# TP de rattrapage de C++

## Les graphes dans tous leurs états

N'ayant pas validé votre unité de C++ et Zeus n'étant pas de bonne humeur ce jour là, il vous foudroie sans sommation alors que vous vaquiez à vos occupations. Vous vous retrouvez maintenant dans le purgatoire où des démons vous obligent à vous replonger dans le C++ pour y faire des choses probablement démoniaques.

**Exercice 1 :** ***Un graphe générique***

Les graphes sont une structure de donnée extrèmement utile en informatique, et les démons voudraient en faire plusieurs usage différents. C'est pour cela qu'ils vous demandent de faire des graphes templétés. Ils vous suggèrent de plus d'implémenter ce graphe en utilisant deux tables de hashage. Une première table mappe tous les noeuds du graphe vers un tuple contenant :

— la donnée associée au noeud
— l'ensemble des successeurs du noeud
— l'ensemble des prédécesseurs du noeud

Et la second table mappe toutes les arcs (c'est à dire, une paire de noeud) à la donnée qui lui est associée. Les ensembles seront implémenté par des ensembles hashés.

1°) Créez un fichier `src/graph.hpp` où toutes vos déclarations seront faites dans le namespace `graph`.

2°) Créez une structure nommée `NodeData` avec les paramètres de template suivants :

— typename N est le type d'un noeud du graphe
— typename ND est le type des données associées aux noeuds du graphe
— typename Hash est la structure contenant une fonction de hashage pour N
— typename Equal est la structure contenant une fonction de test d'égalité pour N

Cette structure contient trois champs :

1. ND m_data la donnée associée au noeud
2. std::unordered_set<N, Hash, Equal> m_succs est l'ensemble des successeurs du noeud
3. std::unordered_set<N, Hash, Equal> m_preds est l'ensemble des prédécesseurs du noeud

Définissez un constructeur pour cette classe prenant en paramètre seulement la valeur pour m_data, et initialisant les ensembles m_succs et m_preds à l'ensemble vide.

Rajoutez les méthodes void add_succ(N succ) et void add_pred(N pred) qui rajoute respectivement un successeur et un prédécesseur au noeud.

3°) Comme les arcs sont des pairs de noeud, et qu'il sont utilisé comme clefs dans une table de hashage, il faut définir la fonction de hashage et d'égalité sur ces pairs :

1. Créez une structure templété pair_hash avec les paramètres typename N et typename Hash = std::hash<N> contenant la méthode

```
std::size_t operator()(const std::pair<N, N>& v) const {
  std::size_t v1 = Hash()(v.first);
  std::size_t v2 = Hash()(v.second);
  return v2 + 0x9e3779b9 + (v1<<6) + (v1>>2);
}
```

2. Créez une structure templété pair_eq avec les paramètres typename N et typename Equal = std::equal_to<N> contenant la méthode

```
bool operator()(const std::pair<N, N>& lhs, const std::pair<N, N>& rhs) const {
  return Equal()(lhs.first, rhs.first) and Equal()(lhs.second, rhs.second);
}
```

4°) Créez une structure nommée `Graph` avec les paramètres de template suivants :
- typename N est le type d'un noeud du graphe
- typename ND est le type des données associées aux noeuds du graphe
- typename ED est le type des données associées aux arcs du graphe
- typename Hash = std::hash<N> est la structure contenant une fonction de hashage pour N
- typename Equal = std::equal_to<N> est la structure contenant une fonction de test d'égalité pour N

Cette structure contient deux champs :

1. std :: unordered_map<N, NodeData<N, ND, Hash, Equal>, Hash, NEqual> m_nodes qui donne pour chaque noeud du graphe le NodeData qui lui est associé
2. std :: unordered_map<std::pair<N, N>, ED, pair_hash<N, Hash>, pair_eq<N, Equal>> m_edges qui donne pour chaque arc du graphe la valeur qui lui est associé

Définissez un constructeur pour cette classe. Ce constructeur ne prend aucun paramètre et initialize les deux table de hashage à l'ensemble vide.

Rajoutez les méthodes void add_node(N n, ND d) et void add_edge(N pred, N succ, ED data) qui rajoute respectivement un noeud et un arc au graphe.

> **Note :** *Faites attention que les successeurs et les prédécesseurs des noeuds sont bien mis à jour.*

**Exercice 2** : ***Un premier test***

Vos tortionnaires démoniaques veulent tester votre première implémentation du graphe sur un projet horrible, nommé `pilelivre`. Ce projet consiste à permettre à des humains naïfs de stocker toute informations les concernants dans un graphe accessible aux démons. Ce graphe sera donc un graphe de données, et pour les identifier, vous utiliserez donc des chaînes de caractères. Dans ce premier tests, seuls quelques formats de données sont disponibles : des personnes, des messages publics, des messages privés et des vidéos de chat.

1°) Créez un fichier `src/pilelivre.hpp` où toutes vos déclarations seront faites dans le namespace `pilelivre`.

2°) Créez une hiérarchie de structure pour les données du graphe :
- Data est la classe mère de toute les données. Cette classe ne contient ni champ ni méthode.
- Person est le type des données associées aux personnes et contient les champs et méthodes :
  - std :: string m_name est le nom de la personne
  - std :: list <Action *> m_action est la liste des actions que cette personne a fait
  - std :: list <Person *> m_friends est la liste des amis de cette personne
  - void add_action(Action *) rajoute une action à la personne

  Cette structure a un seul constructeur prennant en paramètre le nom de la personne.
- Action est le type des actions effectuée par une personne. Cette classe ne contient ni champ ni méthode
- MsgPublic est une action et contient les champs et méthodes :
  - std :: string m_msg est le contenu du message

  Cette structure a un seul constructeur prennant en paramètre le contenu du message.
- MsgPrivate est une action et contient les champs et méthodes :
  - std :: string m_msg est le contenu du message
  - Person * m_dest est la personne à qui le message a été envoyé

  Cette structure a un seul constructeur prennant en paramètre les valeurs de ses deux champs.
- CatPropaganda est une action et contient les champs et méthodes :
  - std :: string m_url est le lien vers la video de chat

  Cette structure a un seul constructeur prennant en paramètre le lien vers la vidéo.
- FriendAdd est une action et contient les champs et méthodes :
  - Person * m_person est le nouvel ami

Cette structure a un seul constructeur prennant en paramètre le nouvel ami.

— FriendRemove est une action et contient les champs et méthodes :
  — Person * m_person est l'ancien ami

  Cette structure a un seul constructeur prennant en paramètre l'ancien ami.

Implémentez dans le fichier `src/pilelivre.cpp` les différents constructeurs et méthodes de ces classes

> **Note :** *Faites attention que rajouter une action* FriendAdd *ou* FriendRemove *n'agit pas seulement sur la liste des actions de la personne.*

3°) Déclarez la structure PileLivre_v1 avec les cinq champs suivants :

1. graph::Graph<std::string, Data *, int> m_content est le graphe des données
2. std :: list <Data *> m_data contient toutes les données du graphe : cette liste est utilisée pour s'assurer que les données sont bien toutes désallouées une unique fois quand l'objet PileLivre_v1 est détruit.
3. std :: size_t m_counter_msg est un compteur pour les messages
4. std :: size_t m_counter_cat est un compteur pour les vidéos de chats
5. std :: size_t m_counter_friend est un compteur pour les actions concernant les amis

> **Note :** *Vu l'implémentation du graphe, nous sommes obligé de donner un type aux valeurs des arcs, même si dans ce test ces valeurs ne sont pas utilisées. Par défaut nous utilisons le type* bool*, et toutes les création d'arc dans ce test donneront* true *comme valeur pour l'arc. Ce soucis étant maintenant bien identifié, nous le résoudrons dans une prochaine étape.*

Cette structure contient un unique constructeur sans paramètre initialisant les deux premiers champs de la classe au vide, et les autres à 0. Elle contient aussi les méthodes suivantes :

— void add_person(std::string name) rajoute une personne dans le graphe
— void add_msg_public(std::string name, std::string msg) rajoute l'action que la personne name a publié un nouveau message public contenant msg
— void add_msg_private(std::string writer, std :: string reader, std :: string msg) rajoute l'action que la personne writer a publié un nouveau message privé, pour reader et contenant msg
— void add_cat(std::string name, std::string url) rajoute l'action que la personne name a publié un nouveau message public contenant msg
— void add_friend(std :: string name, std::string friend) rajoute l'action que la personne name a un nouvel ami friend
— void rm_friend(std :: string name, std::string friend_nomore) rajoute l'action que la personne name a perdu l'ami friend_nomore

Implémentez ces différentes méthode sachant que :

— l'identifiant d'une personne est son nom
— l'identifiant d'un message est de la forme "action/msg/" << this->counter_msg
— l'identifiant d'une vidéo de chat est de la forme "action/cat/" << this->counter_cat
— l'identifiant d'une action sur les amis est de la forme "action/friend/" << this->counter_friend

> **Note :** *N'oubliez pas d'incrémenter un compteur après l'avoir utilisé, afin de s'assurer que les prochaines actions auront bien un identifiant différent.*

4°) Dans le fichier `src/test_v1.cpp` créez une fonction pilelivre ::PileLivre_v1 first_day () qui :

— crée un objet pilelivre ::PileLivre_v1 res
— lui rajoute un personnage nommé James
— fait que James publie un message publique contenant "Yoh, Salut PileLivre !!!"
— lui rajoute un personnage nommé Jack
— fait que James rajoute Jack comme ami
— fait que Jack publie un message privé à James disant : "t'es qui???"

— fait que Jack défasse son amitié naissante avec James
— et finalement, renvoie l'objet res

> **Note :** *N'oubliez pas de tester votre test, et de vérifier qu'à la destruction de l'objet* res*, tous les objets* Data *qu'il contenait sont bien désalloués.*

**Exercice 3** : ***Le parcours du graphe***

Une fonctionnalité assez centrale d'un graphe est de pouvoir le parcourir.

1°) Dans le fichier `src/graph.hpp`, rajoutez à la structure Graph une méthode bool has_node(N node) qui renvoie si le paramètre node de la méthode est bien un noeud du graphe.

> **Note :** *Comme nous sommes en C++17 et non en C++20, la méthode* contains *du template* unordered_map *nous est indisponible : la méthode* has_node *doit donc être implémentée en utilisant la méthode* unordered_map::find.

De plus, implémentez l'opérateur ND operator[](N node) qui renvoie la valeur associée à un noeud, ainsi que l'opérateur ED operator[](std :: pair<N, N> edge) qui renvoie la valeur associée à un arc.

2°) Il nous reste à créer des itérateurs pour accéder facilement aux précédesseurs et successeurs d'un noeud. Ces itérateurs sont mis en place par deux templates, à implémenter entre les structures NodeData et Graph :

1. le template template<typename N> struct_next contient :
   — Un champ Graph const & g
   — un champ N const prev
   — un constructeur prenant en paramètre la valeur de ses deux champs
   — une méthode std :: unordered_set:: iterator begin() qui renvoie l'itérateur begin() sur les successeurs de son champ prev, ou std :: unordered_set:: iterator () si prev n'est pas un noeud du graphe
   — une méthode std :: unordered_set::const_iterator begin() const qui renvoie l'itérateur begin() sur les successeurs de son champ prev, ou std :: unordered_set::const_iterator () si prev n'est pas un noeud du graphe
   — une méthode std :: unordered_set:: iterator end() qui renvoie l'itérateur end() sur les successeurs de son champ prev, ou std :: unordered_set:: iterator () si prev n'est pas un noeud du graphe
   — une méthode std :: unordered_set::const_iterator end() const qui renvoie l'itérateur end() sur les successeurs de son champ prev, ou std :: unordered_set::const_iterator () si prev n'est pas un noeud du graphe
2. le template template<typename N> struct_pred contient :
   — Un champ Graph const & g
   — un champ N const next
   — un constructeur prenant en paramètre la valeur de ses deux champs
   — une méthode std :: unordered_set:: iterator begin() qui renvoie l'itérateur begin() sur les prédecesseurs de son champ next, ou std :: unordered_set:: iterator () si next n'est pas un noeud du graphe
   — une méthode std :: unordered_set::const_iterator begin() const qui renvoie l'itérateur begin() sur les prédecesseurs de son champ next, ou std :: unordered_set::const_iterator () si next n'est pas un noeud du graphe
   — une méthode std :: unordered_set:: iterator end() qui renvoie l'itérateur end() sur les prédecesseurs de son champ next, ou std :: unordered_set:: iterator () si next n'est pas un noeud du graphe
   — une méthode std :: unordered_set::const_iterator end() const qui renvoie l'itérateur end() sur les prédecesseurs de son champ next, ou std :: unordered_set::const_iterator () si next n'est pas un noeud du graphe

3°) Dans la structure Graph, rajoutez les méthodes
- struct_next<N> next(N node)
- struct_next<N> const next(N node) const
- struct_pred<N> pred(N node)
- struct_pred<N> const pred(N node) const

> **Note :** *N'oubliez pas de vérifier que les méthodes que vous venez d'implémenter sont correctes.*

**Exercice 4 :** ***La spécialisation du graphe à*** void

Dans le test précédent, nous avons vu que bien que les arcs n'ont pas de valeur associés, nous devons leur en donner car l'implémentation de graph::Graph la stocke quelque part. De plus, vos démons vous indique que votre prochaine tache utilisera des graphes où les noeuds n'ont pas de données associées. Que faire???

En C++, il est possible de changer l'implémentation d'un template en fonction de ses paramètres : pour cela, il suffit de redéclarer le template partiellement instancié après sa première déclaration.

1°) Reprennez votre fichier `src/graph.hpp` et à la suite de l'implémentation du graphe générique, rajoutez la spécialisation de la classe `NodeData` pour quand typename ND est void :

```
template<typename N, typename Hash, typename Equal>
struct NodeData<N, void, hash, Equal> {
  NodeData(): m_succs(), m_preds() {}
  std::unordered_set<N, Hash, Equal> m_succs;
  std::unordered_set<N, Hash, Equal> m_preds;
  void add_succ(N succ) { ... }
  void add_pred(N pred) { ... }
};
```

où vous complétez l'implémentation des méthodes add_succ et add_pred.

Dans ce code, vous pouvez voir comment on spécialise une classe templétée : on met dans l'entrée template la liste des paramètres qui restent, puis on redéclare la structure (ici `NodeData`) avec les paramètres correspondants (ici, on peut voir qu'on a mis void pour le paramètre ND, les autres paramètres restant inchangés).

2°) suite à cette spécialisation de `NodeData` écrivez la spécialisation de Graph pour quand ND est void. Notez que la méthode de rajout de noeud devient void add_node(N n), et que l'opérateur ND operator[](N node) disparait

3°) suite à cette spécialisation de `NodeData` et de Graph pour quand ND est void, rajoutez la spécialisation de Graph pour quand ND n'est pas void mais ED l'est. En particulier, cette spécialisation ne contient pas le champ m_edges ni l'opérateur ED operator[](std::pair<N, N> edge).

**Exercice 5 :** ***Un premier test : deuxième tentative***

1°) Dans le fichier `src/pilelivre.hpp`, déclarez la structure PileLivre_v2 de façon quasiment identique à PileLivre_v1, sauf que le champ graph::Graph<std::string, Data*, int> m_content est remplacé par graph::Graph<std::string, Data*, void> m_content. Adaptez toutes les méthodes de PileLivre_v1 à PileLivre_v2.

2°) Dans le fichier `src/test_v2.cpp` créez une fonction pilelivre::PileLivre first_day() identique à la fonction pilelivre::PileLivre_v1 first_day() du fichier `src/test_v1.cpp`, sauf qu'elle utilise un pilelivre::PileLivre_v2 au lieu d'un pilelivre::PileLivre_v1.

**Exercice 6 :** ***Un second test***

Les automates finis sont une structure très classique pour la reconnaissance d'expressions régulières.

1°) Créez un fichier `src/automata.hpp` où toutes vos déclarations seront faites dans le namespace `automata`.

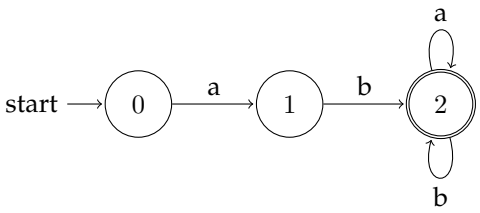
2°) Dans ce fichier, déclarez la structure Automata contenant :
- Un champ graph::Graph<std::size_t, void, char> g
- un champ std :: size_t counter_curr (qui sert pour rajouter des nouveaux noeuds à g)
- un champ std :: unordered_set<std::size_t> terminal (qui stoque les noeuds terminaux de l'automate)
- un constructeur sans paramètre, initialisant : g à un graphe contenant un unique noeud 0 (le neoud de départ de l'automate) ; counter_curr à 1 ; et terminal à l'ensemble vide
- une méthode std :: size_t new_node() qui
  - rajoute un nouveau noeud counter_curr à g
  - incrémente counter_curr
  - renvoie le nouveau noeud créé
- void new_arc(std::size_t pred, std :: size_t succ, char c) qui
  - vérifie que pred et succ sont bien des noeuds du graphe (on peut utiliser le fait que le graphe contient toutes les valeurs strictement inférieures à counter_curr)
  - vérifie qu'il n'existe pas d'arc déjà existant de pred vers succ
  - rajoute l'arc de pred vers succ annoté par c
- une méthode void new_terminal(std::size_t node) qui
  - vérifie que node est bien un noeud du graphe
  - rajoute node à l'ensemble des noeuds terminaux

Implémentez dans le fichier `src/automata.cpp` les différents constructeurs et méthodes de cette structure.

3°) Rajoutez à la structure Automata la méthode bool parse(std :: string word) qui vérifie que le mot en paramètre est bien reconnu par l'automate. L'algorithme pour cette vérification est comme suit

```
state = 0
for every character c of word {
 if state has a successor n of such that the edge (state,n) is tagged with c {
   state = n
 } else { return false }
}
return true
```

4°) Dans le fichier `src/test_v3.cpp` créez une fonction automata::Automata start_ab() qui crée l'automate suivant (le double cercle pour le noeud 2 signifie qu'il est terminal).




Créez une fonction void start_ab_check() qui vérifie (en utilisant les assert de #include <cassert>) que cet automate reconnait bien le mot "abba" mais pas les mots "baba" ni "abcba".