# TP2 C++

HAFEDH Jebalia
SOTIR Paul-Emmanuel Binôme B3311

## INTRODUCTION

La gestion de mémoire est l'un des plus important enjeux de l'informatique de nos jours surtout quand il s'agit des big data. En plus d'apprendre à faire du code de qualité, centré sur les performances, tout en conservant la propreté, ce TP nous permet de d'apprendre à optimiser le code afin améliorer les performances de l'application.

## CHOIX GENERAUX

### Guide de style

Notre guide de style se base sur le guide de style officiel du C++ moderne, disponible sur GitHub (https://github.com/isocpp/CppCoreGuidelines), auquel nous avons ajouté certains choix comme par exemple la convention de nommage (underscore_case), ou les commentaires au format XML définis par Visual Studio.

### Architecture du projet

Travaillant sur Visual Studio, nous avons créé un Makefile générique gérant les dépendances automatiquement pour pouvoir facilement compiler le projet avec gcc. Le Makefile crée des fichiers de dépendance créant de nouvelles règles pour ce Makefile.

Le projet est organisé de manière à isoler les headers, les fichiers sources et les fichiers générés :

```
/
├─ source
│  └─ main.cpp capteur.cpp
├─ includes
│  └─ capteur.h ville.h utils.h vec.h
├─ bin ... Dossier contenant les fichiers générés (exécutable,
│         fichiers objets, fichiers de dépendances).
│  ├─ debug
│  │  └─ dep
│  └─ release
│     └─ dep
└─ tests ... Dossier contenant les tests sous la forme de deux
            fichiers chacun (*.in et *.out) .
```

### Lecture des commandes par l'entée standard

La lecture des commandes se faisant par l'entrée standard et étant donné que l'on peut recevoir une grande quantité de commandes, nous avons choisi, pour des raisons de performances de ne pas utiliser 'std ::cin' mais une fonction plus bas niveau permettant une lecture de l'entrée standard par blocs vers un buffer.

Le code dans la fonction ‘main’ gère cette lecture par blocs et appelle la fonction ‘process_command’ qui parse une commande dans le buffer et l’exécute (appels de méthodes de l’objet de type ville représentant la ville de lyon).

Pour des raisons de propreté (et type-safety) nous avons créé une énumération typée énumérant les types de commandes :

```
enum class cmd { ADD, STATS_C, JAM_DH, STATS_D7, OPT, EXIT };
```

Cependant puisqu’il n’est pas possible de mettre une type std ::string ou char* en underlying type de l’énumération, nous avons créé deux fonction pour convertir une commande (de l’enum ‘cmd’) en la chaine de caractères correspondante et vis versa.

# SPECIFICATION DES CLASSES

## Header utils

Le fichier utils contient quelques fonctions et une classe template ‘pair’ qui peuvent être commun à n’importe quel code de ce TP.

Par exemple, nous avons implémenté notre propre fonction ‘is_nothrow_swappable’. La méta-fonction ‘is_nothrow_swappable’ nous permet de vérifier facilement au compile-type si un ensemble de types, spécifiés grâce à un variadic template, disposent d'une surcharge de la fonction swap marquée ‘noexcept’.

Cette fonction est couramment implémentée avec un ‘trait’ (Structure template faisant office de méta-fonction) mais nous avons préféré la faire sous la forme de fonction récursive marquée ‘constexpr’ :

```
template <typename Head, typename T, typename... Ts>
constexpr bool is_nothrow_swappable() {
	return noexcept(swap(std::declval<Head&>(), std::declval<Head&>())) &&
		is_nothrow_swappable<T, Ts...>(); // Récursion
}

template <typename T>
constexpr bool is_nothrow_swappable() {
	// Fin de la récursion
	return noexcept(swap(std::declval<T&>(), std::declval<T&>()));
}
```

Cette vérification nous permet de faire un ‘static_assert’ pour garantir que tous les swaps effectués dans nos propres surcharges de la fonction swap n'envoient pas d'exception. Nous pouvons ainsi marquer ‘noexcept’ nos surcharges de la fonction swap sans risquer un appel à ‘std::terminate’ (appelé si une exception est lancée dans une fonction ‘noexcept’).

La fonction ‘lowest_pow_of_two_greater_than‘ permet de trouver la plus petite puissance de 2 plus grande qu’un nombre entier donné en paramètre, et ce, le plus rapidement possible. L'implémentation utilise la technique décrite dans une liste d'optimisation d'opérations simples fournie par Stanford (https://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2)

La fonction ‘lowest_pow_of_two_greater_than‘ est une fonction template avec deux surcharges : l'une dédiée aux entiers positifs et l'autre dédiée aux entiers signés. Pour cela nous utilisons le principe "SFINAE" (Substitutuion Failure Is Not An Error) pour que la résolution template choisisse la bonne surcharge en fonction du type donnés en arguments template. Voici les prototypes des deux surcharges :

```
template <typename Integer, typename = typename
       std::enable_if_t <std::is_integral<Integer>::value &&
              std::is_signed<Integer>::value>>
inline Integer lowest_pow_of_two_greater_than(Integer n) { … }

template <typename UnsignedInteger, typename = typename
       std::enable_if_t<std::is_integral<UnsignedInteger>::value &&
              std::is_unsigned<UnsignedInteger>::value>, typename = void>
inline UnsignedInteger lowest_pow_of_two_greater_than(UnsignedInteger n) { … }
```

Les méta-fonctions 'std::is_signed' et 'std::is_unsigned' permettent de discriminer l'une des surcharge en fonction du type d'entier en entrée. L'argument template 'typename = void' dans la seconde surcharge évite que les deux fonctions aient la même signature.

Le fichier utils.h contient également la classe template pair, contenant simplement deux éléments de types T1 et T2. Nous avons surchargé l'opérateur == et != pour cette classe.

## Classe vec

La classe vec est une classe template assez similaire à 'std ::vector'. Un objet vec contient des éléments de type T dans un tableau C de manière à ce que les éléments soient contigus en mémoire.

Lorsqu'un objet 'vec' doit augmenter sa capacité pour ajouter des éléments, nous prenons pour nouvelle capacité la plus petite puissance de 2 inférieure au double de la capacité actuelle. Cela permet des meilleures performances en minimisant le nombre d'allocations dynamiques pour tout nombres d'éléments mais aussi en faisant des opération sur un tableau contigu en mémoire dont la taille est une puissance de 2.

La classe vec contient de nombreuses méthodes dont certaines issues de la classe collection du premier TP :

La méthode 'add' ajoute un élément de type T, un autre objet de type 'vec' au vecteur courant. Dans le cas où l'on ajoute un élément, nous avons créé deux surcharges dont une assez similaire à 'emplace_back' pour 'std ::vector' qui permet de spécifier en argument directement les argument d'un constructeur de l'objet de type T. Cette méthode repose sur les variadic templates et les universal références (rvalue-reference template) afin de pouvoir à la fois 'mover' les rvalues et prendre par référence les lvalues :

```
template<typename... CtorArgsTypes>
T& add(CtorArgsTypes&&... ctor_args); // retourne une référence vers l'objet construit
```

La méthode 'remove' permet de retirer un ou des éléments du vecteur.

Les méthodes 'length' et 'capacity' retourne le nombre d'éléments de 'vec' et sa capacité.

La méthode 'sort' permet de trier les éléments du vecteur grâce à l'algorithme quicksort. Le tri se fait par défaut avec le opérateurs '==' et '<' sur le type T mais l'utilisateur peut spécifier, grâce à un pointeur de fonction, la manière dont sont comparés les éléments. Voici le type des pointeurs de fonction utilisé :

```
using predicate = bool(*)(const T&, const T&);
```

Ces prédicats, utilisés pour comparer les éléments, peuvent être spécifiés lors de la construction d'objets de type 'vec'.

La méthode 'find' trouve l'index d'un élément dans le vecteur. La comparaison se fait avec le prédicat d'égalité (par défaut '=='). La méthode 'find' peut faire soit une recherche dichotomique si le vecteur est trié, soit une simple recherche en itérant sur tous les éléments.

Nous avons également surchargé l'opérateur '[ ]' pour accéder à un élément de 'vec' par son index. Il y a deux surcharges de cet opérateur : une première retourne une copie de l'objet et peux donc être marquée 'const' et une seconde retournant une référence vers l'élément (utile pour modifier l'élément).

La classe 'vec' suit la règle de 4, c'est-à-dire qu'elle implémente : son constructeur de copie, son constructeur de 'move', son opérateur d'assignement par copie et son destructeur. L'opérateur d'assignement par 'move' n'ayant pas besoin d'être implémenté car nous prenons le paramètre de « l'opérateur d'assignement par copie » par valeur (explications plus détaillée). De plus nous suivons le principe de 'copy-and-swap idiom' qui consiste à implémenter la fonction swap (pouvant être une surcharge de 'std ::swap') qui sera utilisé pour implémenter le 'copy assignement operator' et le 'move-constructor'. La fonction swap permet donc d'éviter la duplication de code mais elle aide également à rendre la classe plus sûre au niveau des exceptions :

```
template<typename U>
void swap(vec<U>& lhs, vec<U>& rhs) noexcept;
```

Nous pouvons garantir que la fonction ne lancera pas  d'exceptions, et ne fera pas d'appel à 'std ::terminate' grâce à 'noexcept' et à l'assertion statique (dans la fonction swap):

```
// if one of the following swap calls isn't noexcept, we raise a static_assert
static_assert(is_nothrow_swappable<type_qui_sera_swapé1, type_qui_sera_swapé2, …>(),
       "Swap function could throw and let vec<T> objects in incoherant state!");
```

Les éventuelles exceptions auront donc forcément lieu dans le constructeur de copie où l'on fait attention à toujours laisser l'objet de type 'vec' dans un état cohérent.

Enfin, la classe 'vec' dispose de deux autres constructeur : l'un prenant deux paramètres optionnels pour choisir les prédicats de comparaison :

```
explicit vec::vec(   predicate eq_pred = &default_equality_predicate<T>,
                     predicate comp_inf_pred = &default_inferior_comp_predicate<T>);
```

Et l'autre prenant également la capacité de la collection ainsi que sa capacité maximale:

```
explicit vec::vec(size_type capacity, size_type max_capacity = MAX_ALLOCATION_SIZE,
       predicate eq_pred = &default_equality_predicate<T>,
       predicate comp_inf_pred = &default_inferior_comp_predicate<T>);
```

## Structure capteur_event

La structure 'capteur_encent' est une structure représentant un évènement ADD. Cette structure permet notamment de conserver en mémoire tous les événements en prenant un minimum de mémoire (utile pour la commande OPT). La structure tient sur seulement 32 bits grâce aux 'bit fields', ce qui nous permet de loger 20 000 000 d'évènements sur 80 mégaoctets :

```
struct capteur_event
{
       // ...
       using sensor_t = uint32_t; // alignement sur 32 bits
       sensor_t id : 16;    // (0 - 65536)       // id du capteur (on assume que id ne
                                                                  dépasse pas 65536)
       sensor_t d7 : 3;     // (0 - 7)           // Jour de la semaine
       sensor_t hour : 5;   // (0 - 31)          //  Heure de l'événement
       sensor_t min : 6;    // (0 - 63)          // Minute de l'événement
private:
       sensor_t traff : 2;  // Etat du capteur (trafic)
       // ...
} ;
```

L'attribut 'traff' représente l'état du trafic sur deux bits en interne mais partout ailleurs nous utilisons l'énumération typée 'traffic'. Des accesseurs permettent de modifier et accéder à 'traff' en utilisant cette énumération :

```
enum class traffic : char { vert = 'V', rouge = 'J', orange = 'R', noir = 'N' };
```

La structure dispose d'un constructeur permettant d'initialiser ses attributs :

```
capteur_event(sensor_t id = 0, sensor_t d7 = 1, sensor_t hour = 0, sensor_t min = 0,
       traffic state = traffic::vert);
```

La structure dispose également de surcharges d'opérateurs comme '==' , ' !=', '<', '>', '>=' et '<=' ainsi que la fonction 'swap'. Ces comparaison sont rapides car reposent sur l'ordre des attributs dans la structure et le fait que la structure loge sur 32bits. Par exemple l'opérateur '<' retourne :

```
return reinterpret_cast<const uint32_t&>(lhs) & capteur_event::COMPARISION_MASK <
       reinterpret_cast<const uint32_t&>(rhs) & capteur_event::COMPARISION_MASK (ignore
              l'attribut traff grâce à COMPARISION_MASK)
```

## Structure capteur_stat

La structure capteur_stat représente des statistiques sur un capteur. La structure contient un compteur pour chaque valeurs de l'énumération 'traffic' permettant d'en déduire les pourcentages demandés. Un attribut 'm_stat_timestamp' de type capteur_event représente le timestamp correspondant aux statistiques du capteur. Par exemple pour des statistiques par heures dans la semaine, seul l'heure, l'id, et le jour de la semaine (d7) seront pris en considération dans l'attribut 'm_stat_timestamp'.

Une méthode update met à jour les statistiques en prenant en paramètre un 'capteur_event'.

La structure dispose des mêmes opérateurs que capteur_event et les comparaisons se font seulement sur l'attribut 'm_stat_timestamp'.

## Classe ville

La classe ville représente une ville contenant toutes les statistiques de capteurs et tous les évènements relatifs à un capteur. Cette classe contient des méthodes pour chacune des commandes donnée en entrée :

```
void add_sensor(capteur_event new_event); // ADD
void show_day_traffic_by_hour(capteur_stat::sensor_t d7) const; // JAM_DH
void show_day_traffic(capteur_stat::sensor_t d7) const; // STATS_D7
void show_optimal_timestamp(capteur_stat::sensor_t d7, capteur_stat::sensor_t h_start,
       capteur_stat::sensor_t h_end, const vec<capteur_stat::sensor_t>& seg_ids); // OPT
capteur_stat* get_sensor_stat_by_id(capteur_stat::sensor_t id); // STATS_C
```

Les statistiques concernant les commandes JAM_DH, STATS_D7 sont mises à jour dans 'add_sensor' et stockées respectivement dans les attributs :

```
using hour_jam_stat = pair<unsigned int, unsigned int>;
hour_jam_stat m_week_jam_distribution_by_hour[DAYS_COUNT][HOUR_COUNT];
struct day_traffic_stat { /*...*/ } // structure privée contentant les statistiques d'une
                                       journée de la semaine pour tous les capteurs.
day_traffic_stat m_week_traffic_distribution_by_day[DAYS_COUNT];
```

De plus, la méthode 'add_sensor' stocke les évènements (pour OPT) et les statistiques pour chaque capteur (pour STATS_C) dans l'attribut :

```
using capteur_stat_with_events = pair<capteur_stat, vec<capteur_event>>;
vec<capteur_stat_with_events> m_sensor_stats;
```

‘m_sensor_stats’ est un vecteur contenant un élément pour chaque capteur (chaque ID différents). Cet élément est une paire contenant les statistiques de ce capteur pour la commande STATS_C ainsi que qu’un vecteur contenant tous les événements relatifs à ce capteur.

### Commande OPT

Pour trouver le trajet le plus court en suivant une route dont chaque segment a un capteur à la minute près, il faut nécessairement stocker en mémoire une grande partie des informations issues des commandes ADD. C’est pourquoi nous avons choisi de stocker en mémoire tous les événements.

Nous aurions pu stocker des statistiques pour chaque minute de la semaine au lieu de garder tous les événements mais étant donné qu’un objet ‘capteur_stat’ prend 20 octets en mémoire, dans le pire des cas (un événement pour chaque minutes de la semaine pour chaque capteur) il faudrait stoker 1500 * 7 * 24 * 60 * 20 octets soit environs 300MO en mémoire. Alors que les 20 000 000 d’évènements ne prennent que 80 MO (4 octets par événements). De plus, simplement stoker les évènements plutôt que d’en faire des statistiques lors d’un appel à la méthode ‘add_sensor’ (lors d’un ADD) permet de rendre plus rapide cette méthode, ce qui est important puisque l’utilisateur n’a pas nécessairement besoin de ces statistiques (n’appelle pas forcément la commande OPT). Enfin, cela permet d’avoir un programme plus réutilisable pour des ajouts futurs car à partir des évènements on peut en déduire n’importe quelles statistiques pour une nouvelle commande.

Une fois les événements en mémoire, lors de l’appel à la commande OPT, nous calculons le temps de trajet pour toutes les heures et minutes de départ dans l’intervalle spécifié. Si il n’y a pas eu d’événement pour un capteur donné à une minute donnée, nous considérons que le trafic est maximal (noir). Enfin quand il y a eu plusieurs évènements dans la même minute de la semaine (la semaine dans le mois, le mois ou l’année peuvent diffèrer), nous prenons la moyenne des temps de trajet correspondant aux valeurs de trafic des évènements.

## CONCLUSION ET AMELIORATIONS POSSIBLES

multithread, asynchrone, entrée des commandes autrement que par l'entrée standard (lecture d'un fichier par ex), ...