli Schemi

Esercizio 7: CRTP (Curiously Recurring Template Pattern)

```
template <typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }

    void implementation() {
        cout << "Base implementation" << endl;
    }
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        cout << "Derived implementation" << endl;
    }
};

int main() {
    Derived d;
    d.interface();

    Base<Derived>* b = &d;
    b->interface();
    b->implementation();

    return 0;
}
```

Cosa stampa? Perché questo pattern è utile?

Esercizio 8: Costruzione di oggetti in-place con sfinae

```
#include <iostream>
#include <type_traits>
#include <utility>

class NonCopyable {
public:
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    void use() { std::cout << "NonCopyable used" << std::endl; }
};

template <typename T, typename... Args>
```

```cpp
std::enable_if_t<std::is_constructible<T, Args...>::value, T*>
create(Args&&... args) {
    return new T(std::forward<Args>(args)...);
}

template <typename T, typename... Args>
std::enable_if_t<!std::is_constructible<T, Args...>::value, T*>
create(Args&&... args) {
    std::cout << "Cannot construct T with given arguments" << std::endl;
    return nullptr;
}

int main() {
    auto* nc1 = create<NonCopyable>();
    if (nc1) nc1->use();

    NonCopyable src;
    auto* nc2 = create<NonCopyable>(src);  // Tentativo di copia
    if (nc2) nc2->use();

    delete nc1;
    delete nc2;  // Safe anche se nullptr

    return 0;
}
```

Cosa viene stampato? Perché SFINAE è utile in questo contesto?

Questi esercizi coprono vari aspetti avanzati della programmazione C++ che potrebbero essere utili per approfondire la tua comprensione. Fammi sapere se preferisci altri tipi di esercizi o se hai domande su questi.