1. Carte (2/3) : 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 (`accestaille`) et à la position (`accesposition`).
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) ainsi que le type d'élément : par exemple pour un personnage, la méthode retournera une chaîne du type `per (x,y)[l,h] 12pts nom` ou, pour un obstacle `obs (x,y)[l,h]`.
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`. C'est la façon le plus simple de convertir un type numérique en chaîne.

Il existe autre méthode, plus générale, car elle peut 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` (sous-classe de `std::ostream`) 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()`. Le principal intérêt, c'est qu'il est possible d'enchaîner les sorties sur une instance de `std::ostringstream` (éventuellement à partir de types différents), puis, en une opération (`str`), obtenir la chaîne correspondant à tout ce qui a été envoyé dans le flux. Voir l'exemple ci-dessous et la documentation sur la page https://en.cppreference.com/w/cpp/io/basic_ostringstream

```
#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, Méthode virtuelle.

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, triangle, rectangle dont les côtés sont parallèles aux axes. 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 (entières) 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, 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) et un destructeur (qui n'aurait pas besoin d'être écrit ici). 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 (en traitant les questions suivantes) **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 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 de sortie 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, à l'aide d'une méthode `afficher`. Cet affichage aura la forme suivante : couleur, point d'origine, autres points (s'il y en a).
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 `estcarre1` permettant de tester si un **rectangle** est un carré (cette méthode sera définie uniquement sur les instances de rectangle). Écrire une méthode `estcarre2` permettant de tester si une figure géométrique quelconque est un carré (cette méthode sera définie pour toutes les figures) .
8. Écrire une méthode `typefigure` qui retournera une chaîne qui vaudra `fig` pour une `figuregeometrique`, `seg` pour un segment, `tri` pour un triangle, `rec` pour un rectangle. Modifier la méthode `afficher` pour qu'elle affiche aussi le type de figure.

3. Carte (3/3) : Listes d'éléments

std::list : parcours, ajout, suppression.

Cet exercice 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).

`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 le cadre de cet exercice, nous demandons explicitement d'utiliser des listes (à des fins pédagogiques, pour se familiariser avec ce conteneur) même si les listes n'apportent pas grand-chose par rapport à un vecteur.

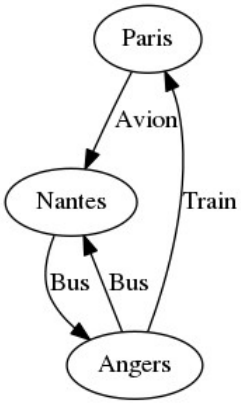
1. Une `carte` est une zone rectangulaire dans laquelle l'origine des coordonnées est le point (0,0) situé dans le coin supérieur gauche. Une `carte` a donc une `taille` (mais pas de `position`). Une `carte` contient un ensemble de `personnage` (représenté par une `std::list`), un ensemble de `obstacle` (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. La classe `carte` sera déclarée et définie dans un couple de fichiers `carte.hh/.cc`.
2. Écrire un constructeur, construisant une `carte` « vide » (c'est-à-dire sans aucun élément).
3. Écrire une (ou des) méthode(s) `ajouter` de `carte` permettant d'ajouter à une carte un `obstacle`, un `objetramassable` et un `personnage`.
4. Écrire une méthode `afficherpersonnages` dans `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 `intersectionelements` 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 `deplacementpossible` 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. Cette méthode ne réalisera pas le déplacement, elle se contentera de tester s'il est possible.

7. Écrire dans `carte` une méthode `deplacement` effectuant le déplacement d'un `personnage` dans une direction passée en paramètre (c'est-à-dire modifiant sa position si le déplacement est possible ; ne faisant rien si le déplacement n'est pas possible).
8. Écrire dans `carte` une méthode `ramassagepossible` 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 `ramassertout` 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. Lire attentivement (et comprendre !) la documentation de la méthode `erase` de std::list https://en.cppreference.com/w/cpp/container/list/erase et notamment ceci : « *References and iterators to the erased elements are invalidated.* » et la section « *Return value* ». Faites un test avec un personnage qui peut ramasser deux objets.

4. Graphe orienté étiqueté

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



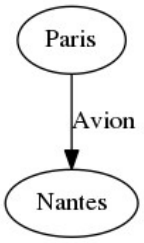
Dans cet exercice, on va écrire une classe permettant de manipuler des graphes orientés, et disposant d'une fonction d'export du graphe au format dot (pour plus d'informations sur ce format : https://graphviz.gitlab.io/_pages/doc/info/lang.html mais il n'est pas indispensable de consulter cette documentation) 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. Les classes que vous écrirez devront pouvoir être utilisées avec ce `main` sans aucune modification. Vous pouvez consulter ce `main` si vous avez un doute sur la signature d'une des méthodes demandées dans les questions suviantes.

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 (à télécharger dans l'espace Moodle) 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) (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.

```
#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");
```

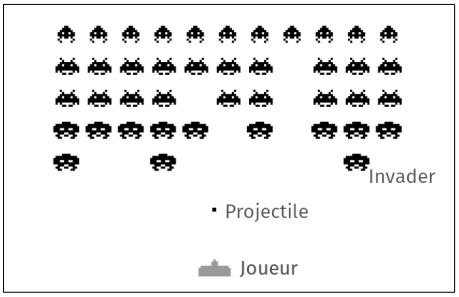
```
    g.ajouterarc(paris, nantes, "Avion");
    sauvergraphe(g, "tout.dot");
    g.supprimersommet(angers);
    sauvergraphe(g, "sansangers.dot");
    return 0;
}
```

5. Space invaders

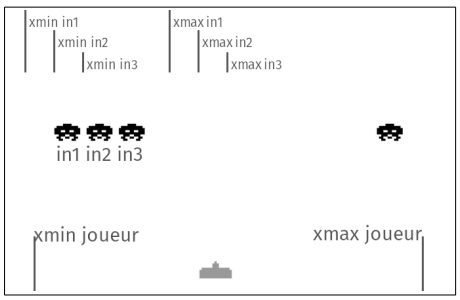
Méthode virtuelle, Classe abstraite.

Nous allons faire une version simplifiée du jeu Space Invaders *pour lequel vous trouverez plus d'informations ici : https://fr.wikipedia.org/wiki/Space_Invaders. Cet exercice est une version légèrement modifiée de l'évaluation pratique de mi-parcours de l'année 2022-23. Notez que dans l'évaluation de 2022-23, seule la partie « moteur » du jeu était demandée (jusqu'à la question 17, donc). La réalisation de la petite interface permettant de voir et jouer la partie (questions 18 et 19) avait été laissée en exercice après l'évaluation.*

1. Les différents éléments qui composent le jeu ont une position, qui est formée d'un couple d'entiers non signés représentant des coordonnées dans le plan. Écrire une classe `position` pour représenter ces positions, et munir cette classe des méthodes suivantes :
   Constructeur à deux arguments entiers, pas de constructeur par défaut ;
   Deux accesseurs nommés `x` et `y` aux coordonnés ; Deux mutateurs sur les coordonnées `setx` et `sety` ;
   Une méthode `tostring` retournant une chaîne de caractères de la forme « (5,3) » ;
   Une méthode `egale` prenant comme paramètre une `position` et retournant `true` si la position courante est la même que la position passée, `false` sinon (en d'autres termes, si a et b sont des `position`, `a.egale(b)` vaudra `true` si a et b repèrent la même position, `false` sinon).
   Une méthode `differente`, comparable à `egale`, mais retournant le résultat inverse : `true` si les positions sont différentes, `false` si elles sont égales.
2. Écrire une classe `taille` permettant de représenter la taille d'un élément sous la forme de deux entiers non signés pour la largeur et la hauteur, et munir cette classe d'un constructeur à deux arguments, accesseurs nommés `w` (largeur) et `h` (hauteur) et mutateur pour la largeur `setw`.



3. Les éléments du jeu sont appelés *objets* et sont de différents types (cf. figure ci-contre) : le *joueur*, les *invaders*, et les *projectiles* (tirés par le *joueur* en direction des *invaders*). Chaque *objet* a une position (repérant son point supérieur gauche) et une taille. La taille d'un *projectile* est (1,1), celle d'un *invader* est (11,8), et celle d'un *joueur* est variable : initialisée à la construction, elle pourra être modifiée par la suite.
   Joueur et *invaders* sont des *objets actifs* : ils se déplacent dans une `direction` (trois valeurs possibles : `stop`, `droite`, `gauche`) et ont une amplitude de déplacement représentée par un couple d'entiers appelés *xmin* et *xmax* représentant entre quel point (*xmin*, à gauche) et quel point (*xmax*, à droite) l'objet actif peut se déplacer. La `direction`, *xmin* et *xmax* sont des paramètres des constructeurs et sont spécifiques à chaque instance. Par exemple, sur la figure ci-contre, on voit que les 3 *invaders in1*, *in2*, *in3* ont des `xmin` et `xmax` différents afin qu'une « ligne » d'*invaders* reste bien formée (une ligne se déplace toujours « en entier »).[1]
   Définir des classes pour représenter ces différents objets, et



1 Vous pouvez aussi consulter le « gif » de l'espace moodle qui est une capture du code qui vous est demandé ici (en incluant les questions suivantes, mais ce qui nous intéresse ici est le mouvement des objets).

définir des constructeurs avec les signatures suivantes :
projectile : position ;
invader : position, xmin, xmax, direction ;
joueur : position, taille, xmin, xmax, direction.

4. Dans ces classes, définir un accesseur `pos` et un mutateur `setpos` pour la position.
5. Pour les *objets actifs*, définir des accesseurs `xmin` et `xmax` pour l'amplitude de déplacement ainsi qu'un accesseur `dir` et mutateur `setdir` pour la direction.
6. Pour un *joueur*, définir une méthode `malus` qui multiplie par 2 la largeur (avec un maximum de 100) et `bonus` qui divise par deux la largeur (avec un minimum de 3).
7. Définir une méthode `tai` permettant d'accéder à la taille.
8. Définir une méthode `contientposition` prenant comme argument une `position` et permettant de tester si la `position` passée est à l'intérieur de l'objet.
9. Écrire dans *objet* une méthode `typeobjet` retournant un caractère : `I` pour *invader*, `P` pour *projectile*, `J` pour *joueur*.
   Écrire une méthode `sortieflux` prenant comme argument un flux de sortie (`std::ostream`) et sortant sur le flux le type de l'objet (le caractère de `typeobjet`) ainsi que sa position. *Réfléchissez à factoriser au mieux le code pour cette question : le code faisant ce traitement ne devrait être écrit qu'une seule fois.*
10. Définir une méthode `appliquerdeplacement` (sans argument) appliquant un déplacement sur l'objet :
    projectile : si *y* est strictement positif, il est décrémenté (un projectile va toujours vers le haut).
    objets actifs : si la direction est `stop`, ne fait rien, sinon déplace l'objet en fonction de la direction (en incrémentant ou décrémentant *x*) sauf si *xmin* ou *xmax* sont atteints[1], dans ce cas-là, ne fait rien. Cas particulier des *invaders* : lors d'un déplacement vers la gauche, si *xmin* est atteint, l'*invader* se déplace d'un vers le bas (*y* est incrémenté) et repart vers la droite (voir le gif sur l'espace moodle). (idem pour un déplacement vers la droite si *xmax* est atteint : déplacement d'un vers le bas et repart vers la gauche).
11. Écrire une classe `jeu` mémorisant une **liste** d'*invaders*, une liste de *projectiles* et un *joueur*. Munir cette classe d'un constructeur par défaut construisant un jeu sans aucun *invader*, aucun *projectile* et un *joueur* en position 144, 80 de taille 9, 8, de xmin 10, xmax 310, direction stop.
12. Écrire des méthodes `ajouter` permettant d'ajouter un *projectile* et un *invader*. La méthode d'ajout de *projectile* refusera d'ajouter un *projectile* s'il existe déjà un *projectile* à la même position. Si tel est le cas, elle retournera `false`. Si l'ajout est effectué normalement, elle retournera `true`.
13. Rajouter dans *jeu* un accesseur au joueur appelé `joueur`, un accesseur à la liste d'invaders appelé `invaders` et retournant une `std::list<invader>` et un accesseur à la liste de projectiles appelé `projectiles` et retournant une `std::list<projectile>`.
14. Écrire une méthode `tirjoueur` ajoutant un nouveau projectile partant du milieu (sur l'axe x) au-dessus (sur l'axe y) du joueur.
15. Écrire une méthode `afficher` prenant comme argument un flux de sortie et sortant sur ce flux les informations (via la méthode `sortieflux` demandée plus haut) sur tous les objets du jeu.
16. Écrire une méthode privée `projectiledisparition` supprimant les *projectiles* dont le y est égal à 0[2], et une méthode privée `projectilecollision` gérant la collision d'un *projectile* avec un *invader* : quand un *projectile* entre en collision avec un *invader*, le projectile comme l'*invader* sont détruits et disparaissent du jeu.
17. Écrire une méthode `tourdejeu` calculant une évolution l'état du jeu : les *invaders*, le *joueur*, les *projectiles* se déplacent (en appelant une fois `appliquerdeplacement`), les projectiles ayant atteint le haut de l'écran disparaissent et les collisions sont gérées.

1 N'oubliez pas de prendre en compte la largeur de l'objet.

2 Cela est visible sur le gif cité ci-dessus, pour les quelques projectiles qui ont raté leur cible.

18. **Réalisation de l'interface.**
    Le code que nous vous proposons de réaliser utilise la bibliothèque SFML https://www.sfml-dev.org/index-fr.php . Elle n'est pas installée sur les ordinateurs « All-in-one » des salles de TP, donc pour réaliser la suite de l'exercice, vous devrez travailler sur vos portables et installer sfml en exécutant la commande suivante (en root) :
    ```
    apt install libsfml-dev
    ```
19. Pour compiler le code, utilisez le `CMakeLists.txt` que nous vous fournissons sur l'espace Moodle (car il faut fournir à CMake des éléments particuliers pour lui indiquer que la compilation doit se faire en prenant en compte la bibliothèque) et le main que vous trouverez au même endroit. Vous pouvez évidemment modifier ce `CMakeLists.txt` pour prendre en compte vos noms de fichiers. Le fichier `sprites.png` (à télécharger sur Moodle) doit être placé dans le répertoire contenant l'exécutable (donc le répertoire build... créé par QtCreator, pas le répertoire contenant vos sources) avant l'exécution. S'il n'est pas trouvé, l'exécutable se terminera.
    Remarque : Le code que nous vous donnons ici est simplement un main très simple permettant de mettre en œuvre les classes demandées lors du TP, il n'est pas forcément très élégant et n'utilise pas toutes les fonctionnalités de SFML, mais le but était de vous montrer un code court permettant de réaliser un petit jeu sur ce que vous avez réalisé. Notez par ailleurs que dans ces conditions, le Space Invaders que nous avons réalisé n'a pas grand intérêt : les invaders devraient lancer des projectiles eux aussi, il devrait y avoir des murs derrière se cacher, un score, etc.

6. 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 pour fournir les fonctionnalités suivantes :

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. Écrire une méthode `appartient` permettant de tester si un entier appartient à l'ensemble.
5. Écrire une méthode `vide` permettant de tester si un ensemble est vide.
6. Écrire une méthode `ajouter` permettant d'ajouter un entier à un ensemble (si l'entier appartient déjà à l'ensemble, la méthode ne fait rien). La méthode devra donc faire deux choses différentes : tester si l'ajout doit être fait et faire l'ajout si nécessaire. Réfléchir à la façon de factoriser au mieux le code pour que l'effort à faire à chaque fois qu'une nouvelle sous-classe d'ensemble sera créée soit minimal.

7. Écrire une méthode `afficher` affichant le contenu d'un ensemble.
8. Écrire une méthode `enlever` permettant d'enlever un élément d'un ensemble. Si l'élément ne fait pas partie de l'ensemble, la méthode ne fera rien.
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, lisez la suite : 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.

   Si vous ne voyez toujours pas comment faire, lisez la suite : un parcours d'une liste, d'un tableau ou d'un `vector` se fait à l'aide d'un objet identifiant la position courante (un indice, un itérateur) qui permet 4 traitements différents :

   Cet objet est initialisé d'une façon particulière au début du parcours (indice mis à 0, iterateur mis à `begin`). Définir une classe `parcours` permettant de repérer un élément dans un ensemble et une méthode `commencer` dans `ensemble` retournant un `parcours` initialisé au début.

   Cet objet doit pouvoir passer « au suivant » (incrémentation dans le cas d'un indice, `++` sur un itérateur). Définir une méthode `suivant` dans `ensemble` prenant comme argument un `parcours` et le faisant passer au suivant.

   Il doit être possible de tester si cet objet a atteint la fin du parcours (comparaison avec la taille du tableau dans le cas d'un indice, comparaison avec `end` dans le cas d'un iterateur). Écrire une méthode `estfini` prenant comme argument un `parcours` et testant si le parcours est fini.

   Et enfin, il faut à partir de cet objet, accéder à l'élément courant de l'ensemble (en utilisant les crochets à partir d'un indice, ou en utilisant `*it` dans le cas d'un iterateur). Écrire une méthode `courant` prenant comme argument un `parcours` et retournant l'élément correspondant à ce parcours.

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).
13. Revenir sur le constructeur par recopie pour permettre de construire par recopie une instance de n'importe quelle sous-classe d'ensemble à partir d'une instance de n'importe quelle sous-classe d'ensemble. En d'autres termes, il faudra pouvoir construire, par exemple, un `ensembletableau` comme copie d'un `ensemblevector`.