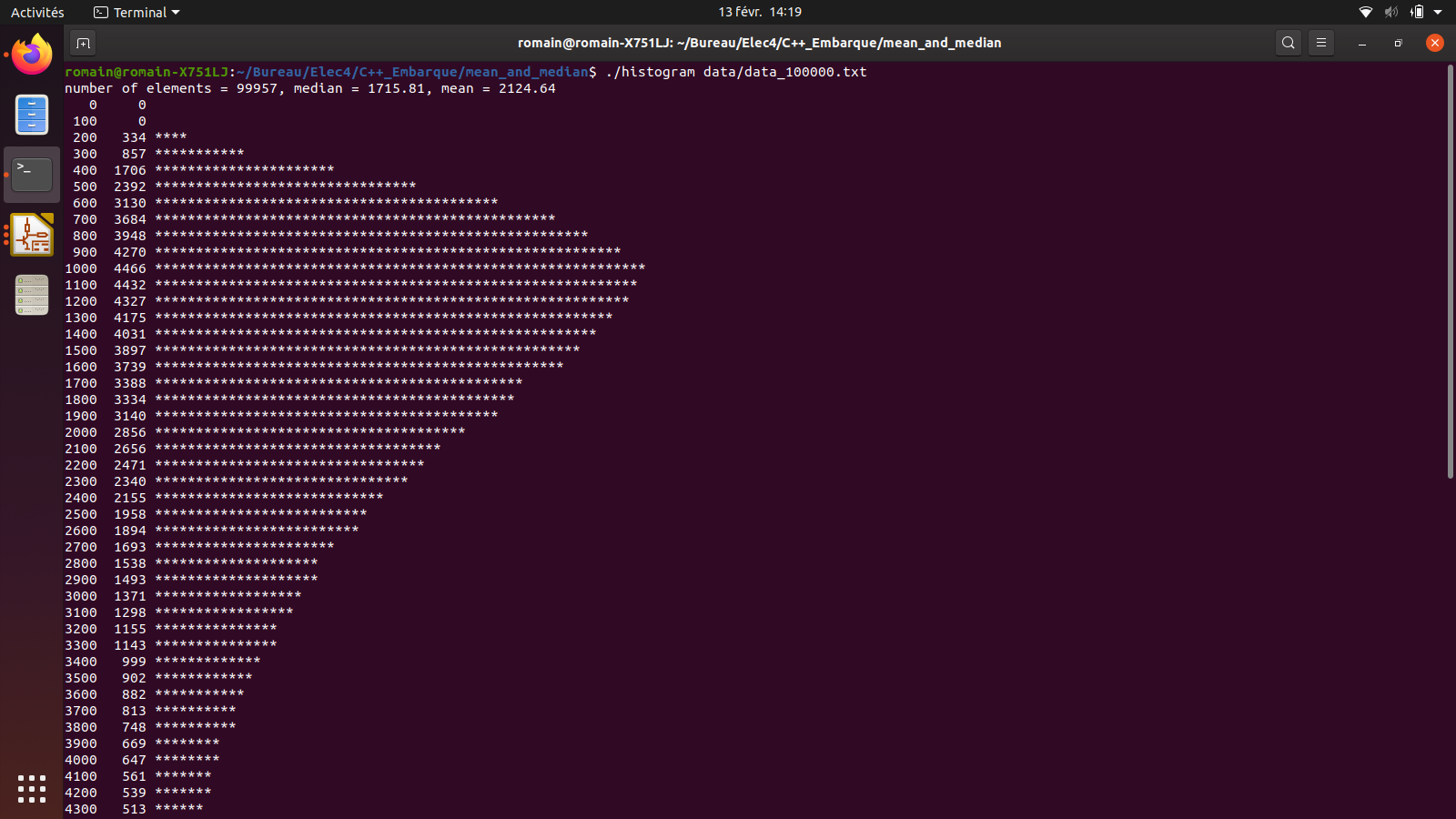
TP 1 « Introduction »

# LES *STRING* ET LES *VECTOR<>*

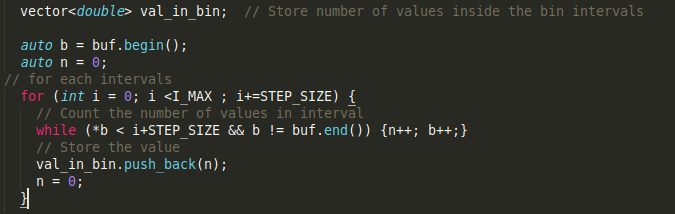
Nous avons repris le code de *mean\_and\_median.cpp*, l’avons dans un fichier *histogram.cpp* et l’avons complété afin d’afficher un histogramme en étoiles sur la console.

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

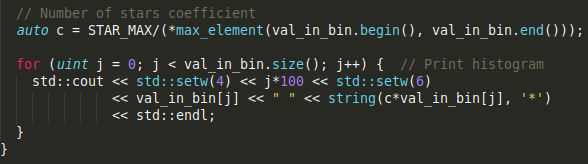


On obtient bien 60 étoiles pour la valeur maximale et un nombre d’étoiles proportionnels pour les autres valeurs.

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.



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.



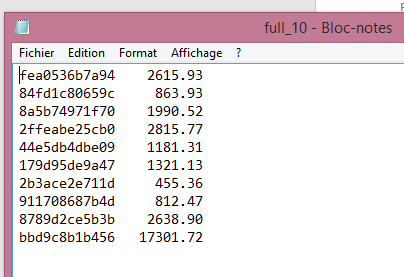
# LES TABLEAUX ASSOCIATIFS

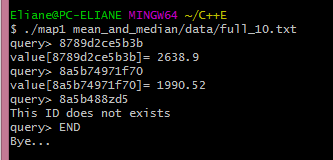
## UNORDERED

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** ».

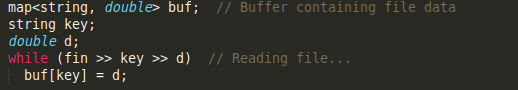
Si nous écrivons « **END** », on quitte le prompt « **query>** ».





On voit que le code de *map1.cpp* fonctionne correctement. Les deux premiers identifiants existent bien et renvoient leur double associé tandis que le 3ème 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.

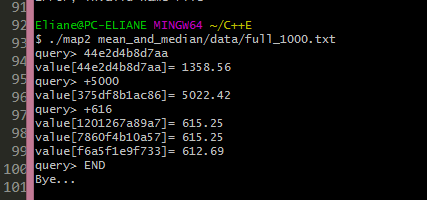


Pour le prompt, il suffit d’une boucle infinie qui vient lire les données en entrée et renvoie la réponse sur la sortie standard.

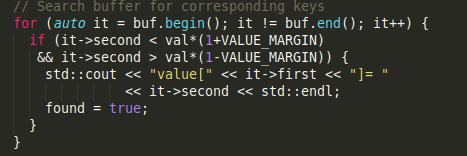
## ORDERED

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 imposer 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 libraire *boost*.



Nous devons donc réaliser une boucle qui va parcourir le buffer à la recherche des valeurs comprises dans l’intervalle *val+-1%.*



## 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\*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.