Lezione 09: Classi, oggetti e template

Fondamenti delle Classi in C++

Accesso ai Membri

- Si usa `.` per oggetti e `→` per puntatori.
- Una **classe** è un **namespace** che contiene membri.
- I membri possono essere:
  - `public`: interfaccia accessibile dall'esterno.
  - `private`: implementazione interna.
- Una `struct` è come una `class`, ma con membri `public` di default.

```
Esempio es{5};
int a = es.metodo(10);

Esempio* ptr = &es;
int b = ptr→metodo(20);
```

Dichiarazioni e Metodi

- Metodi possono essere:
  - Definiti **dentro** la classe (inline, per metodi brevi).
  - Definiti **fuori**, dopo la dichiarazione.
- Un metodo può essere marcato `const` se non modifica l'oggetto.
- Gli oggetti `const` possono invocare solo metodi `const`.

```
class Data {
   int d, m, y;
```

```
public:
    int giorno() const { return d; }
};

const Data d{1, 1, 2025};
int giorno = d.giorno(); // ok
// d.setGiorno(2);       // errore: metodo non `const`
```

Costruttori, Distruttori e Inizializzazione

Costruttori

- Vengono chiamati alla creazione dell'oggetto.
- Possono avere **valori di default**.
- Supportano **overloading**.
- È possibile inizializzare direttamente gli attributi.
- Sintassi alternativa con `{}`: `Date today{23,6,1983};`

```
class Data {
    int d, m, y;
public:
    Data(int giorno, int mese, int anno)
        : d{giorno}, m{mese}, y{anno} {}
};
```

Distruttori

- Nome: `~NomeClasse()`
- Usati per rilasciare risorse alla distruzione dell'oggetto.

```
class File {
    FILE* f;
public:
```

```
    ~File() {
        if (f) fclose(f);
    }
};
```

Copia e Move Semantics

Copy Semantics

- **Copia per membro** (default).
- Serve se vogliamo duplicare l'oggetto.
- Può essere ridefinita (copy constructor & assignment operator).

```
class Libro {
    std::string titolo;
public:
    Libro(const Libro& other) : titolo(other.titolo) {}
};
```

Move Semantics

- Utile per oggetti grandi: evita copie profonde.
- Usa un **rvalue reference** (`Type&&`) e `std::move()` .
- Dopo il move, i puntatori vanno messi a `nullptr` per sicurezza.

```
class Buffer {
    char* data;
public:
    Buffer(Buffer&& other)
        : data(other.data) {
        other.data = nullptr;
    }
};
```

Membri Static

- Collegati alla **classe**, non alle istanze.
- Utili per attributi/metodi condivisi tra tutte le istanze.

```
class Contatore {
    static int count;
public:
    static void incrementa() { ++count; }
    static int getCount() { return count; }
};

int Contatore::count = 0;
```

Ereditarietà e Polimorfismo

Inheritance

- Sintassi: `class Manager : public Employee { ... }`
- Costruzione: **base → derivata**

  Distruzione: **derivata → base**
- I costruttori **non** vengono ereditati automaticamente.

```
class Persona {
public:
    std::string nome;
};

class Studente : public Persona {
public:
    int matricola;
};
```

Virtual

- `virtual` abilita il **polimorfismo** dinamico.
- Un metodo `= 0` è **pure virtual** → rende la classe **astratta**.
- `final` impedisce ulteriori override.

```
class Animale {
public:
    virtual void verso() const { std::cout << "???\n"; }
};

class Cane : public Animale {
public:
    void verso() const override { std::cout << "Bau\n"; }
};
```

Accesso Ereditato

- `public`: mantiene visibilità originale.
- `protected`: pubblici/protetti diventano protetti.
- `private`: tutto diventa privato nella sottoclasse.

Template di Classe

Definizione

```
template <typename T>
class Box {
    T valore;
public:
    void set(T v) { valore = v; }
    T get() const { return valore; }
};
```

```
Box<int> numeri;
Box<std::string> parole;
```

- `typename` e `class` sono equivalenti.
- Supporta più parametri template.
- Metodi **inline** possono omettere la riga `template<>`.

Note Finali

- Ogni membro può accedere agli altri, indipendentemente dall'ordine.
- Le dichiarazioni vanno inserite negli header.
- Evitare `protected` per gli attributi: meglio usare `private`.