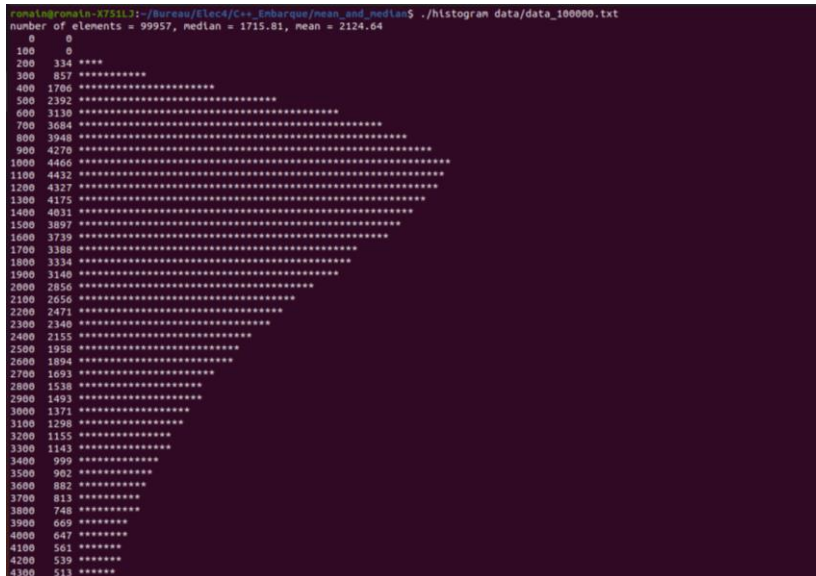


# TP 1 « Introduction »

## LES STRING ET LES VECTOR<>

Nous avons repris le code de *mean\_and\_median.cpp*, l'avons dans un fichier *histogram.cpp* et l'avons complété afin d'afficher un histogramme en étoiles sur la console.

On obtient ainsi le résultat suivant pour le fichier *data\_100000.txt*



On obtient bien 60 étoiles pour la valeur maximale et un nombre d'étoiles proportionnels pour les autres valeurs.

Pour compter le nombre de valeurs comprises dans chaque intervalle, on réalise deux boucles imbriquées. La première va passer d'intervalle en intervalle et la seconde va parcourir le buffer jusqu'à la première valeur qui sort de l'intervalle courant soit atteinte.

```
vector<double> val_in_bin; // Store number of values inside the bin intervals

auto b = buf.begin();
auto n = 0;
// for each intervals
for (int i = 0; i < I_MAX ; i+=STEP_SIZE) {
    // Count the number of values in interval
    while (*b < i+STEP_SIZE && b != buf.end()) {n++; b++;}
    // Store the value
    val_in_bin.push_back(n);
    n = 0;
}
```

Pour l'affichage on utilise les valeurs récupérées dans le tableau précédent pour calculer un nombre d'étoiles entre 0 et 60 proportionnel à la valeur courante.

```
// Number of stars coefficient
auto c = STAR_MAX/(*max_element(val_in_bin.begin(), val_in_bin.end()));

for (uint j = 0; j < val_in_bin.size(); j++) { // Print histogram
    std::cout << std::setw(4) << j*100 << std::setw(6)
               << val_in_bin[j] << " " << string(c*val_in_bin[j], '*')
               << std::endl;
}
}
```

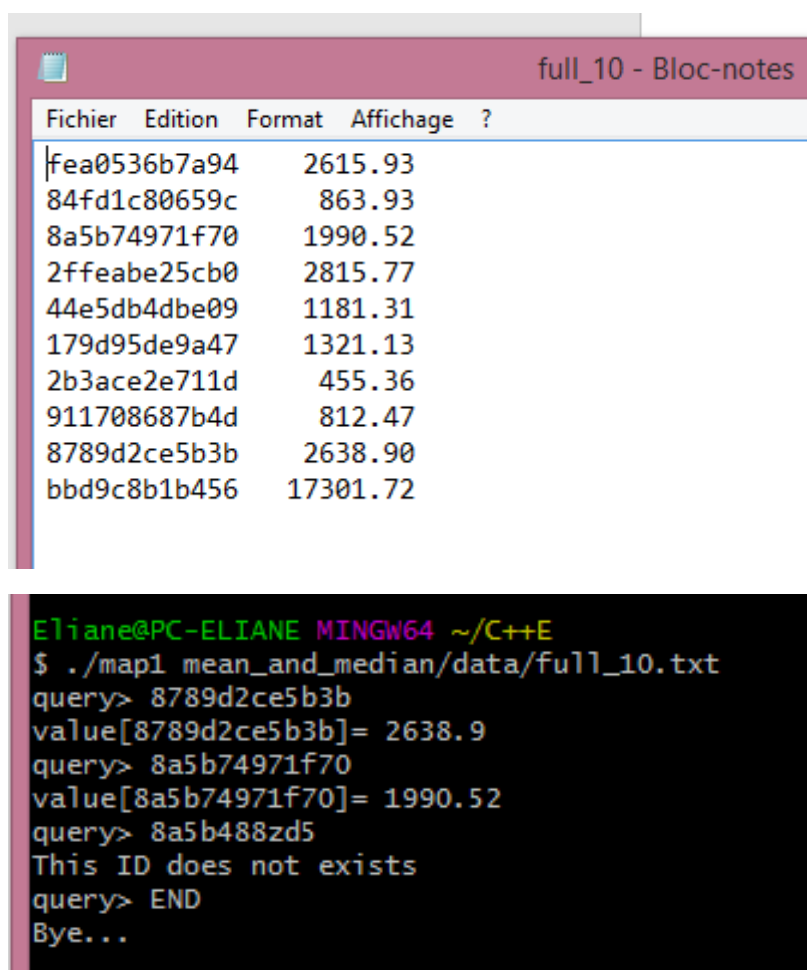
## LES TABLEAUX ASSOCIATIFS

### UNORDERED

Afin de pouvoir accéder aux doubles, il est préférable d'utiliser le conteneur *map*. Ce conteneur permet d'utiliser l'identifiant comme une clé d'accès à la valeur. Cela permet un accès direct et évite d'utiliser des algorithmes de recherche pour trouver chaque valeur.

Comme pour l'exercice précédent, nous reprenons le code du fichier *mean\_and\_median.cpp*, le mettons dans un fichier *map1.cpp* et le complétons afin de réaliser cette fonction. Avec un prompt « **query>** », on entre l'identifiant. Si l'identifiant n'existe pas, la console affiche « **This ID does not exist** ».

Si nous écrivons « **END** », on quitte le prompt « **query>** ».



The image shows two screenshots. The top one is a Notepad window titled 'full\_10 - Bloc-notes' containing a table with two columns: an identifier and a numerical value. The bottom one is a terminal window showing the execution of a C++ program that reads from the file 'full\_10.txt' and responds to queries.

Fichier	Edition	Format	Affichage	?
fea0536b7a94			2615.93	
84fd1c80659c			863.93	
8a5b74971f70			1990.52	
2ffeabe25cb0			2815.77	
44e5db4dbe09			1181.31	
179d95de9a47			1321.13	
2b3ace2e711d			455.36	
911708687b4d			812.47	
8789d2ce5b3b			2638.90	
bbd9c8b1b456			17301.72	

```
Eliane@PC-ELIANE MINGW64 ~/C++E
$ ./map1 mean_and_median/data/full_10.txt
query> 8789d2ce5b3b
value[8789d2ce5b3b]= 2638.9
query> 8a5b74971f70
value[8a5b74971f70]= 1990.52
query> 8a5b488zd5
This ID does not exist
query> END
Bye...
```

On voit que le code de *map1.cpp* fonctionne correctement. Les deux premiers identifiants existent bien et renvoient leur double associé tandis que le 3<sup>ème</sup> identifiant entré ne renvoie rien puisqu'il n'existe pas dans le fichier. « **END** » quitte bien le prompt « **query>** » avec un message « **Bye...** ».

Voici comment nous avons rempli le buffer contenant tous les identifiants et valeurs.

```
map<string, double> buf; // Buffer containing file data
string key;
double d;
while (fin >> key >> d) // Reading file...
    buf[key] = d;
```

Pour le prompt, il suffit d'une boucle infinie qui vient lire les données en entrée et renvoie la réponse sur la sortie standard.

## ORDERED

Le programme *map2* possède les mêmes fonctions que *map1*, mais on rend possible la recherche de clé par valeur. Pour différencier une clé et une valeur, on impose le caractère '+' devant les valeurs.

On utilise à nouveau le container *map*. Le conteneur *bimap* est peut-être plus adaptée à ce programme, mais il demande le téléchargement de la librairie *boost*.

```
91
92 Eliane@PC-ELIANE MINGW64 ~/C++E
93 $ ./map2 mean_and_median/data/full_1000.txt
94 query> 44e2d4b8d7aa
95 value[44e2d4b8d7aa]= 1358.56
96 query> +5000
97 value[375df8b1ac86]= 5022.42
98 query> +616
99 value[1201267a89a7]= 615.25
100 value[7860f4b10a57]= 615.25
101 value[f6a5f1e9f733]= 612.69
102 query> END
103 Bye...
```

Nous devons donc réaliser une boucle qui va parcourir le buffer à la recherche des valeurs comprises dans l'intervalle  $val \pm 1\%$ .

```
// Search buffer for corresponding keys
for (auto it = buf.begin(); it != buf.end(); it++) {
    if (it->second < val*(1+VALUE_MARGIN)
        && it->second > val*(1-VALUE_MARGIN)) {
        std::cout << "value[" << it->first << "]= "
                    << it->second << std::endl;
        found = true;
    }
}
```

## COMPLEXITE

Il est dit dans la documentation du container *map* que la complexité de la fonction *find* est logarithmique par rapport à la taille du conteneur. Cela se justifie par le fait que contrairement à un *vector*<> par exemple, on ne parcourt pas tous les identifiants pour trouver le double voulu.

La complexité d'une boucle for réalisant  $i$  boucles est  $i \cdot O(i)$ , avec  $O(i)$  la complexité des opérations comprises dans la boucle.

Ainsi pour map1 la complexité d'une query est de  $\log(n)$ ,  $n$  le nombre de lignes dans le fichier, et pour map2 la complexité est de  $\log(n)$  pour une recherche par clé et  $n$  pour une recherche par valeur.

## EXERCICE BONUS

Nous avons réalisé le générateur de doubles dans le fichier *random\_gene.cpp*. La génération est semi-aléatoire et suit une loi normale algorithmique de moyenne log 7.6 et d'écart type 0.45.

Pour 100000 valeurs, nous obtenons le résultat suivant avec l'histogramme.

