LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ Université Paris-Saclay

Projet de programmation C++

CDataframe

Halim Djerroud

Février 2025

Table des matières

1. Préambule

Les logiciels tableurs tels que "LibreOffice Calc" ou "MS Excel", sont des outils puissants pour manipuler des données, les trier, visualiser des graphiques, calculer des sommes et des moyennes, et bien plus encore. Cependant, lorsque vous recevez quotidiennement des milliers de données sous forme de fichiers structurés (CSV par exemple), et que vous devez effectuer tous ces traitements de façon répétitive, l'utilisation de logiciels tableurs devient fastidieuse. En effet, leur fonctionnement naturel impose l'ouverture de chaque fichier manuellement et la répétition des mêmes actions plusieurs fois, ce qui peut être chronophage et source d'erreurs.

Une des solutions existantes pour l'automatisation des tâches dans les tableurs est la programmation de Macros. Certains logiciels tableurs comme "Calc" permettent d'enregistrer ces macros pour automatiser des séquences d'actions effectuées sur des tâches simples et répétitives. Néanmoins, leur utilisation peut vite devenir complexe et difficile à gérer pour des traitements plus élaborés.

L'autre solution consiste à utiliser directement des scripts puissants et flexibles en passant par un langage de programmation tels que Python. En effet, grâce à sa librairie **Pandas**, il est possible de réaliser un large éventail de fonctions pour importer, nettoyer, analyser et visualiser des données. Cette souplesse passe par l'utilisation d'une structure de données, propre à Pandas, appelée **"DataFrame"** qui se gère de façon similaire à une feuille de calcul se trouvant dans un tableur.

Toutefois, une telle librairie n'est pas disponible nativement en C++, c'est pourquoi nous souhaitons dans ce projet proposer une alternative en développant une librairie écrite en langage C++ et qui permet de réaliser quelques unes des fonctionnalités existantes sur **Pandas**.

2. Introduction

L'objectif de ce projet est de créer un ensemble de classes en langage C++ (appelées communément une librairie) qui permettent de faciliter la manipulation de données.

Pour ce faire, il est important de comprendre le fonctionnement d'une feuille de calcul dans un tableur ou encore comprendre le fonctionnement d'un **"DataFrame"** dans **Pandas**.

En effet, cette structure est composée de cellules organisées sous forme d'un tableau 2D (matrice). Toutefois, une utilisation désordonnée de ces cellules peut vite devenir un casse-tête. Il est donc important que l'utilisateur organise lui-même les cellules afin de correspondre à une abstraction du problème qu'il souhaite résoudre. Il existe probablement plusieurs façons d'organiser des données et les tableurs peuvent être utilisés pour d'autres fins telles que la création des emplois du temps ou l'élaboration de menus hebdomadaires, etc. mais dans ce projet nous souhaitons adopter l'idée de les organiser comme étant un ensemble de colonnes qui offrent des fonctionnalités similaires à celles qu'offre le **"DataFrame"** dans **Pandas**.

3. Description du projet

Dans ce projet, nous allons implémenter une structure composée d'un ensemble de colonnes, appelée : **CDataframe**. Chaque colonne a un titre et permet de stocker un nombre indéfini de données de même type. Toutes les colonnes doivent avoir le même nombre de données afin de former une matrice. Si des données sont manquantes alors elles sont remplacées par des valeurs par défaut que l'on définira plus tard.

Une fois la structure créée, nous souhaitons pouvoir offrir à l'utilisateur la possibilité d'effectuer au minimum l'ensemble des fonctionnalités suivantes :

1. Alimentation
   - Création d'un CDataframe vide
   - Remplissage du CDataframe à partir de saisies utilisateurs
   - Remplissage en dur du CDataframe
2. Affichage
   - Afficher tout le CDataframe
   - Afficher une partie des lignes du CDataframe selon une limite fournie par l'utilisateur
   - Afficher une partie des colonnes du CDataframe selon une limite fournie par l'utilisateur
3. Opérations usuelles
   - Ajouter une ligne de valeurs au CDataframe
   - Supprimer une ligne de valeurs du CDataframe
   - Ajouter une colonne au CDataframe
   - Supprimer une colonne du CDataframe
   - Renommer le titre d'une colonne du CDataframe
   - Vérifier l'existence d'une valeur (recherche) dans le CDataframe
   - Accéder/remplacer la valeur se trouvant dans une cellule du CDataframe en utilisant son numéro de ligne et de colonne
4. Analyse et statistiques
   - Afficher le nombre de lignes
   - Afficher le nombre de colonnes
   - Nombre de cellules égales à x (x donné en paramètre)
   - Nombre de cellules contenant une valeur supérieure à x (x donné en paramètre)
   - Nombre de cellules contenant une valeur inférieure à x(x donné en paramètre)

Afin de simplifier la réalisation de ce projet, nous allons procéder par étapes en décomposant le travail en 3 parties :

- **Partie 1 :** La structure ne contiendra que des données de type "entier", organisées en colonnes où chacune des colonnes va avoir un titre.
  Sur cette structure, nous devons pouvoir appliquer au minimum toutes les fonctionnalités décrites ci-dessus.

- **Partie 2 :** Dans cette partie, il est demandé d'étendre le travail précédent sur deux volets :
  - Développer de nouvelles fonctionnalités en plus des opérations de base réalisées dans la partie 1
  - Permettre d'organiser des données de types différents dans un même CDataframe où chaque colonne doit avoir des données de même type, mais que deux colonnes différentes peuvent avoir des données de types différents tel qu'illustré dans la figure 1 ci-dessous :

Titre	Colonne 1	Colonne 2		Colonne n
	Place	Code	....	Indice-p
	52	Lima		1.158
	44	Bravo		6.135
	15	Zulu		NULL
	18	Tango		NULL

FIGURE 1 – Exemple d'un tableau de données.

— **Partie 3 :** Fichier CSV, conteneurs STL et fonctionnalités avancées

Première partie
Un CDataframe d'entiers

4. CDataframe

Nous souhaitons stocker des données en colonnes, chaque colonne dispose d'un en-tête et des données. Toutes les données stockées dans la colonne sont du même type.

Pour stocker nos données il nous faut une structure qui va être à la fois bien solide pour permettre de contenir facilement un grand nombre de données, et flexible afin de permettre d'ajouter, supprimer, déplacer les colonnes ou les lignes.

En C++, nous allons utiliser une classe qui encapsule les données et les méthodes associées. La structure globale ressemble à un tableau 2D, mais l'ajout de titres aux colonnes et la gestion dynamique des données rendent l'utilisation d'une classe plus appropriée.

Il est donc nécessaire de penser à une classe pour une colonne qui lui permettra d'être représentée par un titre et un conteneur de données (nous utiliserons `std::vector` pour la gestion dynamique). Pour qu'enfin, le CDataframe ne soit autre qu'un conteneur de colonnes comme illustré sur la Figure 2.

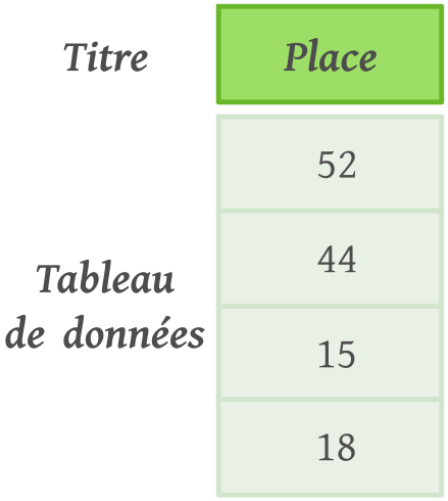

FIGURE 2 – Description d'une colonne.

4.1. Les colonnes

Une classe `Column` contient le titre de la colonne qui est une chaîne de caractères et des données de type entier dans cette partie. Pour un accès rapide aux données (accès direct) nous allons utiliser `std::vector<int>`.

En C++, `std::vector` gère automatiquement la mémoire et l'allocation dynamique. Le vecteur s'agrandit automatiquement lorsque nécessaire, mais nous pouvons utiliser la méthode `reserve()` pour pré-allouer de l'espace et éviter des réallocations fréquentes.

```
    const size_t REALLOC_SIZE = 256;
```

Cette classe `Column` est donc le premier élément à définir dans votre projet.

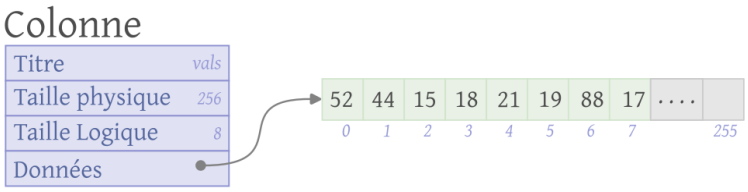

FIGURE 3 – Colonne d'un CDataframe.

4.1.1. Créer une colonne

Le constructeur de la classe permet de créer une colonne vide à partir d'un titre. Il initialise les attributs de la colonne.

Définition de la classe :

```
/**
 * @brief Column class for storing integer values
 */
class Column {
private:
    std::string title;
    std::vector<int> data;

public:
    /**
     * @brief Constructor - Create a column
     * @param columnTitle : Column title
     */
    Column(const std::string& columnTitle);

    // Autres méthodes à définir...
};
```

Exemple d'utilisation :

```
Column myCol("My column");
```

4.1.2. Insérer une valeur dans une colonne

Cette méthode permet d'insérer une valeur dans une colonne. Le vecteur gère automatiquement la mémoire, donc pas besoin de vérifier l'espace disponible manuellement.

Déclaration de la méthode :

```
/**
 * @brief : Add a new value to a column
 * @param value : The value to be added
 * @return : true if the value is added, false otherwise
 */
bool insertValue(int value);
```

Exemple d'utilisation :

```
Column myCol("My column");
int val = 5;
if (myCol.insertValue(val))
    std::cout << "Value added successfully to my column" << std::endl;
else
    std::cout << "Error adding value to my column" << std::endl;
```

4.1.3. Destructeur

En C++, le destructeur est appelé automatiquement lorsque l'objet est détruit. Avec `std::vector` et `std::string`, la gestion de la mémoire est automatique.

Déclaration :

```
/**
 * @brief : Destructor - Free allocated memory
 */
~Column();
```

Note : Avec les conteneurs STL, le destructeur peut souvent être laissé par défaut car la libération est automatique.

4.1.4. Afficher le contenu d'une colonne

La méthode suivante doit permettre d'afficher le contenu d'une colonne. Pour chaque ligne elle doit aussi afficher le numéro de la ligne (indice) suivi de la valeur contenue.

Déclaration de la méthode :

```
/**
 * @brief: Print a column content
 */
void print() const;
```

Exemple d'utilisation :

```
Column myCol("My column");
myCol.insertValue(52);
myCol.insertValue(44);
myCol.insertValue(15);
myCol.print();
```

Sortie :

```
[0] 52
[1] 44
[2] 15
```

4.1.5. Autres méthodes

En plus des méthodes précédentes, il faudra implémenter l'ensemble des méthodes qui permettent la réalisation des opérations suivantes :

— Retourner le nombre d'occurrences d'une valeur x (x donné en paramètre).
— Retourner la valeur présente à la position x (x donné en paramètre).

— Retourner le nombre de valeurs qui sont supérieures à x (x donné en paramètre).
— Retourner le nombre de valeurs qui sont inférieures à x (x donné en paramètre).
— Retourner le nombre de valeurs qui sont égales à x (x donné en paramètre).

Il est à noter que d'autres méthodes utiles seront potentiellement à ajouter lorsque le besoin se présente dans la suite de ce projet.

4.2. Le CDataframe

Jusqu'à présent nous n'avons pas implémenté de classe qui permet de créer un CDataframe, car ce dernier est composé de colonnes. Maintenant que nous avons nos briques de base, il est possible de concevoir un CDataframe qui n'est autre qu'un conteneur de colonnes comme illustré sur la Figure 4.

En C++, nous utiliserons `std::vector<Column>` ou `std::vector<std::shared_ptr<Column>>` pour stocker les colonnes.

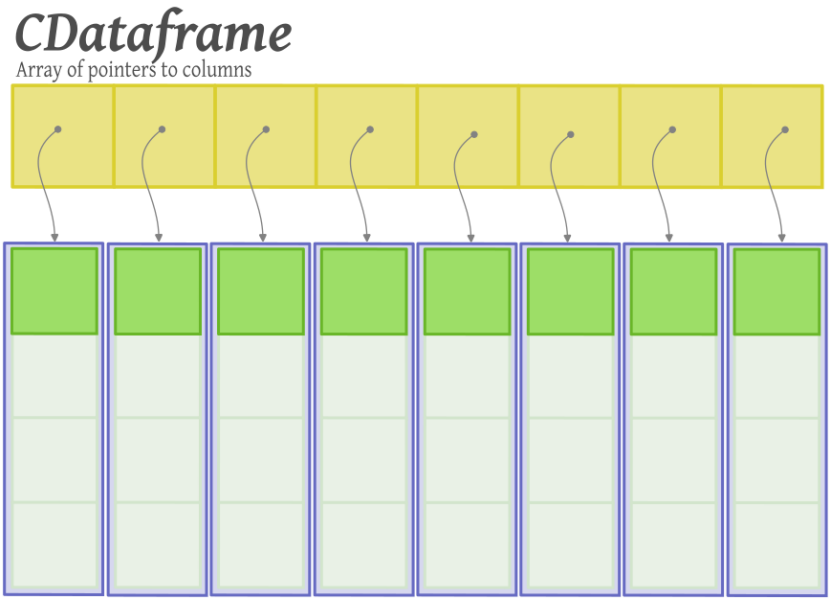

FIGURE 4 – Structure d'un CDataframe.

4.2.1. Fonctionnalités

Pour pouvoir utiliser le CDataframe, il faut implémenter un ensemble de méthodes qui se traduiront comme fonctionnalités que l'utilisateur pourra choisir au travers d'un menu dans son programme principal.

Les fonctionnalités de base que doit assurer votre CDataframe sont celles listées dans la description du projet, à savoir :

1. Alimentation
   — Création d'un CDataframe vide
   — Remplissage du CDataframe à partir de saisies utilisateurs
   — Remplissage en dur du CDataframe
2. Affichage
   — Afficher tout le CDataframe

   - Afficher une partie des lignes du CDataframe selon une limite fournie par l'utilisateur
   - Afficher une partie des colonnes du CDataframe selon une limite fournie par l'utilisateur
3. Opérations usuelles
   - Ajouter une ligne de valeurs au CDataframe
   - Supprimer une ligne de valeurs du CDataframe
   - Ajouter une colonne au CDataframe
   - Supprimer une colonne du CDataframe
   - Renommer le titre d'une colonne du CDataframe
   - Vérifier l'existence d'une valeur (recherche) dans le CDataframe
   - Accéder/remplacer la valeur se trouvant dans une cellule du CDataframe en utilisant son numéro de ligne et de colonne
4. Analyse et statistiques
   - Afficher le nombre de lignes
   - Afficher le nombre de colonnes
   - Nombre de cellules contenant une valeur égale à x (x donné en paramètre)
   - Nombre de cellules contenant une valeur supérieure à x (x donné en paramètre)
   - Nombre de cellules contenant une valeur inférieure à x(x donné en paramètre)

Deuxième partie
Un CDataframe presque parfait

Dans cette deuxième partie, l'idée est d'améliorer le CDataframe précédent, et ce, en agissant sur deux volets :

— Améliorer la structure de la colonne et du CDataframe pour offrir une utilisation plus large de celui-ci.
— Ajouter des fonctionnalités avancées

5. De nouvelles structures

Afin de permettre au CDataframe de stocker des données de types différents, il est nécessaire de modifier la structure de la colonne. En C++, nous allons utiliser des templates et/ou `std::variant` pour gérer les types multiples. Il s'agira toujours d'un ensemble de colonnes. Seulement, nous souhaitons que les données d'une même colonne soient de même type mais que celles de deux colonnes différentes soient de deux types différents.

5.0.1. La colonne

La figure 5 ci-après montre la structure d'une colonne.

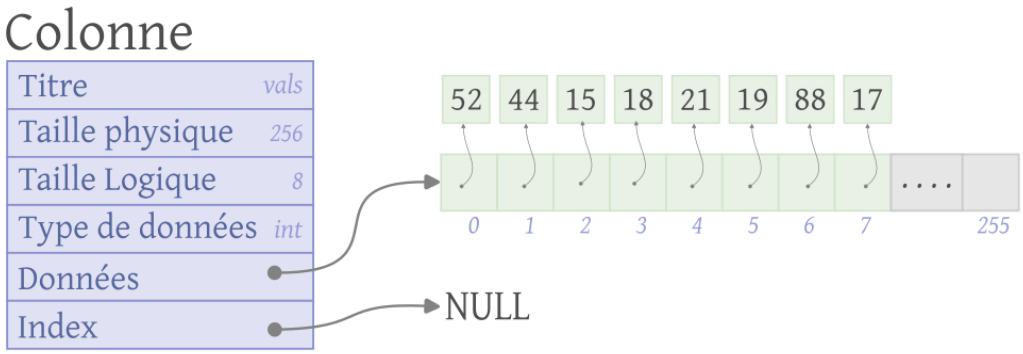

FIGURE 5 – Colonne.

Une colonne doit stocker des données de différents types. En C++, nous utiliserons `std::variant` ou des templates pour gérer cela. Voici les types possibles :

— entiers naturels : `uint32_t`
— entiers relatifs : `int32_t`
— petits entiers naturels : `uint16_t`
— petits relatifs : `int16_t`
— grands entiers naturels : `uint64_t`
— grands relatifs : `int64_t`
— très petits entiers naturels : `uint8_t`
— très petits relatifs : `int8_t`

— flottants simple précision : `float`
— flottants double précision : `double`
— Chaînes de caractères : `std::string`
— Objet quelconque : `std::any` ou templates

En C++, nous pouvons utiliser une énumération pour identifier le type :

```
enum class ColumnType {
    NULLVAL = 1,
    UINT,
    INT,
    USHORT,
    SHORT,
    ULONG,
    LONG,
    UCHAR,
    CHAR,
    FLOAT,
    DOUBLE,
    STRING,
    OBJECT
};
```

Pour stocker les données, nous utiliserons `std::variant` ou une classe template :

```
using ColumnValue = std::variant<
    std::monostate,      // Pour NULL
    uint32_t,
    int32_t,
    uint16_t,
    int16_t,
    uint64_t,
    int64_t,
    uint8_t,
    int8_t,
    float,
    double,
    std::string,
    std::any             // Pour les objets quelconques
>;
```

La classe Column peut alors être définie comme suit :

```
class Column {
private:
    std::string columnName;
    size_t size;
    ColumnType columnType;
    std::vector<ColumnValue> data;
    std::vector<size_t> index;
    bool validIndex;
    bool sortAscending;
```

```
public:
    /**
     * @brief Constructor - Create a new column
     * @param type : column type
     */
    Column(ColumnType type);

    // Autres méthodes...
};
```

5.1. Créer une colonne

Le constructeur permet de créer une colonne d'un type donné. Il initialise tous les attributs de la colonne.

Déclaration du constructeur :

```
/**
 * @brief Create a new column
 * @param type : column type
 */
Column(ColumnType type);
```

Exemple d'utilisation :

```
 Column myCol(ColumnType::CHAR);
```

5.2. Insérer une valeur dans une colonne

La méthode suivante permet d'insérer une valeur dans une colonne. Cette méthode est générique grâce à `std::variant`.

Déclaration de la méthode :

```
/**
 * @brief: Insert a new value into a column
 * @param value : The value to insert (as ColumnValue)
 * @return: true if the value is correctly inserted, false otherwise
 */
bool insertValue(const ColumnValue& value);

// Ou version template pour plus de flexibilité
template<typename T>
bool insertValue(const T& value);
```

Exemple d'utilisation :

```
Column myCol(ColumnType::CHAR);
char a = 'A', c = 'C';
myCol.insertValue(a);
myCol.insertValue(std::monostate{}); // Pour NULL
myCol.insertValue(c);
```

Avec `std::vector`, la gestion de la mémoire est automatique, mais vous pouvez utiliser `reserve()` pour optimiser :

```cpp
bool Column::insertValue(const ColumnValue& value) {
    // Vérification du type si nécessaire
    data.push_back(value);
    size++;
    return true;
}
```

5.3. Destructeur

Le destructeur est géré automatiquement par les conteneurs STL.

Déclaration :

```cpp
/**
 * @brief: Destructor - free the space allocated by a column
 */
~Column() = default; // Peut être par défaut avec STL
```

5.4. Afficher une valeur

La méthode suivante permet de récupérer une valeur d'une colonne à une position donnée, dans une chaîne de caractères.

```cpp
/**
 * @brief: Convert a column value to string
 * @param i: The index of the value to retrieve
 * @return: String representation of the value
 */
std::string valueToString(size_t i) const;
```

Par exemple dans la colonne illustrée dans l'exemple suivant :

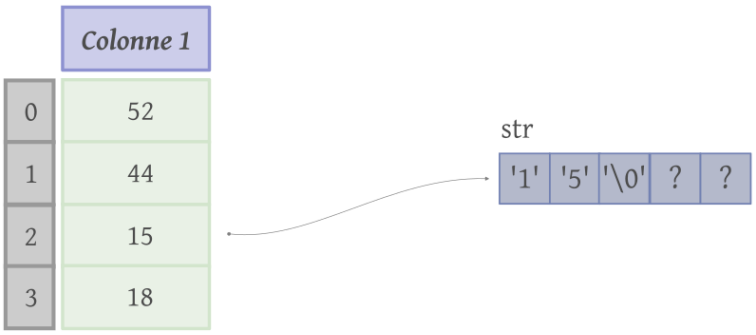

FIGURE 6 – Convertir le contenu de la ligne 2 dans une chaîne de caractère.

Exemple d'utilisation :

```cpp
Column myCol(ColumnType::INT);
int a = 52, b = 44, c = 15, d = 18;
myCol.insertValue(a);
myCol.insertValue(b);
myCol.insertValue(c);
myCol.insertValue(d);
```

```
std::string str = myCol.valueToString(2);
std::cout << str << std::endl;
```

Sortie du programme :

```
15
```

Astuce : En C++, vous pouvez utiliser `std::ostringstream` ou `std::to_string`, et avec `std::variant`, utilisez `std::visit`.

Exemple d'implémentation :

```
std::string Column::valueToString(size_t i) const {
    return std::visit([](auto&& arg) -> std::string {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, std::monostate>) {
            return "NULL";
        } else if constexpr (std::is_same_v<T, std::string>) {
            return arg;
        } else {
            return std::to_string(arg);
        }
    }, data[i]);
}
```

5.5. Afficher le contenu d'une colonne

La méthode suivante doit permettre d'afficher le contenu d'une colonne. Pour chaque ligne elle doit aussi afficher le numéro de ligne suivi de la valeur du contenu. Si la valeur est nulle (`NULL`) alors on affiche la chaîne de caractères `NULL`.

Déclaration de la méthode :

```
/**
 * @brief: Display the contents of a column
 */
void print() const;
```

Exemple d'utilisation :

```
Column myCol(ColumnType::CHAR);
char a = 'A', c = 'C';
myCol.insertValue(a);
myCol.insertValue(std::monostate{});
myCol.insertValue(c);
myCol.print();
```

Sortie :

```
[0] A
[1] NULL
[2] C
```

5.6. Informations sur une colonne

La méthode `info` permet d'afficher les informations essentielles sur une colonne. Cela inclut son type, le nombre de valeurs insérées, ainsi que le nombre de valeurs nulles.

Déclaration de la méthode :

```
    /**
     * @brief: Display the information of a column
     */
    void info() const;
```

Exemple d'utilisation :

```
    Column col(ColumnType::INT);
    int x = 10, y = 20;
    col.insertValue(x);
    col.insertValue(std::monostate{});
    col.insertValue(y);
    col.info();
```

Sortie :

```
Type       : INT
Taille     : 3
Nulls      : 1
```

6. Trier une colonne

Trier les valeurs des colonnes permet d'afficher les valeurs dans un certain ordre (croissant ou décroissant) et permet aussi de vérifier l'existence (recherche) d'une valeur dans une colonne très rapidement en utilisant la technique de recherche dichotomique.

En C++, nous utiliserons un vecteur d'indices pour éviter de modifier l'ordre réel des données. Cela est particulièrement utile dans un dataframe où plusieurs colonnes doivent rester synchronisées.

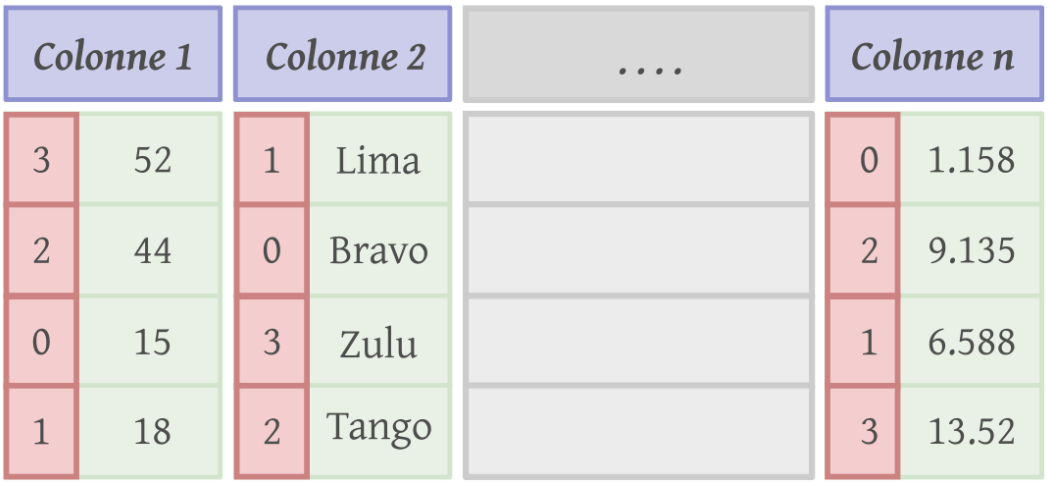

FIGURE 7 – Colonne avec index.

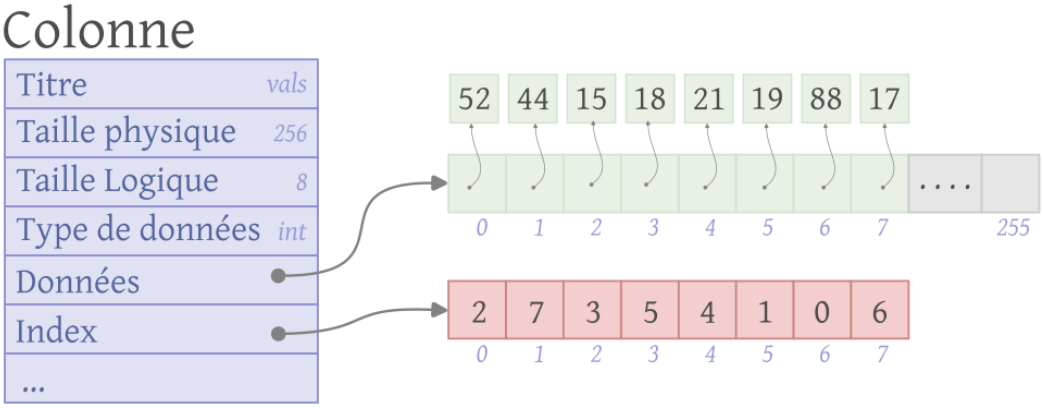

FIGURE 8 – Détails de la colonne avec index.

La classe Column est enrichie avec :

```
class Column {
private:
    ...
    std::vector<size_t> index;
    bool validIndex;
    bool sortAscending;

public:
    ...
```

```
};
```

6.1. Trier une colonne

La méthode suivante permet de trier une colonne selon un ordre croissant ou décroissant.

```
/**
 * @brief: Sort a column according to a given order
 * @param ascending : true for ascending, false for descending
 */
void sort(bool ascending = true);
```

En C++, nous pouvons utiliser `std::sort` avec une fonction de comparaison personnalisée :

```
void Column::sort(bool ascending) {
    // Initialiser l'index si nécessaire
    if (index.empty()) {
        index.resize(size);
        std::iota(index.begin(), index.end(), 0);
    }

    // Trier l'index
    std::sort(index.begin(), index.end(),
        [this, ascending](size_t a, size_t b) {
            if (ascending) {
                return compareValues(data[a], data[b]) < 0;
            } else {
                return compareValues(data[a], data[b]) > 0;
            }
        });

    validIndex = true;
    sortAscending = ascending;
}
```

6.1.1. Algorithmes de tri

En C++, la bibliothèque standard offre des algorithmes de tri très efficaces :

- `std::sort` : tri rapide optimisé (généralement introsort)
- `std::stable_sort` : tri stable qui préserve l'ordre relatif
- Pour un tri partiel : `std::partial_sort`

Pour l'insertion d'une nouvelle valeur dans une colonne triée, vous pouvez utiliser :

```
// Après insertion d'une valeur
if (validIndex) {
    validIndex = false; // Index invalide, nécessite mise à jour
}
```

6.2. Afficher le contenu d'une colonne triée

Cette méthode permet d'afficher les valeurs d'une colonne dans un ordre trié.

Déclaration de la méthode :

```
/**
 * @brief: Display the contents of a column in sorted order
 * @param ascending: true for ascending, false for descending
 */
void printSorted(bool ascending = true);
```

Exemple d'utilisation :

```
Column col(ColumnType::INT);
int a = 5, b = 2, c = 8;
col.insertValue(a);
col.insertValue(b);
col.insertValue(c);
col.printSorted(true);   // affichage croissant
```

Sortie :

```
[1] 2
[0] 5
[2] 8
```

6.3. Effacer l'index d'une colonne

La méthode suivante permet de supprimer un index.

Déclaration de la méthode :

```
/**
 * @brief: Remove the index of a column
 */
void eraseIndex();
```

Implémentation :

```
void Column::eraseIndex() {
    index.clear();
    validIndex = false;
}
```

6.4. Vérifier si une colonne dispose d'un index

Cette méthode permet de vérifier l'état de l'index d'une colonne.

Déclaration de la méthode :

```
/**
 * @brief: Check if an index is correct
 * @return: -1: index not existing,
 *           0: the index exists but invalid,
 *           1: the index is correct
 */
```

```
int checkIndex() const;
```

6.5. Mettre à jour un index

Cette méthode permet de mettre à jour un index.

Déclaration de la méthode :

```
/**
 * @brief: Update the index
 */
void updateIndex();
```

6.6. Recherche dichotomique

En utilisant la recherche dichotomique il faut vérifier si une valeur donnée en paramètre existe bien dans une colonne.

Déclaration de la méthode :

```
/**
 * @brief: Test if a value exists in a column
 * @param val: The value to search for
 * @return: -1: column not sorted,
 *           0: value not found
 *           1: value found
 */
template<typename T>
int searchValue(const T& val) const;
```

En C++, vous pouvez utiliser `std::binary_search` ou `std::lower_bound` :

```
template<typename T>
int Column::searchValue(const T& val) const {
    if (!validIndex) return -1;

    // Utiliser std::binary_search ou implémentation manuelle
    auto it = std::lower_bound(index.begin(), index.end(), val,
        [this](size_t idx, const T& value) {
            return compareToValue(data[idx], value) < 0;
        });

    if (it != index.end() && compareToValue(data[*it], val) == 0) {
        return 1; // Trouvé
    }
    return 0; // Non trouvé
}
```

7. Bonus

7.1. Appliquer une fonction à une colonne

En C++, nous pouvons utiliser des templates et des lambdas pour appliquer des fonctions.

```
/**
 * @brief : Apply a function to all elements of a column
 * @param func : Function to apply (lambda or function pointer)
 * @return : Result of the operation
 */
template<typename Func, typename ResultType>
ResultType applyFunction(Func func) const;
```

Exemple d'utilisation avec lambda :

```
Column col(ColumnType::INT);
// ... remplir la colonne ...

// Calculer la somme
int sum = col.applyFunction<int>([](int acc, int val) {
    return acc + val;
});

// Calculer la moyenne
double mean = static_cast<double>(sum) / col.getSize();
```

7.2. Appliquer une fonction à plusieurs colonnes

Cette méthode permet d'appliquer une fonction sur deux colonnes et créer une nouvelle colonne.

```
/**
 * @brief : Apply a function on 2 columns to create a 3rd
 * @param col1 : First column
 * @param col2 : Second column
 * @param func : Function to apply
 * @return : Pointer to the new column created
 */
template<typename Func>
static std::unique_ptr<Column> applyFunction(
    const Column& col1,
    const Column& col2,
    ColumnType resultType,
    Func func
);
```

8. Implémentation d'un dataframe

Jusqu'à présent nous n'avons pas implémenté de classe qui permet de créer un dataframe car, ce dernier est composé de colonnes. Maintenant que nous avons nos briques de base, nous proposons d'implémenter les méthodes suivantes :

En C++, nous utiliserons `std::vector` ou `std::list` pour stocker les colonnes. Le choix dépend des opérations les plus fréquentes :

— `std::vector<Column>` : accès rapide par indice, bon pour l'affichage
— `std::vector<std::shared_ptr<Column>>` : partage de colonnes, flexibilité
— `std::list<Column>` : insertions/suppressions fréquentes en milieu de liste

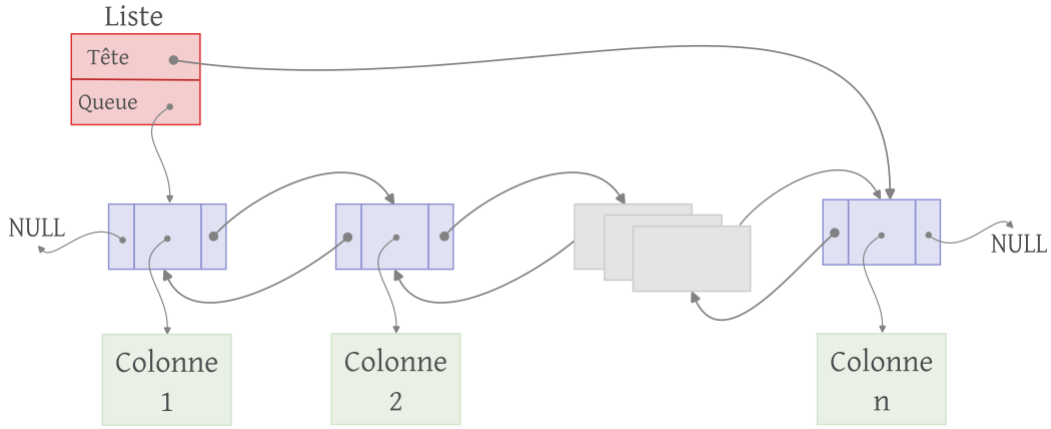

FIGURE 9 – Organisation en conteneur.

La classe CDataframe peut être définie comme suit :

```cpp
/**
 * @brief CDataframe class
 */
class CDataframe {
private:
    std::vector<std::shared_ptr<Column>> columns;
    // Ou : std::list<Column> columns;

public:
    /**
     * @brief Constructor - Create a dataframe
     * @param types : Vector of column types
     */
    CDataframe(const std::vector<ColumnType>& types);

    // Autres méthodes...
};
```

Conseil *Nous vous proposons de séparer votre code en plusieurs fichiers.*

— **CDataframe.h/.cpp :** Classe principale du dataframe avec toutes les opérations publiques

— **Column.h/.cpp :** Classe de gestion des colonnes
— **ColumnValue.h :** Définition du type variant et énumérations
— **Utils.h/.cpp :** Fonctions utilitaires (tri, recherche, etc.)

Une méthode qui permet de créer un *dataframe* vide.

Déclaration du constructeur :

```
/**
 * @brief Create a dataframe
 * @param types : Vector of column types
 */
CDataframe(const std::vector<ColumnType>& types);
```

Exemple d'utilisation :

```
std::vector<ColumnType> types = {
    ColumnType::INT,
    ColumnType::CHAR,
    ColumnType::INT
};
CDataframe df(types);
```

Déclaration de la méthode :

```
/**
 * @brief Delete a column by name
 * @param colName : Column name
 */
void deleteColumn(const std::string& colName);
```

Déclaration de la méthode :

```
/**
 * @brief Set column names
 * @param names : Vector of column names
 */
void setColumnNames(const std::vector<std::string>& names);
```

Déclaration de la méthode :

```
/**
 * @brief Get the number of columns
 * @return : Number of columns
 */
size_t getColumnsCount() const;
```

Déclaration de la méthode :

```
/**
 * @brief Insert a new row
 * @param values : Vector of values to insert
 * @return : true if successful
 */
bool insertRow(const std::vector<ColumnValue>& values);
```

Déclaration de la méthode :

```
    /**
     * @brief Get the number of rows
     * @return : Number of rows
     */
    size_t getRowsCount() const;
```

Déclaration de la méthode :

```
    /**
     * @brief Display dataframe information
     */
    void info() const;
```

La méthode suivante va permettre d'afficher l'en-tête du *dataframe*, c'est-à-dire afficher uniquement le nom des colonnes.

Déclaration de la méthode :

```
    /**
     * @brief Display the dataframe header
     */
    void printHeader() const;
```

La méthode suivante permet d'afficher le contenu d'un *dataframe*. Plus exactement elle va permettre d'afficher des lignes consécutives. Les indices de début et de fin sont donnés en paramètres.

Déclaration de la méthode :

```
    /**
     * @brief: Display a portion of the dataframe line by line
     * @param first: index of the first line to display (inclusive)
     * @param last: index of the last line to display (exclusive)
     */
    void printByLine(size_t first, size_t last) const;
```

Les trois méthodes suivantes sont des variantes de la méthode précédente :

Déclaration des méthodes :

```
    /**
     * @brief Display the entire dataframe
     */
    void print() const;

    /**
     * @brief Display the first 10 rows of the dataframe
     */
    void printHead() const;

    /**
     * @brief Display the last 10 rows of the dataframe
     */
    void printTail() const;
```

Déclaration de la méthode :

```
    /**
```

```
     * @brief Load from a CSV file
     * @param filename : CSV file path
     * @param types : Vector of column types
     * @return : Unique pointer to the loaded dataframe
     */
    static std::unique_ptr<CDataframe> loadFromCSV(
        const std::string& filename,
        const std::vector<ColumnType>& types
    );
```

En C++, vous pouvez utiliser `std::ifstream` et `std::getline` pour lire le fichier :

```
    std::unique_ptr<CDataframe> CDataframe::loadFromCSV(
        const std::string& filename,
        const std::vector<ColumnType>& types) {

        std::ifstream file(filename);
        if (!file.is_open()) {
            throw std::runtime_error("Cannot open file: " + filename);
        }

        auto df = std::make_unique<CDataframe>(types);

        std::string line;
        // Lire l'en-tête
        if (std::getline(file, line)) {
            std::vector<std::string> headers = parseLine(line);
            df->setColumnNames(headers);
        }

        // Lire les données
        while (std::getline(file, line)) {
            std::vector<ColumnValue> values = parseDataLine(line, types);
            df->insertRow(values);
        }

        return df;
    }
```

Déclaration de la méthode :

```
    /**
     * @brief Automatic loading from CSV file
     * @param filename : CSV file path
     * @return : Unique pointer to the loaded dataframe
     */
    static std::unique_ptr<CDataframe> loadFromCSVAuto(
        const std::string& filename
    );
```

Cette méthode permet d'exporter un *dataframe* sous forme d'un fichier CSV.

```
    /**
```

```
     * @brief Export to a CSV file
     * @param filename : Output file path
     */
    void saveToCSV(const std::string& filename) const;
```

Exemple d'implémentation :

```
    void CDataframe::saveToCSV(const std::string& filename) const {
        std::ofstream file(filename);
        if (!file.is_open()) {
            throw std::runtime_error("Cannot create file: " + filename);
        }

        // Écrire l'en-tête
        printHeader(file);
        file << "\n";

        // Écrire les données
        for (size_t i = 0; i < getRowsCount(); ++i) {
            for (size_t j = 0; j < columns.size(); ++j) {
                file << columns[j]->valueToString(i);
                if (j < columns.size() - 1) file << ",";
            }
            file << "\n";
        }
    }
```

9. Conseils supplémentaires pour C++

9.1. Utilisation des conteneurs STL

Le C++ offre de nombreux conteneurs optimisés :

- `std::vector` : tableau dynamique, accès O(1), insertion en fin O(1) amorti
- `std::list` : liste doublement chaînée, insertion/suppression O(1)
- `std::deque` : double-ended queue, accès O(1), insertion aux deux bouts O(1)
- `std::map` / `std::unordered_map` : pour indexer par nom de colonne

9.2. Gestion de la mémoire

Utilisez les smart pointers pour éviter les fuites mémoire :

- `std::unique_ptr` : propriété exclusive
- `std::shared_ptr` : propriété partagée avec comptage de références
- `std::weak_ptr` : référence faible sans comptage

9.3. Templates et généricité

Les templates permettent une grande flexibilité :

```
template<typename T>
class TypedColumn : public Column {
private:
    std::vector<T> data;
public:
    void insert(const T& value) { data.push_back(value); }
    T get(size_t index) const { return data[index]; }
};
```

9.4. Surcharge d'opérateurs

Le C++ permet de surcharger les opérateurs pour une syntaxe intuitive :

```
class CDataframe {
public:
    // Accès par []
    Column& operator[](size_t index) { return *columns[index]; }

    // Accès par nom de colonne
    Column& operator[](const std::string& name);

    // Opérateur de flux pour l'affichage
    friend std::ostream& operator<<(std::ostream& os,
                                    const CDataframe& df);
};
```

9.5. Exceptions

Utilisez les exceptions pour gérer les erreurs :

```
void CDataframe::deleteColumn(const std::string& name) {
    auto it = std::find_if(columns.begin(), columns.end(),
        [&name](const auto& col) { return col->getName() == name; });

    if (it == columns.end()) {
        throw std::out_of_range("Column not found: " + name);
    }

    columns.erase(it);
}
```

9.6. Algorithmes STL

Profitez des algorithmes de la bibliothèque standard :

```
#include <algorithm>
#include <numeric>

// Tri
std::sort(index.begin(), index.end(), comparator);

// Recherche
auto it = std::find(data.begin(), data.end(), value);

// Transformations
std::transform(data.begin(), data.end(), result.begin(), func);

// Accumulation (somme, produit, etc.)
int sum = std::accumulate(data.begin(), data.end(), 0);
```

9.7. Move semantics

Utilisez la sémantique de déplacement pour optimiser :

```
class Column {
public:
    // Constructeur par déplacement
    Column(Column&& other) noexcept
        : data(std::move(other.data))
        , index(std::move(other.index)) {}

    // Opérateur d'affectation par déplacement
    Column& operator=(Column&& other) noexcept {
        data = std::move(other.data);
        index = std::move(other.index);
        return *this;
    }
};
```