Prog. Avanzata

A sinistra Mirko che si sega sapendo che Giulia non imparerà una sega con Roberti.

“

Essere o non essere Roberti, questo non è minimamente un dilemma.

”

~ Mirkolo 🍻

Costruttori

I costruttori sono metodi speciali che vengono chiamati automaticamente quando un oggetto viene creato. I costruttori possono essere utilizzati per inizializzare gli attributi di un oggetto.

Costruttore "particolare": lista di inizializzazione

In C++ è possibile creare una lista di inizializzazione per inizializzare gli attributi di un oggetto. Questa lista di inizializzazione viene posta tra i due punti `:` e viene utilizzata per inizializzare gli attributi della classe.

Costruttore "particolare": lista di inizializzazione

Il motivo per cui è preferibile utilizzare la lista di inizializzazione è che gli attributi vengono inizializzati prima che il corpo del costruttore venga eseguito. Per le costanti o i riferimenti, la lista di inizializzazione è l'unico modo per inizializzarli.

```cpp
class Foo {
    public:
        int a;
        int b;

        Foo(int a, int b) : a(a), b(b) { }
};

int main() {
    Foo f(42, 69);
    std::cout << f.a << std::endl; // Output: 42
    std::cout << f.b << std::endl; // Output: 69
}
```

Membri statici

In C++ possiamo dichiarare attributi e metodi statici all'interno di una classe. Questi membri sono condivisi tra tutte le istanze della classe.
Per fare ciò, basta aggiungere la keyword `static` davanti al membro.

Attributi statici

Gli attributi statici esistono in un'unica copia per tutta la classe. Questo significa che se un'istanza modifica un attributo statico, tutte le altre istanze vedranno la modifica.

Metodi statici

I metodi statici sono metodi che possono essere chiamati senza creare un'istanza della classe. I metodi statici non possono accedere ad attributi non statici.

```cpp
class Counter {
    public:
        static int count;

        static void incrementCounter() { count++; }
};

int Counter::count = 0;

int main() {
    Counter c1;
    std::cout << Counter::count << std::endl; // Output: 0
    Counter::incrementCounter();
    std::cout << Counter::count << std::endl; // Output: 1
}
```

La costanza

In C++ possiamo dichiarare attributi e metodi costanti all'interno o esternamente ad una classe. Utiliziamo il modificatore `const` per dichiarare un attributo o un metodo costante.

Attributi costanti

Gli attributi costanti non possono essere modificati dopo la loro inizializzazione. Questo significa che una volta assegnato un valore ad un attributo costante, non possiamo più modificarlo.

```cpp
class Foo {
    public:
        const int a;
        static const int b;

        Foo() : a(0) {}
};

const int Foo::b = 42;

int main() {
    Foo f;
    std::cout << f.a << std::endl; // Output: 0
    std::cout << Foo::b << std::endl; // Output: 42
}
```

Costanza e puntatori

Quando dichiariamo un puntatore costante, il puntatore non può essere modificato, ma il valore a cui punta può essere modificato. Se invece dichiariamo un puntatore costante a dati costanti, né il puntatore né il valore a cui punta possono essere modificati. Analogamente, possiamo dichiarare un puntatore a dati costanti.

```
int main(int argc, char* argv) {
    int a = 42;
    const int* b = &a;          // Puntatore a dati non costanti
    const int* const c = &a;    // Puntatore costante a dati costanti
    int* const d = &a;          // Puntatore costante a dati non costanti

    *d = 0;                     // OK
    d = nullptr;                // Errore
    b = nullptr;                // OK
    *b = 0;                     // Errore
    c = nullptr;                // Errore
    *c = 0;                     // Errore
}
```

Metodi costanti

I metodi costanti non possono modificare gli attributi della classe. Questo significa che un metodo costante non può modificare gli attributi della classe a meno che non siano dichiarati come `mutable`.

```cpp
class Foo {
    public:
        int a;
        mutable int b;

        void setA(int a) : a(a), b(0) {  }
        int getA() const { return a; }
        int modifyB() const { b++; }
};

int main() {
    Foo f;
    f.setA(42);
    std::cout << f.getA() << std::endl; // Output: 42
    f.modifyB();
    std::cout << f.b << std::endl; // Output: 1
}
```

Metodi `inline`

In C++ possiamo dichiarare un metodo `inline` per indicare al compilatore di sostituire la chiamata del metodo con il corpo del metodo stesso. Questo può essere utile per ottimizzare il codice.

```cpp
class Foo {
    public:
        int a;
        int b;

        inline int sum() { return a + b; }
};

int main() {
    Foo f;
    f.a = 42;
    f.b = 69;
    std::cout << f.sum() << std::endl; // Output: 111
}
```

Esercizio 1

Si scriva una classe `Point` che rappresenti un punto nello spazio R^2. La classe deve avere due attributi `x` e `y` di tipo `double` e un costruttore che inizializzi i due attributi. Si scriva un metodo `distance` che calcoli la distanza euclidea tra due punti. Si definisca un attributo statico `counter` che tenga traccia del numero di istanze create.

Esercizio 2

Si scriva un pgm C++ che definisca la classe `Veicolo` e la classe figlia `Auto`. La classe `Veicolo` deve avere attributi `marca`, `modello` e `cilindrata`. La classe `Auto` deve avere un attributo `numeroPorte` (MAX 5). Si definisca `print_info` e si faccia l'overloading dell'operatore `<<` (per sport).