TP 9 : Généricité

1 Introduction

Le but de ce TP est d'illustrer la généricité paramétrique en écrivant une table de hachage. Une table de hachage est une structure de données qui stocke des paires (clé, valeur). Cela permet par exemple de stocker des dictionnaires ou des tableaux associatifs. On peut par exemple imaginer une table de hachage qui va stocker des dates d'anniversaires. On pourrait alors l'utiliser ainsi dans un main :

```
HashTable t;

t.add("Marcus Garvey", "17/08/1887");
t.add("John Lennon", "9/10/1940");
t.add("Zinedine Zidane", "23/10/1972");

cout << "Zizou est ne le " << t.get("Zinedine Zidane") << endl;
```

Dans ce cas les clés sont des chaînes de caractères (les noms des personnes), et les valeurs sont également des chaînes de caractères (les dates). Cependant on veut pouvoir utiliser la même classe quelque soit le type des clés et le type des valeurs (on pourrait par exemple stocker les dates sous forme d'objets *Date*).

1.1 Définition de la classe générique *Pair*

Comme dit précédemment, une table de hachage contient en fait des paires. Construire donc une classe *Pair* qui contient deux attributs privés $first$ et *second*, de types paramétrés respectifs F et S. Lui ajouter un constructeur par paramètres, des getters, setters, ainsi qu'une surcharge de l'opérateur *operator* << pour pouvoir l'afficher avec *cout* par exemple. L'en-tête du fichier *Pair.h* sera donc le suivant :

```
template<typename F, typename S>
class Pair {
protected:
        F first;
        S second;
public:
        //a completer
};
```

On rappelle que pour la généricité, l'implémentation des méthodes se fait dans le fichier *.h*, à la suite de la définition de la classe.
N'oubliez pas de compiler et tester votre code au fur et à mesure !

1.2 Définition simple de la classe générique HashTable

Construire enfin la classe $HashTable$ paramétrée par K et V, respectivement le type des clés et des valeurs que l'on va stocker. Cette classe ne contient comme attribut qu'un tableau dynamique de paires, ainsi que le nombre de paires que la table contient. Au niveau des méthodes publiques :

- un constructeur par défaut, qui allouera pour notre tableau la place nécessaire (dans un premier temps on réservera 100 cases, et on supposera que cette taille ne sera jamais atteinte).
- une méthode *add* qui prend en paramètre une clé et une valeur, et qui va créer la paire correspondante et l'ajouter dans notre table. On ajoutera dans un premier temps les données les unes à la suite des autres.
- une méthode *get* qui prend en paramètre une clé, et renvoie la valeur correspondante si celle-ci est présente, et $NULL$ sinon.
- une surcharge de l'opérateur *operator* << afin d'afficher toutes les entrées avec *cout* par exemple.

Testez votre classe dans le main en utilisant différents types pour les clés et les valeurs, y compris des objets. Vous remarquerez que pour pouvoir utiliser un objet en tant que clé, il faut que la classe correspondantes surcharge les opérateurs *operator* == (pour la recherche dans la méthode *get*) et *operator* << (pour l'affichage).

1.3 Vrai fonctionnement d'une table de hachage

En fait, une table de hachage n'ajoute pas les éléments à la fin du tableau au fur et à mesure, mais utilise une *fonction de hachage*. Une fonction de hachage est une fonction qui prend en entrée une clé et retourne un entier, appelé le code de hachage. Etant donné une paire (clé, valeur) à insérer, la fonction *add* calcule le code de hachage de la clé, qui, modulo la taille du tableau, donnera l'indice où sera stocké la paire. L'avantage de cette méthode est que lors de la recherche d'un élément, la

méthode *get* n'a pas à parcourir tout le tableau, mais doit juste calculer le code de hachage de la clé désirée pour obtenir directement l'emplacement de la paire.
Une subtilité dans ce fonctionnement est la notion de collision : en effet, selon la fonction de hachage choisie ou bien de l'opération de modulo, deux clés différentes peuvent avoir le même code de hachage. Plusieurs solutions existent pour contourner ce problème. On se contentera ici de trouver la première place vide dans le tableau à partir du code obtenu. Il faut dans ce cas faire attention au cas où l'algorithme arrive à la fin du tableau, en revenant au début. Afin de tester si votre code résiste aux collisions, modifiez le constructeur pour avoir un tableau de taille plus petite, dans le but d'augmenter la probabilité d'avoir des collisions.

Modifier la classe *HashTable* pour respecter ce fonctionnement. Pour cela, ajouter un paramètre générique à la classe qui sera la fonction de hachage. Le début de la définition de classe aura alors cette forme :

```
template<typename K, typename V, int hash(K)>
class HashTable {
...
};
```

Et on utilisera dans la classe la fonction *hash* (paramètre générique) pour calculer le code de hachage d'une clé. Essayer votre classe dans le main en définissant dans un premier temps une table de hachage avec *string* comme type de clé (et ce que vous voulez en type de valeur). Il faut donc également définir une fonction de hachage pour les string. On pourra par exemple utiliser la fonction suivante:

```
int hashString(string s) {
    int code=0;
    for (int i = 0 ; i < s.size() ; i++) {
        code += (int)s[i]*i;
    }
    return code;
}
```

On utilisera ensuite ceci dans notre main de la manière suivante :

```
HashTable<string, int, hashString> table;

table.add("Toto", 0668958472);
table.add("Tata", 0668952411);
```