# Matrice3D - Giugno 2018


## Puleri Gianluca - 807064 - g.puleri@campus.unimib.it


### Compilazione

- `make doc` per creare la documentazione con Doxygen
- `make` per compilare il progetto C++
- `make run_valgrind` per compilare il progetto C++ con stampe di debug e flag `-g` ed eseguire valgrind con la flag `--leak-check=yes`
- vi sono altre opzioni utilizzate in fase di scrittura del codice e di testing

### Introduzione

Il progetto richiede la progettazione e realizzazione di una classe generica Matrice3D che implementa una matrice a 3 dimensioni di celle. Ogni cella può essere acceduta dando le coordinate del piano (z), riga (y) e colonna (x).

### Scelte implementative

#### Implementazione della Matrice3D

Di seguito, le mie scelte progettuali per quanto riguarda il cuore della classe.

- **struct cell**:
  - la matrice si appoggia ad una `struct cell` molto semplice. Questa consiste in:
    - valore templato `T value`
    - valore bool `filled`
    - costruttore di default
    - costruttore dato `value`
- **campi della classe**:
  - ho scelto di utilizzare 5 campi:
    - `_matrix3D`, il puntatore al puntatore al puntatore di cell
    - `_z`, il numero di piani
    - `_y`, il numero di righe
    - `_x`, il numero di colonne
    - `_size`, la dimensione della matrice
  - la scelta è dovuta al fatto che questi campi sono sufficienti a svolgere tutte le operazioni necessarie
- Ho scelto di implementare due modi differenti per la conversione della matrice:
  - tramite costruttore template
  - tramite operator= template Questa scelta è stata fatta per rendere possibile sia la creazione di un nuovo oggetto T a partire da uno U, sia per convertire semplicemente la matrice stessa da U a T.

## Implementazione di metodi, operatori, iteratori

1. **costruttori**
   - sono stati implementati vari costruttori:
     1. costruttore di default, `Matrice3D()`
        - questo costruttore non è particolarmente utile, in quanto viene inizializzata una Matrice3D di dimensione 0. L'unico motivo per utilizzarlo è per istanziare una Matrice3D vuota ed utilizzarla per copiarci dentro un altra Matrice3D tramite `operator=`
     2. costruttore secondario dati z,y,x, `Matrice3D(int z, int y, int x)`
     3. costruttore secondario dati z,y,x e value, `Matrice3D(int z, int y, int x, const T &value)`
     4. costruttore copia, `Matrice3D(const Matrice3D &other)`
     5. costruttore template, `template <typename U> Matrice3D(const Matrice3D<U> &other)`.
        - Esso è utilizzato per la conversione della matrice in un altro tipo
2. **iteratori**
   - sono stati implementati sia `const_iterator` che `iterator`. Questo per consentire sia un accesso read/write, che un accesso in sola lettura, quando non necessaria la scrittura.
   - è stato scelto un `random access iterator`.
3. **operator()**
   - tramite questo operatore è possibile leggere/scrivere il valore di una determinata cella specificandone le coordinate.
4. **is_filled**
   - è un metodo tramite il quale è possibile sapere se una determinata cella è avvalorata o no.
5. **slice**
   - è un metodo che, dato il valore del piano z, ritorna la Matrice2D contenente tutte le celle del piano richiesto.
6. **get_size**, **get_z**, **get_y**, **get_x**
   - danno, rispettivamente: dimensione, numero di piani, di righe e di colonne.
7. **trasforma**
   - è una funzione generica che, data una Matrice3D A (su tipi T) e un funtore F, ritorna una nuova Matrice3D B (su tipi Q) i cui elementi sono ottenuti applicando il funtore agli elementi di A.

## Test

Sono stati creati dei test (con stampa a video) sia su tipi primitivi che per tipi custom tramite le seguenti funzioni:

- `test 2D primitivi`
- `test 2D custom`
- `test 3D primitivi`
- `test 3D custom`

## Informazioni varie

- in fase di compilazione e testing, sono state utilizzate le flag `-Wall -Wextra -pedantic` di `g++` ai fini di porre particolare attenzione alla conformità agli standard e di non dimenticare nulla.