1. Carte (2) : Éléments

Héritage, Constructeurs, Liste d'initialisation, Appel à la super méthode.

Les classes demandées ici utilisent les classes `taille` et `position` définies dans la première partie de cet exercice, récupérez donc les fichiers .hh/.cc de ces classes et incorporez-les dans le projet de cet exercice. On veut développer quelques classes qui pourraient servir de base au développement d'un jeu dans lequel des personnages se déplacent sur une carte, la carte pouvant contenir des obstacles qui empêchent le déplacement ainsi que des objets. On supposera qu'il s'agit d'une carte en 2D, sans aucune notion d'altitude, et que chaque objet sera représenté par un rectangle dont les côtés sont parallèles aux axes.

1. Définir la classe `element` comme la racine de tous les éléments figurant sur la carte. Les attributs et méthodes de cette classe seront donc ceux qui sont hérités par tous les éléments. Un élément a une `position`, une `taille`.
   Écrire aussi le/les constructeur(s). Quand on définit un/des constructeur(s) d'une classe, il faut toujours réfléchir à des constructeurs qui ont un sens afin que les instances créées soient correctement initialisées. Ici, un élément doit obligatoirement avoir une taille et une position.
   Écrire les accesseurs à la taille et à la position.
2. Écrire une méthode `tostring` retournant une chaîne de caractères représentant l'élément (taille et position). La méthode retournera une chaîne, elle ne fera pas d'affichage. Vous pouvez écrire une méthode `tostring` dans `taille` et `position` et appeler ces méthodes depuis `element::tostring`. Voir l'explication à la fin de cet exercice pour la conversion d'un entier en chaîne.
3. Écrire une méthode `contientposition` prenant comme paramètre une `position` et retournant vrai si l'élément contient la `position` (c'est-à-dire que le point représenté par la `position` passée est à l'intérieur du rectangle repéré par la `position` et la `taille` de l'`element`), faux sinon.
4. Sur la carte, nous aurons 3 types d'`element` : les `obstacle`, les `personnage` (qui peuvent se déplacer) et les `objetramassable` (qui peuvent être ramassés par un `personnage`). Chaque `objetramassable` dispose d'un nombre de points (qui seront attribués au `personnage` qui le ramasse). Chaque `personnage` dispose donc d'un nombre de points acquis (en début de partie : 0) qui est mis à jour à chaque fois qu'il ramasse un objet. Chaque `personnage` a, de plus, un nom sous la forme d'une chaîne de caractères.
   Écrire ces 3 classes avec constructeurs et accesseurs correspondants. Au moment de la création des instances de ces 3 classes, une `position` et une `taille` doivent obligatoirement être précisées.
5. Modifier le comportement de `tostring` sur les `objetramassable` et `personnage` afin qu'elle retourne, en plus de la taille et la position, les informations supplémentaires de ces classes (nombre de points, nom).
6. Écrire une méthode `ramasser` de `personnage` prenant comme paramètre un `objetramassable` et ajoutant au personnage le nombre de points correspondant à l'objet ramassé.

Conversion d'un entier en chaîne de caractères.

`std::to_string` http://en.cppreference.com/w/cpp/string/basic_string/to_string permet de convertir un entier ou un réel en `std::string`.
Il existe une méthode pouvant être utilisée sur n'importe quel type disposant d'un opérateur de sortie sur un flux (en d'autres termes, tous les types pour lesquels `std::cout << x` fonctionne) : La classe `std::ostringstream` fournit un flux de sortie (et peut donc s'utiliser avec << pour lui « envoyer » tout type de données) qui ne fait aucun affichage mais mémorise son contenu et le rend disponible sous forme d'une string à l'aide de la méthode `str()`.

```
#include <sstream>
#include <string>
#include <iostream>

int main() {
    std::ostringstream out;
    int entier(42);
    out << entier; // fonctionne avec tout type pouvant être envoyé sur flux
```

```
    std::string s(out.str());
    std::cout << s << std::endl;
    return 0;
}
```

2. Figures géométriques

Héritage, Exécution des constructeurs/destructeurs, Constructeur par recopie.

On veut manipuler des figures géométriques dans le plan, par exemple pour stocker le contenu d'une image vectorielle. Il existe des figures de plusieurs types : segment, point, triangle, rectangle (dont les côtés sont parallèles aux axes), texte. Testez vos classes au fur et à mesure, en écrivant un `main` *qui appelle vos méthodes.*

1. Définir un type `couleur` composé des trois composantes rouge, vert, bleu, chacune de ces composantes étant un entier court non signé.
2. Définir un type `point` représentant les coordonnées d'un point dans le plan, et munir cette classe des méthodes nécessaires pour la suite de l'exercice :
   Définir un constructeur par défaut construisant le point (0,0), un constructeur prenant comme paramètre deux coordonnées, et un constructeur par recopie (qui n'aurait pas besoin d'être écrit ici, le constructeur par recopie implicite faisant exactement ce que l'on attend). Les constructeurs et destructeur de cette classe afficheront un message (« constructeur par défaut », « constructeur par recopie », « destructeur ») sur le flux de sortie standard, permettant de visualiser l'appel à ces méthodes. À chaque fois que vous exécuterez le programme **essayez de comprendre** les affichages effectués par ces constructeurs / destructeurs. Si les affichages ne correspondent pas à ce que vous attendez (par exemple : « trop » de constructeurs sont appelés) **essayez de comprendre pourquoi, et modifiez votre code** afin que seuls soient créés les objets requis.
3. *Chaque figure a une couleur et un point d'origine. Dans le cas d'un segment, il y a aussi l'extrémité du segment qui doit être mémorisée. Dans le cas d'un texte, le texte affiché doit évidemment être stocké. Dans le cas d'un triangle, il y a évidemment trois points au total à mémoriser. Il est rappelé que seuls les rectangles dont les côtés sont parallèles aux axes doivent être pris en compte.* Déclarer et définir une classe `figuregeometrique` ainsi que des sous-classes pour représenter les différentes figures géométriques. Là aussi, les constructeurs et destructeurs afficheront un message sur le flux standard, et vous ferez les mêmes vérifications que pour `point`.
4. Toutes les figures géométriques doivent être capables de s'afficher à l'écran, sous forme textuelle. Cet affichage aura la forme suivante : type de la figure (point, triangle, etc.), point d'origine, couleur, autres points (s'il y en a), texte (dans le cas d'un texte). Le type de la figure affiché sera simplement une chaîne de caractères parmi *Point*, *Triangle*, *Texte*, etc. Réfléchir à la façon de factoriser au mieux le code entre la super-classe et les sous-classes.
5. Écrire une méthode `memeorigine` permettant de tester si une figure a le même point d'origine qu'une autre figure. Vérifier dans le `main` que cette méthode peut être appliquée (par exemple) entre un texte et un rectangle.
6. Écrire une méthode `translation` permettant d'appliquer une translation, définie par un vecteur (donc un `point`) sur une figure géométrique.
7. Écrire une méthode `estcarre` permettant de tester si un rectangle est un carré.

3. Une classe ensemble

Classe abstraite, Parcours d'une structure de données.

Dans cet exercice, on veut écrire une classe `ensemble` *permettant de stocker un ensemble d'entiers, mais on veut pouvoir tester plusieurs implémentations de cette classe (toujours en mémorisant des entiers, mais de différentes façons : tableau, vector, liste, etc), en modifiant le moins possible le code utilisant la classe* `ensemble`*. Pour cela, on écrira une classe* `ensemble` *abstraite, qui déclare toutes les méthodes et une première sous-classe* `ensembletableau` *dans laquelle les entiers contenus dans l'ensemble seront stockés dans un tableau alloué dynamiquement et redimensionné au besoin (à la façon de la* `sequence` *dans le TP2 exercice 2), sans chercher à faire une implémentation efficace (vous pouvez « redimensionner » le tableau à chaque ajout), et une deuxième sous-classe* `ensemblevector` *dans laquelle les entiers contenus dans l'ensemble seront mémorisés dans un* `std::vector`*. Définir les*

méthodes dans la classe et dans chacune des deux sous-classes permettant de :

1. Créer un ensemble vide.
2. Créer un ensemble par copie d'un ensemble existant.
3. Écrire une fonction de test `testensemble` dans le même fichier que celui contenant le `main`, cette fonction prendra comme paramètres (au moins) deux `ensemble` et sera complétée au fur et à mesure que vous ferez les questions suivantes afin de tester les fonctionnalités demandées. Dans le `main`, qui sera très simple, vous instancierez des `ensembletableau` et des `ensemblevector` que vous passerez à `testensemble` et vous pourrez ainsi vérifier au fur et à mesure que vos deux implémentations d'`ensemble` se comportent de la même manière et sont donc interchangeables.
4. Tester si un entier appartient à l'ensemble.
5. Tester si un ensemble est vide.
6. Ajouter un entier à un ensemble (si l'entier appartient déjà à l'ensemble, la méthode ne fait rien).
7. Afficher le contenu d'un ensemble.
8. Enlever un élément d'un ensemble.
9. Parcourir un ensemble. Attention, cela mérite réflexion : prévoir la possibilité de parcourir un ensemble sans « montrer » comment est implémenté l'ensemble, c'est-à-dire, prévoir la possibilité de parcourir un ensemble qui sera exactement la même du point de vue de l'utilisateur d'`ensemble`, quelle que soit la sous-classe d'`ensemble` utilisée (puisque les méthodes devront être déclarées dans la classe abstraite). Il ne s'agit pas ici d'écrire une méthode faisant (par exemple) l'affichage du contenu de l'`ensemble`, mais il s'agit de fournir une/des méthode(s) permettant, depuis une autre classe ou le `main`, d'avoir accès à tous les éléments de l'ensemble, par exemple pour les afficher, rechercher une valeur, etc. Si vous ne voyez pas comment faire, réfléchissez à la façon dont vous parcourez un tableau (en utilisant un indice (entier), initialisé à zéro, incrémenté à chaque itération, jusqu'à ce qu'il atteigne la taille du tableau) ou le fonctionnement des `iterator` de `std::vector`, et inspirez-vous de cela pour proposer une solution.

Les questions suivantes devront être traitées après la question précédente, car la possibilité de parcourir un ensemble permet d'écrire « facilement » les méthodes des questions suivantes, alors que sans le parcours, il sera difficile d'écrire un code correct.

10. Réaliser l'union de deux ensembles.
11. Réaliser l'intersection de deux ensembles.
12. Réaliser la différence entre deux ensembles (le résultat contient les valeurs qui sont dans un premier ensemble et qui ne sont pas dans un deuxième ensemble).

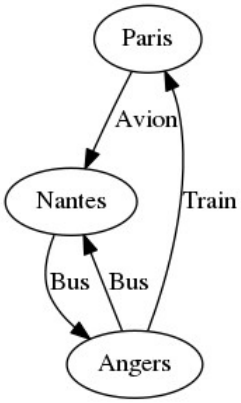
4. Utilisation de `std::list`

std::list, Parcours, Attribut de classe (chapitre 4 du cours), struct.

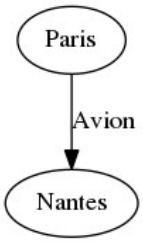
`std::list` fournit le type de données liste (simplement) chaînée. http://en.cppreference.com/w/cpp/container/list . Par rapport à `std::vector`, les avantages d'une liste chaînée sont : l'absence de « redimensionnements » coûteux, la possibilité d'insérer / supprimer des éléments de façon efficace à n'importe quelle position (alors que seules les insertions / suppressions en fin sont efficaces dans un `std::vector`). Le principal inconvénient est l'impossibilité d'accéder à un élément donné de façon efficace : il faut parcourir la liste depuis le début (accès en temps linéaire vs. accès en temps constant). Pour cette raison, il n'y a pas de méthode permettant d'accéder à un élément par son indice (l'opérateur `[]` ou la méthode `at` de `std::vector`). Il conviendra donc de choisir le conteneur le plus adapté en fonction des besoins.

Dans cet exercice, on va écrire une classe permettant de manipuler des graphes orientés, et disposant d'une fonction d'export du graphe au format dot (pour plus d'informations sur ce format : https://graphviz.gitlab.io/_pages/doc/info/lang.html mais il n'est pas indispensable de consulter cette documentation) qui est un format textuel permettant de représenter des graphes et pour lequel l'outil graphviz (http://www.graphviz.org/) fournit un ensemble d'outils permettant de visualiser ces graphes. À la fin de l'exercice, vos classes seront capables de générer un fichier au format dot qui pourra ensuite être converti en image au format png telle que l'image ci-dessous. La dernière question de l'exercice contient un `main` qui génère le graphe ci-dessous.

1. Écrire une classe `sommet` représentant un sommet d'un graphe et ses arcs sortants. Cette classe comportera comme attributs :
   un identifiant du sommet (de type `identifiant`, équivalent à `unsigned int`) : cet identifiant unique sera attribué automatiquement à la construction (sans être passé au constructeur) ;
   une étiquette de sommet (une chaîne de caractères) ;
   une `std::list` d'arcs sortants. Pour cela, on déclarera un type structuré `arcsortant` composé de l'étiquette de l'arc (une chaîne) et l'extrémité de l'arc (un `identifiant`).



2. Munir la classe des méthodes suivantes :
   Constructeur prenant comme paramètre une étiquette de sommet (et construisant un sommet n'ayant pas d'arc sortant).
   `id` et `etiquette` deux accesseurs à l'identifiant et à l'étiquette.
   `ajouterarc` prenant comme paramètre un identifiant de sommet extrémité et une étiquette d'arc et rajoutant l'arc sortant au sommet. S'il existe déjà un arc sortant vers le même sommet, l'étiquette de celui-ci sera remplacée par l'étiquette passée en paramètre (il ne peut y avoir deux arcs entre deux sommets donnés).
   `supprimerarc` prenant comme paramètre un identifiant de sommet et supprimant l'arc sortant vers ce sommet (s'il existe, ne faisant rien sinon).
   `supprimertousarcs` supprimant tous les arcs sortants d'un sommet.
   `dot_etiquette` prenant comme paramètre un flux de sortie et envoyant sur ce flux une construction du type « identifiant `[label="etiquette"] ;` » en préfixant l'identifiant par un « c ». Par exemple pour un sommet dont l'identifiant est 1 et l'étiquette est « a », cette méthode sortira « `c1 [label="a"];` ».
   `dot_arcssortants` prenant comme paramètre un flux de sortie et envoyant sur ce flux de sortie, pour chacun des arcs sortants, une construction du type « identifiant du sommet `->` identifiant du sommet `;` ». Par exemple, pour un sommet dont l'identifiant est 1 et ayant deux arcs vers les sommets 2 et 3, cette méthode sortira sur le flux « `c1 -> c2 ; c1 -> c3 ;` ». Si l'étiquette de l'arc n'est pas la chaîne vide, on sortira cette étiquette de la même façon que pour un sommet, c'est-à-dire « `[label="etiquette"]` » juste avant le « ; ». Voir l'exemple en bas de la question suivante, pour l'arc allant de `c1` à `c3` et étiqueté « Avion ».
3. Écrire une classe `graphe` contenant une liste de `sommet`. Munir cette classe des méthodes suivantes :
   `ajoutersommet` prenant comme paramètre une étiquette de sommet et retournant l'identifiant du sommet ajouté.
   `ajouterarc` prenant comme paramètre deux identifiants de sommets et une étiquette d'arc et ajoutant un arc entre les deux sommets, étiqueté par l'étiquette passée. La méthode retournera `true` si l'ajout s'est correctement déroulé, et `false` s'il n'a pu être fait (identifiant de sommets introuvables).
   `supprimersommet` prenant un identifiant et supprimant le sommet (ainsi que tous les arcs sortants et **entrants** de ce sommet).
   `dot_sortie` prenant comme paramètre un flux de sortie et sortant sur ce flux une expression du type « `digraph G {` *les sorties de `dot_etiquette` pour chacun des sommets, les sorties de `dot_arcssortants` pour chacun des sommets* `}` ». Par exemple, ci-dessous, un graphe et le fichier au format dot associé.




```
Digraph G {
c1 [label="Paris"];
c1 -> c3 [label="Avion"];
c3 [label="Nantes"];
}
```

4. Le fichier source ci-dessous génère le graphe de la figure par des appels aux méthodes demandées ci-dessus et l'enregistre dans un fichier `tout.dot` (qui est généré dans le répertoire courant, c'est-à-dire le répertoire dans lequel a été construit l'exécutable, qui doit être dans `/tmp`) (nous verrons un peu plus tard les fichiers plus en détail). La commande dot offre différentes opérations sur des fichiers au format dot, dont la conversion en png. Ouvrir un terminal et aller dans le répertoire dans lequel a été généré le fichier `tout.dot` et exécuter la commande
   `dot -T png tout.dot -o tout.png`
   Le fichier png ainsi obtenu peut être ouvert par `xdg-open tout.png` qui ouvre l'application associée

par défaut au format png, vérifier que les fichiers `tout.png` et `sansangers.png` (à générer à partir de `sansangers.dot`) ainsi construits sont les mêmes que ceux présents dans ce sujet.
http://www.info.univ-angers.fr/pub/genest/fichiers/l3_poo/tp/3/maingraphe.cc

```
#include "graphe.hh"
#include <fstream>

void sauvergraphe(graphe const & g, std::string const & nomfichier) {
    std::ofstream f(nomfichier);
    g.dot_sortie(f);
}

int main() {
    graphe g;
    auto paris(g.ajoutersommet("Paris"));
    auto angers(g.ajoutersommet("Angers"));
    auto nantes(g.ajoutersommet("Nantes"));
    g.ajouterarc(angers, nantes, "Bus");
    g.ajouterarc(nantes, angers, "Bus");
    g.ajouterarc(angers, paris, "Train");
    g.ajouterarc(paris, nantes, "Avion");
    sauvergraphe(g, "tout.dot");
    g.supprimersommet(angers);
    sauvergraphe(g, "sansangers.dot");
    return 0;
}
```

5. Carte (3) : Listes

std::list : parcours, ajout, suppression.

Cet exercice est la suite de la deuxième partie de cette série et utilise les classes définies précédemment. Cet exercice ne requiert pas d'utiliser le polymorphisme (que nous aurons l'occasion d'utiliser dans les exercices suivants).

1. Une `carte` est une zone rectangulaire dans laquelle l'origine des coordonnées est le point (0,0) situé dans le coin supérieur gauche. Une `carte` a donc une `taille` (mais pas de `position`). Une `carte` contient un ensemble de `personnage` (représenté par une `std::list`), un ensemble de `obstacle` (représenté par une `std::list`) et un ensemble de `objetramassable` (représenté par une `std::list`). L'ordre des éléments dans les différentes listes n'a aucune importance.
2. Écrire un constructeur, construisant une `carte` « vide » (c'est-à-dire sans aucun élément).
3. Écrire une (ou des) méthode(s) de `carte` permettant d'ajouter à une carte un `obstacle`, un `objetramassable` et un `personnage`.
4. Écrire une méthode d'affichage d'une `carte` qui affichera à l'écran la taille de la carte ainsi que les positions et tailles de tous les `personnage` qui la composent.
5. Écrire une méthode permettant de tester si deux `element` ont une intersection commune sur la carte. Pour faciliter l'écriture de cette méthode, commencer par écrire une méthode qui prend comme paramètre deux intervalles de `coordonnee` et qui retourne vrai si et seulement si ces deux intervalles ont une intersection non vide (si vous n'arrivez pas à écrire cela simplement, essayez de prendre le problème à l'envers et essayez d'écrire une méthode qui cherche si deux intervalles n'ont aucune valeur commune). Cette méthode travaillant sur les intervalles sera ensuite utilisée pour calculer l'intersection d'`elements` (c'est-à-dire de rectangles).
6. Écrire dans `carte` une méthode prenant comme paramètre un `personnage` et un déplacement (vers le nord, le sud, l'est, l'ouest... réfléchir à la meilleure façon de représenter un déplacement) et retournant `true` si le déplacement est possible et `false` sinon. Un déplacement correspond à un changement de position d'une unité dans la direction correspondante. Le déplacement est possible si la nouvelle position du `personnage` n'est occupée par aucun autre `personnage`, aucun `objetramassable`, aucun `obstacle` et s'il ne sort pas de la carte.
7. Écrire dans `carte` une méthode effectuant le déplacement d'un `personnage` dans une direction passée en paramètre (c'est-à-dire modifiant sa position, si le déplacement est possible).

8. Écrire dans `carte` une méthode prenant comme paramètre un `personnage` et un `objetramassable` et retournant `true` si le personnage peut ramasser l'objet et `false` sinon. Un personnage peut ramasser un objet s'il « touche » cet objet.
9. Écrire dans `carte` une méthode prenant comme paramètre un `personnage` et ayant pour effet de faire ramasser à ce personnage tous les `objetramassable` qui sont à sa portée. Les objets ramassés sont supprimés de la carte.