

# Algorithmes et structures de données pour ingénieurs

GLO-2100

**Travail à faire individuellement  
À rendre avant la date indiquée  
sur le portail du cours**  
(Voir modalités de remise à la fin de l'énoncé)

Tout travail remis constitue une contribution originale et distincte des travaux remis par d'autres. Le plagiat est strictement défendu. Tout travail plagié sera transmis au Commissaire aux infractions relatives aux études de l'Université Laval et sera donc passible de sanctions très punitives. De plus, les étudiants ou étudiantes ayant collaboré au plagiat seront soumis aux sanctions normalement prévues à cet effet par les règlements de l'Université.

## *Travail Pratique #1*

*Utilisation des conteneurs de la  
STL pour la mise en œuvre d'une  
classe gérant l'accès aux  
données GTFS*



UNIVERSITÉ  
**LAVAL**

Faculté des sciences et de génie  
Département d'informatique  
et de génie logiciel

# 1 Objectif général

Ce TP constitue le premier TP d'une séquence de 3 TP portant sur les données du réseau de transport de la capitale (RTC) de la ville de Québec. Le but du TP2 sera de mettre en œuvre les algorithmes et les structures de données vous permettant d'obtenir l'itinéraire du voyage (empruntant le réseau de la RTC) le plus rapide pour vous déplacer d'un point à un autre quelques soient ces points. Pour y arriver, vous devrez, dans ce TP1, implémenter la classe `DonneesGTFS` dont le but est de gérer l'accès aux données GTFS de la RTC. Pour arriver à faire ce qui est demandé dans ce TP1 vous devrez utiliser les conteneurs standard de la STL tels que `vector`, `set`, `map`, `multimap`, `unordered_set`. L'objectif de ce TP est donc de vous permettre de vous familiariser avec l'utilisation de ces conteneurs afin que vous puissiez, à l'avenir, les utiliser correctement et efficacement.

## 2 Données ouvertes de la RTC

Nous utiliserons les données ouvertes du Réseau de Transport de la Capitale (RTC), qui répertorient, entre autres, l'emplacement des stations de bus dans la ville et les informations sur les lignes de bus (comme les horaires et les arrêts). Ces données sont disponibles dans un format compressé (zip) à l'adresse suivante: <http://www.rtcquebec.ca/Portals/0/Admin/DonneesOuvertes/googletransit.zip> Une fois le dossier décompressé, vous verrez que celui-ci contient plusieurs fichiers textes CSV (Comma Separated Values) ayant les caractéristiques suivantes:

- le séparateur de colonnes est la virgule (,);
- les données de certaines colonnes sont entre guillemets;
- la première ligne précise les titres des différentes colonnes.

Les fichiers de cet archive que nous utiliserons sont: `calendar_dates.txt`, `routes.txt`, `stops.txt`, `stop_times.txt`, `transfers.txt`, `trips.txt`.

Notez que tous les fichiers adhèrent au format GTFS ([https://fr.wikipedia.org/wiki/General\\_Transit\\_Feed\\_Specification](https://fr.wikipedia.org/wiki/General_Transit_Feed_Specification)), un format standardisé pour communiquer des horaires de transports en commun et les informations géographiques associées. Ainsi, outre le RTC, la Société de Transport de Montréal (STM) utilise ce même format pour rendre publique ses horaires d'autobus et de métro. Ainsi, une application basée sur ce format pourra être éventuellement utilisée pour traiter différents réseaux de transport en commun.

**Dans le but d'uniformiser la correction, nous vous fournissons avec cet énoncé une version assez récente des horaires publiés du réseau de transport. Il va donc falloir que vous utilisiez exclusivement le dossier RTC fourni avec cet énoncé afin d'éviter de perdre des points.**

## 3 Les classes fournies

Pour ce TP, nous vous fournissons plusieurs classes déjà implémentées. Les fichiers `.h` et `.cpp` sont fournies en archive et sont documentés. **On vous demande de ne pas modifier ces classes.** Ces classes sont les suivantes :

- **Coordonnees**: utilisée pour représenter les coordonnées GPS d'un endroit. Principalement, cette classe est munie d'une méthode statique `is_valid_coord` permettant de valider si une paire potentielle (lat, long) forme une coordonnée GPS valide (voir [https://fr.wikipedia.org/wiki/Coordonn%C3%A9es\\_g%C3%A9ographiques](https://fr.wikipedia.org/wiki/Coordonn%C3%A9es_g%C3%A9ographiques)). Aussi, elle est munie d'un opérateur ( - ) de calcul de la distance (donc toujours positive) entre deux coordonnées GPS. L'implémentation de ce calcul a été réalisée grâce aux informations se trouvant au [https://fr.wikipedia.org/wiki/Distance\\_du\\_grand\\_cercle](https://fr.wikipedia.org/wiki/Distance_du_grand_cercle). Notez que le constructeur de cette classe ne construit un objet de la classe que si les données en paramètre sont valides.
- **Station**: Une station est un emplacement physique où un bus effectue des arrêts. Un objet station est construit à partir des données que l'on retrouve dans le fichier `stops.txt`. Chaque ligne de ce fichier représente une station. Le fichier `stops.txt` est composé des colonnes suivantes :
  - `stop_id` : identifiant unique d'une station ;
  - `stop_name` : contient le nom de la station ;
  - `stop_desc` : contient la description de la station ;
  - `stop_lat` : contient la latitude de la station au format wgs84 (voir [https://fr.wikipedia.org/wiki/WGS\\_84](https://fr.wikipedia.org/wiki/WGS_84));
  - `stop_lon` : contient la longitude de la station au format wgs84 ;
  - `stop_url` : contient l'URL d'une page décrivant une station en particulier ;
  - `location_type` : différencie les stations entre-elles ; si ce champ est vide, ce qui est le cas dans les données publiées par le RTC, c'est une station par défaut ;
  - `wheelchair_boarding` : identifie si l'embarquement de fauteuil roulant est possible à cette station ; une valeur
    - 0 (ou vide) indique que cette information n'est pas disponible pour cette station ;
    - 1 indique qu'un véhicule passant par cette station peut accueillir au moins un fauteuil roulant ;
    - 2 indique qu'aucun fauteuil roulant ne peut être pris en charge à cette station.

Dans le cadre de ce travail, nous n'utilisons qu'une partie de ces données ; plus précisément `stop_id` (`m_id`), `stop_name` (`m_nom`), `stop_desc` (`m_description`), `stop_lat` et `stop_long` (`m_coords`).

Notez que l'attribut `m_arrets` n'est pas initialisé explicitement par le constructeur. Ce conteneur `multimap<Heure, Arret :Ptr>` est donc vide initialement et sera rempli grâce à la méthode `addArret`. Les classes `Heure` et `Arret :Ptr` sont décrites plus loin. Un `multimap` est un map permettant des duplicatats de clés. Alors que dans un map, chaque élément doit avoir une clé unique, dans un `multimap` plusieurs éléments peuvent avoir la même valeur de clé. La clé pour ce conteneur est un objet de type `Heure`. En quelque sorte, ce conteneur contient tous les arrêts se produisant à cette station ordonnés selon l'heure d'arrivée de l'arrêt. Mais pour économiser la mémoire, seulement les pointeurs (`shared_ptr`) aux arrêts sont stockés. Le parcours de ce conteneur par itération du premier au dernier élément visite les arrêts en ordre croissant de leur heure d'arrivée. La classe `Arret` est décrite plus loin.

- **Ligne**: Cette classe représente une ligne d'autobus du réseau de transport contenue dans le fichier `routes.txt`. Un objet `Ligne` est construit à partir des données que l'on retrouve dans le fichier `routes.txt`. Chaque ligne de ce fichier représente un objet `Ligne`. Pour votre information, le fichier `routes.txt` est composé des colonnes suivantes :
  - `route_id` : identifiant unique de la ligne d'autobus ;
  - `agency_id` : identifiant unique de l'agence (RTC) ;

- `route_short_name` : texte permettant d'identifier un parcours sur les horaires et panneaux de signalisation ; typiquement, il s'agit du numéro de la ligne ("7", "800", "801", "13A", "13B", etc.) ;
- `route_long_name` : texte permettant d'identifier, avec plus de précisions, une ligne sur les horaires et panneaux de signalisation (ce champ est vide dans le fichier fourni par le RTC) ;
- `route_desc` : texte décrivant la ligne ; le RTC a choisi de préciser les destinations desservies ;
- `route_type` : identifie le type de véhicule de transport 4 ; la valeur est fixée à 3 pour indiquer qu'il s'agit d'autobus ;
- `route_url` : comprend une URL qui mène vers une page décrivant la ligne en question ;
- `route_color` : couleur permettant d'identifier visuellement la ligne ;
- `route_text_color` : couleur permettant d'identifier visuellement un texte décrivant la ligne.

Nous n'allons qu'utiliser: `route_id` (`m_id`), `route_short_name` (`m_numero`), `route_desc`(`m_description`), `route_color`(`m_categorie` de type `CategorieBus`: voir ci-dessous).

Notez qu'il peut exister plus d'une ligne avec le même numéro (`route_short_name`) dans le fichier `routes.txt`. Il peut donc y avoir deux id différents possédant le même numéro (ex : la 800). Cela est dû au fait qu'un dossier GTFS peut chevaucher plusieurs saisons différentes d'une année (et des saisons différentes donnent des ids différents à un même numéro de ligne). C'est donc `route_id`(`m_id`) qui est l'identifiant d'une ligne et non son `route_short_name`(`m_numero`).

- **CategorieBus**: il s'agit d'une énumération permettant de différencier les différents types de bus. La correspondance entre cette énumération et le champ `route_color` dans le fichier `routes.txt` est la suivante :
  - - "97BF0D" (Verte) <--> Métro bus
  - - "013888" (Bleue) <--> Le bus
  - - "E04503" (Orange) <--> Express
  - - "1A171B" (Noir) ou "003888" (Bleue) <--> Couche tard
- **Voyage**: permet de décrire un voyage d'une ligne, i.e., un trajet entre deux stations terminales. Un objet voyage est construit à partir des données que l'on retrouve dans le fichier `trips.txt`. Chaque ligne de ce fichier représente un voyage. Pour votre information, le fichier `trips.txt` est composé des colonnes suivantes :
  - `route_id` : identifiant unique d'une ligne ;
  - `service_id` : identifiant unique d'un ensemble de voyages offerts par la RTC ;
  - `trip_id` : identifiant unique d'un voyage ;
  - `trip_headsign` : indique aux passagers la destination du voyage ;
  - `trip_short_name` : texte permettant d'identifier un trajet sur les horaires et panneaux de signalisation ;
  - `direction_id` : valeur binaire indiquant le sens du voyage ;
  - `block_id` : champ identifiant le bloc auquel appartient le voyage ; un bloc est constitué de deux ou plusieurs voyages successifs effectués avec le même véhicule ;
  - `shape_id` : contient l'identifiant de la forme de l'itinéraire ;
  - `wheelchair_accessible` : identifie si l'embarquement de fauteuil roulant est possible à une certaine station du voyage.

Dans le cadre de ce travail, nous n'utilisons que `trip_id`(`m_id`), `route_id`(`m_ligne`), `service_id`(`m_service_id`), `trip_headsign`(`m_destination`).

Notez que l'attribut `m_arrets` n'est pas initialisé dans le constructeur. Ce conteneur `set<Arret::Ptr, compArret>` est donc vide initialement et pourra être rempli grâce à la méthode `ajouterArret`. Le type `Arret::Ptr` est décrit ci-dessous (voir classe `Arret`) et le type `compArret` est le foncteur de comparaison utilisé pour ordonner les éléments `Arret::Ptr`. En gros, les Arrêts (plus, spécifiquement les pointeurs d'arrêts) sont ordonnés selon leur numéros de séquence. Comme décrit à la classe `Arret`, chaque arrêt d'un même voyage possède un numéro de séquence spécifiant l'ordre de l'arrêt dans le voyage. Les `Arret::Ptr` stockés dans ce conteneur sont stockés par ordre de leur numéro de séquence.

- **Arret:** Un arrêt est une composante d'un voyage, c'est une opération spatio-temporelle qui s'effectue lors d'un voyage d'une ligne donnée (ex: la ligne 800 fait, lors du voyage xxxxx, un arrêt à la station du Desjardins à 11h32). Il est important de ne pas confondre la station et l'arrêt. Un objet `Arret` est construit à partir des données se trouvant dans le fichier `stop_times.txt`. Chaque ligne de ce fichier représente un arrêt. Le fichier `stop_times.txt` est composé des colonnes suivantes :
  - `trip_id` : identifiant du voyage ;
  - `arrival_time` : spécifie l'heure d'arrivée de l'autobus pour cet arrêt (d'un voyage en particulier) ;
  - `departure_time` : spécifie l'heure de départ de l'autobus pour cet arrêt (d'un voyage en particulier) ;
  - `stop_id` : identifiant unique d'une station ;
  - `stop_sequence` : identifie le numéro de séquence de cet arrêt dans l'ensemble des arrêts effectués pour un voyage en particulier (il permet ultimement de reconstituer l'ensemble des arrêts effectués pour un voyage dans le bon ordre) ;
  - `pickup_type` : indique si les passagers peuvent embarquer à l'arrêt en question ou si seules les descentes sont permises ;
  - `drop_off_type` : indique si les passagers sont déposés à l'arrêt selon l'horaire prévu ou que le débarquement n'est pas disponible.

Nous n'aurons besoin que de `stop_id` (`m_station_id`), `arrival_time` (`m_heure_arrivee`), `departure_time` (`m_heure_depart`), `stop_sequence` (`m_numero_sequence`), et `trip_id` (`m_voyage_id`). Cette classe surcharge les opérateurs `<` et `>` qui permettent de comparer deux arrêts selon leur heure d'arrivée et de départ (voir l'implémentation).

**Important :** notez que cette classe définie (à l'aide d'un `typedef`) le type `Arret::Ptr` comme étant le type `shared_ptr<arret>`. Le type `shared_ptr<>` a été introduit avec le standard C++11. Les prototypes des méthodes de cette classe sont définis dans `<memory>`. Un objet type `shared_ptr<Objet>` est un pointeur à un objet de type `Objet` muni d'un compteur de références et de méthodes permettant l'allocation de l'objet `Object` et sa destruction lorsqu'il n'existe plus de `shared_ptr<Object>` pointant sur l'objet. De plus, si l'objet doit être déplacé en mémoire tous les `shared_ptr` pointant sur l'objet seront assignés automatiquement à la bonne adresse; ce qui n'est évidemment pas le cas avec les pointeurs ordinaires et qui entraîne de gros problèmes.

Pour créer un `shared_ptr<Arret>`, il suffit de faire :

```
shared_ptr<Arret> arret_ptr = make_shared<Arret>(...);
```

où (...) désigne l'ensemble des paramètres du constructeur d'un objet Arret. Ainsi dans cet exemple, `arret_ptr` est maintenant un shared pointer pointant sur un objet Arret existant dans le « heap ». Vous pouvez alors stocker une copie de `arret_ptr` dans différents conteneurs. L'objet Arret sera détruit seulement (et automatiquement) lorsque tous les shared pointers pointant sur lui auront été détruits. Dans l'exemple ci-dessus, puisque `arret_ptr` a été créé sur la Pile (i.e., l'espace temporaire dans le corps d'une méthode), `arret_ptr` sera détruit lorsque la méthode terminera; ce qui décrémentera le compte de référence à l'objet Arret. Vous n'aurez donc pas à détruire explicitement l'objet avec l'opérateur `delete` car ça sera fait automatiquement lorsque le compte de référence pour cet objet deviendra zéro. Pour connaître le nombre de `shared_ptr` pointant sur l'objet Arret vous pouvez faire `arret_ptr.use_count()`;

Finalement, l'énoncé dans la classe Arret :

```
typedef shared_ptr<Arret> Ptr;
```

Permet d'utiliser `Arret::Ptr` à la place `shared_ptr<Arret>`. Donc, pour créer le `shared_ptr` `arret_ptr`, nous pouvons simplement faire :

```
Arret::Ptr arret_ptr = make_shared<Arret>(...)
```

Pour plus d'informations sur les shared pointers, voir la bibliothèque <memory> à [www.cplusplus.com/reference/](http://www.cplusplus.com/reference/).

Nos conteneurs `m_arrets` dans `Voyage` et `Station` contiennent donc des `shared_ptr<Arret>`. Typiquement, chaque arret que l'on crée sur le « heap » est pointé par un `shared_ptr<Arret>` dans le conteneur `m_arrets` de `Station` et par un `shared_ptr<Arret>` dans le conteneur `m_arrets` de `Voyage` car chaque arret appartient à la fois à un voyage (`m_voyage_id`) et à une station (`m_station_id`).

- **Heure**: représente l'heure d'une journée, mais pour être compatible avec les données de la RTC nous permettons qu'elle puisse encoder un nombre d'heures supérieures à 24 mais inférieure à 30 (i.e., 26:02:00 est une heure valide et utilisée par la RTC). Cela est dû au fait qu'un service du réseau de transport qui a commencé dans une journée peut se terminer plus tard que minuit de « cette journée » mais sûrement avant le prochain service de la journée suivante qui commence à 6h. Le constructeur par défaut de la classe instancie un objet avec l'heure actuelle alors que celui avec des paramètres instancie l'objet à l'heure identifiée par ses paramètres. Cette classe surcharge les opérateurs de comparaison ainsi que l'opérateur ( - ) qui retourne la différence entre deux heures en nombre de secondes.
- **Date**: représente une date. La date courante est créée avec le constructeur par défaut et une date quelconque est créée avec le constructeur avec paramètres. La classe est munie des opérateurs de comparaison.
- Le fichier **calendar\_dates.txt** contient des données relatives aux dates où les bus sont opérationnels. Pour votre information, il est composé des champs :
  - `service_id` : identifiant unique d'un ensemble de voyages effectuées par diverses lignes selon un horaire de service défini. En gros, vu que les voyages du dimanche ont un horaire différent de ceux en semaine, il peut y avoir un `service_id` pour les voyages en semaine et un autre pour les voyages du dimanche ;

- date : spécifie une date durant laquelle un service est offert ;
- exception\_type : indique si le service est disponible à la date spécifiée.

Notez qu'un service prédéfini (service\_id) peut être donné à plusieurs dates différentes et une date peut être associée à plusieurs services prédéfinis. Ceci étant dit, étant donnée une date, ce sont les service\_ids qui lui sont associés qui permettent de trouver les voyages à cette date car chaque voyage est effectué dans le cadre d'un horaire de service prédéfini (service\_id).

L'interface et l'implémentation Date et Heure se trouvent dans auxiliaires.h et auxiliaires.cpp.

## 4 Travail à faire

Pour ce TP, vous devez implémenter certaines méthodes de la classe DonneesGTFS. Les fichiers .h et .cpp de cette classe sont fournis avec cet énoncé, mais il vous reste à implémenter certaines méthodes en respectant rigoureusement les prototypes fournis. Le non respect d'un prototype entraine automatiquement la note de zéro pour l'implémentation de la méthode.

La classe DonneesGTFS encapsule les données de la RTC et en permet leur gestion et manipulation. Elle servira grandement à la réalisation de votre TP2 qui lui, sera plus ambitieux que ce premier TP.

Cette classe est constituée des attributs suivants :

```
std::vector<std::string> string_to_vector(const std::string &s, char delim);

Date m_date; //la date d'intérêt
Heure m_now1; //l'heure de début d'intérêt (à partir de laquelle on considère les arrêts)
Heure m_now2; //l'heure de fin d'intérêt (à partir de laquelle on ne considère plus les arrêts)

unsigned int m_nbArrets; //le nombre d'arrets au total présents dans cet objet
bool m_tousLesArretsPresentes; //indique si tous les arrêts de la date et de l'intervalle [now1, now2) ont été ajoutés

std::unordered_map<unsigned int, Ligne> m_lignes; //la clé unsigned int est l'identifiant m_id de l'objet Ligne
std::map<unsigned int, Station> m_stations; //la clé unsigned int est l'identifiant m_id de l'objet Station
std::unordered_set<std::string> m_services; //le string est l'identifiant du service (service_id)
std::map<std::string, Voyage> m_voyages; //le string est l'identifiant (trip_id) de l'objet Voyage
std::vector<std::tuple<unsigned int, unsigned int, unsigned int>> m_transferts; // <from station id, to station id, min_transfer_time>
std::multimap<std::string, Ligne> m_lignes_par_numero; //le string est l'attribut m_numero de l'objet ligne
```

Le constructeur est déjà implémenté et initialise les attributs m\_date, m\_now1 et m\_now2 selon la valeur des paramètres fourni au constructeur. Il initialise également m\_nbArrets à zéro et m\_tousLesArretsPresentes à false.

Seuls les services et voyages du jour spécifié par m\_date sont ajoutés dans l'objet DonneesGTFS. De plus seuls les arrêts de ce jour existants durant l'intervalle de temps compris dans [m\_now1, m\_now2) seront ajoutés dans l'objet DonneesGTFS.

Vous devez absolument utiliser tous les attributs membres tels que spécifiés. Vous avez donc l'obligations d'utiliser les conteneurs m\_lignes, m\_stations, m\_services, m\_voyages, m\_transferts, et m\_lignes\_par\_numero selon leur type spécifié ci-dessus (et dans DonneesGTFS.h). Vous pouvez ajouter

d'autres attributs si vous le désirez et d'autres méthodes privées ou publiques pour vous faciliter la tâche, mais ce qu'il y a présentement dans `DonneesGTFS` est largement suffisant pour vos besoins.

Toutes les lignes présentes dans le fichier `routes.txt` doivent être ajoutées aux conteneurs `m_lignes` et `m_lignes_par_numero`. Notez qu'un conteneur `unordered_map<Key, Value>` s'utilise comme un `map<Key, Value>` à l'exception du fait que c'est un conteneur non ordonné. Les paires `<Key, Value>` ne sont donc pas ordonnées selon la valeur de `Key` et ce n'est pas efficace de parcourir tous les éléments de ce conteneur par itération. Par contre, ce conteneur est le plus efficace qui soit pour accéder à un élément par sa clé (`Key`). C'est pour cette raison qu'il est employé ici car il sera très utile au TP2. Concernant le conteneur `m_ligne_par_numero`, il sera utile uniquement pour ce TP et servira à afficher les lignes par ordre croissant de numéro de ligne; c'est pour cela que chaque ligne sera stockée à la fois dans `m_lignes` et `m_lignes_par_numero`. Un `multimap<Key, Value>` est comme un `map<Key, Value>` sauf que plusieurs éléments présents dans un `multimap` peuvent avoir la même valeur de clé (`Key`).

L'ajout de toutes les lignes aux conteneurs `m_lignes` s'effectue à l'aide de la méthode `ajouterLignes` dont le prototype est fourni ainsi que les commentaires de spécification. Vous devez implémenter cette méthode. Pour ce faire, vous devez lire une ligne à la fois dans le fichier dont le nom est passé en paramètre. Pour chaque ligne du fichier, **vous devez enlever tous les guillemets présents** et stocker la ligne complète dans un `vector<string>` à l'aide de la méthode privée `string_to_vector` qui est fournie. Ensuite, il faut extraire les composantes pertinentes du `vector<string>` et créer un objet `Ligne` à l'aide de ces composantes pour finalement stocker cet objet dans `m_lignes` et `m_lignes_par_numero`.

L'ajout des stations (présentes dans le fichier `stops.txt`) au conteneur `m_stations` s'effectue à l'aide de la méthode `ajouterStations` dont le prototype est fourni ainsi que les commentaires de spécification. Vous devez implémenter cette méthode de manière analogue à ce que nous venons de décrire pour `ajouterLignes`.

L'ajout des services (de la date `m_date`) dans le conteneur `m_services` s'effectue à l'aide de la méthode `ajouterServices` que vous devez implémenter. Seuls les services avec `exception_type == « 1 »` doivent être ajoutés. Le conteneur de type `unordered_set<Key>` s'utilise de façon analogue au conteneur `set<Key>` sauf qu'un `unordered_set` n'est pas un conteneur ordonné de données; les accès par itération ne sont donc pas efficaces, par contre les accès à l'aide de la clé (`Key`) sont très efficaces.

L'ajout des voyages de la date `m_date` au conteneur `m_voyages` s'effectue à l'aide de la méthode `ajouterVoyagesDeLaDate` que vous devez implémenter. Une fois que cette méthode est exécutée, il vous faut ajouter les arrêts des voyages de la date (`m_date`) qui s'effectue dans l'intervalle `[now1, now2)`. L'ajout de ces arrêts s'effectue à l'aide de la méthode `ajouterArretsDesVoyagesDeLaDate`. Pour chaque arrêt de `m_date` et s'effectuant dans `[now1, now2)`, nous devez créer un objet `Arret` sur le « heap » (à l'aide de `make_shared<Arret>(...)`) et ajouter le `shared pointer` retourné par cette méthode au voyage auquel appartient cet arrêt. Votre code effectuant cela devrait ressembler à

```
Arret::Ptr a_ptr = make_shared<Arret>(station_id, heure_arrivee, heure_depart, numero_sequence, voyage_id);
m_voyages[voyage_id].ajouterArret(a_ptr);
```

(ceci constitue un très gros indice, je ne vous en dit pas plus et ne m'en demandez pas plus...).

Après avoir ajouter tous les arrêts dans les voyages de `m_voyages`, faites le nettoyage : enlevez tous les voyages de `m_voyages` ne contenant aucun arrêt (car ils sont inutiles). Ensuite, pour chaque arrêt dans `m_voyages` ajoutez une copie du `Arret::Ptr` aux arrêts de la station de `m_station` concernée par cet arrêt.



Au final, pour chaque objet Arrêt que vous avez créé dans le « heap », un `shared_ptr` est inséré dans les arrêts du voyage concerné de `m_voyages` et un autre `shared_ptr` (pointant sur le même objet Arrêt) est inséré dans les arrêts de la station concernée de `m_stations`.

Avant de quitter la méthode `ajouterArretsDesVoyagesDeLaDate` vous devez enlever toutes les stations de `m_stations` qui n'ont aucun arrêt, car ces stations sont inutiles pour les besoins en transport de l'intervalle d'intérêt `[now1, now2)`. Finalement, avant de quitter la méthode `ajouterArretsDesVoyagesDeLaDate` vous devez affecter `m_tousLesArretsPresentes` à `true`. Il s'agit de la méthode la plus complexe que vous aurez à implémenter pour ce TP.

Le fichier `transfers.txt` contient toutes les paires de stations (`from_station_id`, `to_station_id`) pour lesquelles un transfert est permis et utile. Notez qu'il est permis de faire un transfert d'une station à elle-même. Cela se produit, par exemple, lorsque deux voyages différents passent par une même station à des temps différents. Il peut alors être utile pour un passager de débarquer à cette station pour embarquer dans le prochain bus qui effectue un voyage différent. Notez cependant que ce ne sont pas toutes les stations qui sont présentes dans `transfer.txt`. En effet, certaines stations, par exemple, desservent uniquement des voyages effectuant le même parcours. De telles stations ne peuvent donc pas être utilisées rationnellement pour faire un transfert.

Vous devrez donc ajouter les transferts (du fichier `transfers.txt`) dans le conteneur `m_transferts` à l'aide de la méthode `ajouterTransferts` que vous devez implémenter. Notez que `m_transferts` est un vecteur de tuple `<from_station_id, to_station_id, min_transfer_time>` ou `min_transfer_time` est le temps **minimal**, en secondes, qu'il faut allouer pour un transfert. **Par contre, lorsque le délai d'attente minimal est de 0 secondes, on vous demande de changer cela pour que le délai d'attente minimal soit de 1 seconde.** Important : ne pas inclure le transfert dans le conteneur `m_transferts` lorsque l'une des deux stations en cause n'est pas présente dans `m_stations` en raison du fait qu'elle ne contenait aucun arrêt dans l'intervalle `[now1, now2)`. Pour cette raison, cette méthode doit être exécutée uniquement après avoir exécuté `ajouterArretsDesVoyagesDeLaDate`. Ceci est contrôlé grâce à l'attribut `m_tousLesArretsPresentes`.

Après avoir ajouté les lignes, stations, services, voyages, arrêts, et transferts dans l'objet `DonnéesGTFS`, votre fonction `main` (qui est fournie!) affichera les données contenues dans cet objet en utilisant les méthodes :

```
void afficherLignes() const;
void afficherStations() const;
void afficherArretsParVoyages() const;
void afficherArretsParStations() const;
void afficherTransferts() const;
```

Ces méthodes sont déjà implémentées pour vous. Par contre, lorsque vous aurez implémenté les méthodes demandées (et qui servent à remplir les conteneurs de `DonnéesGTFS`), vous devrez vous assurer que la fonction `main`, tel que fournie, produit exactement le même résultat que ce qui se trouve dans le fichier `out.txt` fourni avec cet énoncé. Il s'agit de ce que vous devez obtenir pour la date et l'intervalle d'heure que vous trouverez dans le `main.cpp` fourni.

**Important** : notez que vous nous fournissons le fichier `CMakeLists.txt` pour votre projet. Dans ce cas, les exécutables et les bibliothèques statiques produites se trouvent dans le même répertoire que les fichiers sources. Le répertoire `build` (ou le répertoire `cmake-build-debug` lorsque l'on utilise `CLion`) se trouve

également dans ce répertoire. C'est ce CMakeLists.txt qu'utiliseront les correcteurs pour corriger votre TP sur la machine virtuelle du cours. **Vous devez donc vous assurer que votre programme compile (et s'exécute) sans erreur sur la machine virtuelle du cours avec le CMakeLists.txt fourni.**

## 5 Questions ?

Si vous avez des questions pertinentes à propos de ce TP, nous vous demandons de poser de telles questions sur le forum des TPs qui se trouve sur le portail du cours. Comme cela, la réponse sera visible à toute la classe, et non seulement à un seul individu. Toute question envoyée directement au professeur par courriel sera laissée sans réponse afin d'éviter de donner de l'information pertinente à un seul individu. Afin de ne pas encombrer inutilement le forum, essayez de ne pas multiplier vos questions et choisissez-les judicieusement. Finalement, notez que tout TP de cette nature requiert une certaine dose de recherche à effectuer par soi-même afin de trouver la façon d'implémenter ce qui est demandé.

## 6 Fichiers à remettre

Vous devez remettre uniquement les fichiers DonneesGTFS.h et DonneesGTFS.cpp que vous devez insérer dans une archive ayant pour nom NomDeFamille\_Prénom.zip. SVP, ne remettre aucun autre fichier. Vous devez remettre votre travail dans la boîte de dépôt du portail du cours. Aucune remise par courriel ne sera acceptée. Tout travail remis en retard perdra 3 points par heure de retard. Plus spécifiquement, un retard de 1 seconde à 3599 secondes occasionne la perte de 3 points, et après un retard de plus de 33 heures, votre note tombe automatiquement à zéro. Vous pouvez remettre autant de versions que vous le désirez. Seul le dernier dépôt sera corrigé.

**ATTENTION : vous avez la responsabilité de vérifier l'intégrité des fichiers que vous avez remis dans la boîte de dépôt.** Donc, nous vous demandons de vérifier ce que vous avez remis (en téléchargeant sur votre poste ce que vous avez téléversé dans la boîte de dépôt) afin d'en vérifier l'intégrité. Le mécanisme de téléversement du Portail ne corrompt pas les fichiers. Cependant, il peut arriver que votre archive ait été corrompue si, lorsque vous l'avez créée, certains des fichiers étaient ouverts par d'autres applications (Eclipse ou CLion par exemple...). Vérifiez donc l'intégrité de votre archive après l'avoir construite!

**Tout fichier remis qui est inutilisable sera considéré comme un fichier non remis.**

## 7 Critères de correction

- L'exactitude de chaque méthode que vous devez implémenter vaut la grande majorité des points (environ 80%). En gros, vos méthodes sont sensées faire ce qui est demandé.
- La clarté du code des méthodes implémentées vaut le reste des points (environ 20%).
- Une version de DonneesGTFS.h et DonneesGTFS.cpp qui, avec les modules fournis, ne compile pas sur la machine virtuelle du cours avec le CMakeLists.txt fourni se verra attribuée la note de zéro (c'est inacceptable de remettre un programme qui ne compile pas).
- Si le programme résultant de cette compilation plante lors de son exécution. Une note de zéro se verra attribuée au travail (c'est inacceptable de remettre un programme qui plante).

## 8 Plagiat

Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.

**Prenez note qu'il est interdit de publier (sur le Web par exemple) ce TP et sa solution et ce, même après la fin de cette session.**