Compte Rendu – TP 1 – C++ pour l'embarqué

Quentin Combal
Jiang Wei
Paul Tychyj

1. Histogramme

Le code histogram.cpp lit des valeurs dans un fichier, les trie par valeurs en comptant de 100 en 100, puis trace une jauge de caractères '*' représentant le nombre de données dans chaque catégorie.

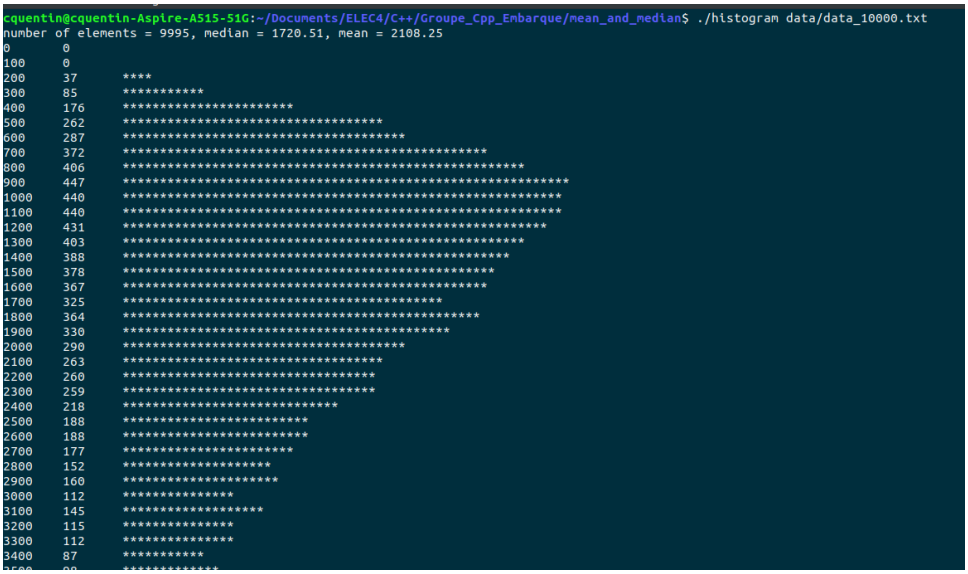
On reprend la structure du code mean_and_median.cpp. On initialise le **vector<double> bin_count(80, 0)**, un vecteur de double de taille 80, initialisé avec toutes ses valeurs à 0. Ensuite, on lit le vecteur **buf** et on incrémente la bonne cellule de **bin_count** pour chaque donnée lui appartenant.

```
// Determine the histogram values
vector<int> bin_count(80, 0);   // Set all the bin counts to 0
uint bin;
for (auto it = buf.begin(); it != buf.end(); it++) {
  auto val = *it;
  bin = val/100;
  bin_count[bin]++;  // Increment the corresponding bin count
}
```

On récupère la valeur maximale du vecteur **bin_count** par la fonction **std::max_elements**, puis on calcule le nombre d'étoiles à imprimer pour chaque cellule en fonction du maximum.

```
  // Plot the histogram
  auto max_elements = *max_element(bin_count.begin(), bin_count.end());
  for (bin = 0; bin < bin_count.size(); bin++) {
    std::cout << bin*100 << "\t" <<
                 bin_count[bin] << "\t" <<
                 string(bin_count[bin]*60 /max_elements, '*') <<  "\n";
  }
}
```

Résultats :



2. Map_1

Pour ce programme, nous avons besoin d'associer chaque identifiant du fichier avec la valeur qui lui est associée. Les containers de type **Map** permettent de stocker des paires clé-valeur dont on choisit les types. Nous utilisons donc un container **Map** dans lequel les clés seront des **String** correspondant aux identifiants et les valeurs seront des **double**.

La lecture des deux champs de la ligne se fait avec l'opérateur >>.

```
string key;
double val;

while (fin >> key >> val) {
  buf[key] = val;
}
```

Lorsque l'utilisateur rentre une clé sur l'entrée standard, on détermine si cette clé existe avec la fonction **map::find**.

```
} else if (buf.find(input) == buf.end()) {
  // Input is a key that does not exist
  std::cout << "ID could not be found" << "\n";

} else {
  // Input is a key that exists
  std::cout << "Value[" << input << "] = " << buf[input] <<"\n";
}
```

Resultats :

```
cquentin@cquentin-Aspire-A515-51G:~/Documents/ELEC4/C++/Groupe_Cpp_Embarque/mean_and_median$ ./Map1 data/full_100.txt
Query> 2ffeabe25cb0
Value[2ffeabe25cb0] = 2815.77
Query> 55520b814cad
Value[55520b814cad] = 1569.62
Query> abcd
ID could not be found
Query> 55520b814ca0
ID could not be found
Query> END
```

3. Map_2

La fonctionnalité ajoutée dans ce programme est la recherche par valeur. A chaque entrée de l'utilisateur, le programme détermine si ce dernier recherche une clé ou une valeur (indiqué par le caractère '+'). Si on recherche une valeur, le programme affiche toutes les clés qui correspondent à la valeur (±1%), y compris si plusieurs clés ont la même valeur.

```
} else if (input[0] == '+') {
  // Input is a value to look for
  // Look for values between v-1% and v+1% and output any matching element
  val = std::stod(input);
  double min = val*0.99;
  double max = val*1.01;

  for (auto it = buf.begin(); it != buf.end() ; ++it) {
    // it-> first is the key and iy->second is the corresponding value
    if ( (it->second > min) && (it->second < max) )
      std::cout << "Value[" << it->first << "] = " << it->second <<"\n";
  }
```

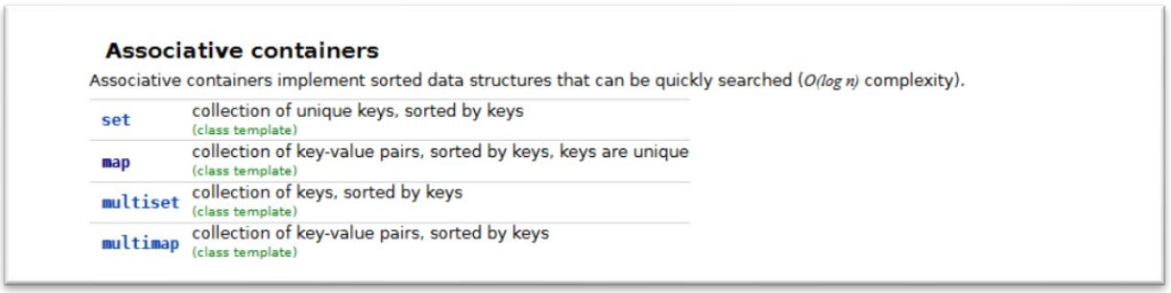
On accède aux clés ou aux valeurs par les membres respectifs de l'itérateur **it->first** ou **it->second.**

Resultats

```
cquentin@cquentin-Aspire-A515-51G:~/Documents/ELEC4/C++/Groupe_Cpp_Embarque/mean_and_median$ ./Map2 data/full_100.txt
Query> 55520b814cad
Value[55520b814cad] = 1569.62
Query> 55520b814ca0
ID could not be found
Query> +1569
Value[209ec51521d1] = 1581.01
Value[2cbc1c276531] = 1582.38
Value[55520b814cad] = 1569.62
Query> +2500
Query> END
Bye...
```

4. Complexité d'une **query** :

Pour une **query** de type **clé**, on utilise la fonction **map::find** pour trouver l'élément recherché. D'après la documentation *cppreference* sur les containers, la recherche sur une **map** de taille n est de complexité **O(log(n))**. Une recherche de clé est donc de complexité logarithmique.



Associative containers

Associative containers implement sorted data structures that can be quickly searched (*O(log n)* complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

En revanche pour une **query** de type valeur, le programme parcourt tout le container pour trouver toutes les valeurs correspondantes. La complexité de ce type de **query** est donc **O(n)**, complexité linéaire.