

TP2 : OPÉRATIONS SUR LES TABLEAUX

```
//Workers.h

#ifndef WORKERS
#define WORKERS
#include <vector>

class Workers{
public:
    Workers(int _nbThreads);
    ~Workers();
    void initialize(std::vector& tab);
    void maximum(std::vector& tab);
    void nbOccurrences(std::vector& tab, int occ);
    void firstOccurence(std::vector& tab, int occ);

    void waitResult();
    int getResult();

private:
    const int nbThreads;
    std::vector<std::thread> threads;
    int results;
};

#endif
```

Le but de ce TP est de créer une classe nommée « Workers » dédiée à faire des opérations en parallèle sur des tableaux. Quatre sortes d'opérations sont à implémenter :

```
// Initialise le tableau en le remplissant de nombre aleatoire
void initialize(std::vector& tab, int size);
// Trouve le maximum du tableau
void maximum(std::vector& tab);
// Trouve le nombre d'occurrences d'une valeur
void nbOccurrences(std::vector& tab, int occ);
// Trouve la position de la premiere occurrence d'une valeur
void firstOccurence(std::vector& tab, int occ);
```

Quand une méthode doit fournir un résultat, celui-ci est obtenu via la méthode `getResult`. Ce TP se divise en deux parties et deux programmes distincts.

Sans la gestion des workers

Dans cette première partie, la méthode `waitResult()` n'est pas à implémenter. Les quatre méthodes (`initialize`, `maximum`, `nbOccurrences` et `firstOccurence`) créent directement un certain nombre de *threads* dès leurs appels puis les détruisent quand elles terminent. Ces méthodes sont bloquantes tant que le calcul associé n'est pas fini.

```
//Main.cc

int main(int argc, char * argv[]){
    if(argc != 2){
        std::cout << "Parameter 1: number of threads" << std::endl;
        return 0;
    }

    Workers workers(atoi(argv[1]));
    std::vector<int> vec();
    workers.initialize(vec,100);
    workers.maximum(vec);
    int max = workers.getResult();

    return 0;
}
```

Avec la gestion des workers

Pour cela, cette classe possède un certain nombre de *threads* créés dès la création d'une instance de la classe (dans le constructeur). Ainsi, les *threads* sont exécutés dans le constructeur de la classe et sont détruits dans le destructeur. Ils doivent alors se mettre en « pause » à la création d'une instance de la classe. Vous devez utiliser des variables conditions pour gérer la synchronisation des *threads*. La méthode `waitResult` attend qu'un travail soit terminé par une des quatre méthodes (`initialize`, `maximum`, `nbOccurrences` et `firstOccurrence`). Cela permet au processus utilisant la classe de faire autre chose pendant que les workers travaillent. Cela doit fonctionner à répétition comme dans cet exemple :

```
//Main.cc

int main(int argc, char * argv[]){
    if(argc != 2){
        std::cout << "Parameter 1: number of threads" << std::endl;
        return 0;
    }

    Workers workers(atoi(argv[1]));
    std::vector<int> vec();
    workers.initialize(vec,100);
    //Some works
    workers.waitResult();
    workers.maximum(vec);
    //Some works
    workers.waitResult();
    int max = workers.getResult();
    return 0;
}
```

Ainsi, quand une des quatre méthodes de calcul est appelée, elle doit réveiller les *threads*, dire au *threads* quel est le travail à effectuer puis les rendormir une fois que la tâche a été réalisée. Quand la méthode `waitResult` est appelée par le processus appelant, celle-ci vérifie si le résultat a été calculé, si ce n'est pas le cas, elle attend que le résultat soit calculé et que les *threads* soient rendormis.