TP 2 — Classes : concepts de base

1. Carte (1/3) : Taille et position

Classe, Attributs, Méthodes, Constructeurs, Liste d'initialisation, Instanciation, Méthodes const.

Dans cet exercice nous allons écrire deux classes extrêmement simples qui seront utilisées plus tard pour gérer une carte dans laquelle sont placées différentes entités avec des caractéristiques différentes (personnages, objets, etc.). Toutes ces entités ont une forme rectangulaire sur une carte qui peut être vue, elle aussi, comme un rectangle. On aura donc besoin de les repérer par leur position *(le point du plan situé en haut à gauche du rectangle) et leur* taille *(largeur et hauteur).*

Les types et classes demandés ici seront déclarés dans un fichier `tailleposition.hh` et définis dans `tailleposition.cc`. Vous écrirez un fichier `main.cc` contenant un `main` de test.

1. Déclarer un type `coordonnee` comme un entier, ce type sera utilisé à la fois pour représenter abscisse, ordonnée, largeur et hauteur.
2. Déclarer une classe `position` composée de deux attributs de type `coordonnee` `x` et `y`, ainsi que des méthodes suivantes : constructeur prenant comme paramètre deux coordonnées, constructeur par recopie, accesseur à x, accesseur à y, mutateur de x, mutateur de y, méthode `comparaison` prenant comme paramètre une `position` et retournant un booléen valant `true` si la position passée est la même que la position courante, et `false` sinon.
3. Déclarer une classe `taille` composée de deux attributs `largeur` et `hauteur` de type `coordonnee`. Munir cette classe d'un constructeur, accesseurs, mutateurs.

2. Séquence de couleurs (3/5) : Classe

Classe, Allocation dynamique, Constructeur, Constructeur par recopie, Destructeur.

On veut écrire sous forme de classe la structure de données définie dans la partie « Séquence de couleurs (2/5) », en écrivant un code robuste, *c'est-à-dire qu'il est « impossible » de prendre en défaut.*

1. Réfléchir à différentes façons de provoquer des erreurs d'exécution dans les fonctions définies dans la partie 2 de cette suite d'exercices. Écrire et tester un `main` mettant en œuvre une ou deux erreurs possibles identifiées précédemment (à faire dans le projet de la partie 2). La suite de l'exercice devra être traitée en gardant ces erreurs à l'esprit et en imaginant que la classe que vous écrirez ici sera utilisée par un autre développeur, qui pourrait « mal » utiliser votre classe, sans que cette « mauvaise » utilisation ne provoque d'erreurs d'exécution.
2. Écrire une classe fournissant les mêmes fonctionnalités que celles vues précédemment. La classe devra être écrite de façon à ce que le code ci-dessous (pouvant être téléchargé dans l'espace Moodle) fonctionne sans aucune modification. La plupart des méthodes n'ont pas besoin de description supplémentaire et correspondent à ce qui a été décrit précédemment, seules certaines méthodes sont décrites plus précisément dans les questions suivantes, mais vous devez évidemment toutes les écrire, et tester votre code au fur et à mesure. Gardez à l'esprit que tout ce qui est alloué dynamiquement (`new, new[]`) doit être libéré (`delete, delete []`). (et n'oubliez pas la question 6 ci-après)

```
#include "sequence.hh"
#include <iostream>

int main() {
    sequence a;
    a.ajouter(couleur::rouge);
    a.ajouter(couleur::bleu);
    a.ajouter(couleur::rouge);
```

```
    a.ajouter(couleur::vert);
    sequence b(a); // voir question 3
    a.afficher(a.acces(0)); std::cout << "\n";
    for (indicesequence i=0; i<a.taille(); ++i) {
          a.afficher(a.acces(i));
          std::cout << " ";
    }
    std::cout << "\n";
    a.vider();
    std::cout << "A: "; a.afficher(std::cout); std::cout << "\n"; // voir
question 4
    std::cout << "B: "; b.afficher(std::cout); std::cout << "\n";
    std::cout << a.comparer(b) << "\n";
    a.copier(b); // voir question 5
    std::cout << a.comparer(b) << "\n";
    return 0;
}
```

3. Constructeur par recopie créant une nouvelle séquence par copie d'une séquence existante.
4. La méthode `afficher` prendra comme paramètre un flux (instance de `std::ostream`) et affichera sur ce flux le contenu de la séquence. `std::cout` est une instance de `std::ostream`, mais ce n'est pas la seule : `std::cerr` en est une autre, et écrire dans un fichier (nous verrons cela plus tard) se fait par l'envoi dans une instance d'une sous-classe de `std::ostream`. Écrire un code prenant un flux comme paramètre (plutôt que d'utiliser directement `std::cout` dans le corps de la méthode) permet de réutiliser la méthode dans un autre contexte.
5. Copie en écrasant le contenu de l'objet sur lequel la méthode est appelée avec le contenu de l'objet passé en paramètre.
6. Chercher une façon de provoquer une erreur dans le code des méthodes de cette classe en écrivant un programme de test utilisant « mal » la classe. Si c'est possible, cela signifie que la classe pourrait être plus robuste.

3. Utilisation de `std::vector`

std::vector, Boucle for d'intervalle, Passage par valeur, référence, référence constante.

Référence : http://en.cppreference.com/w/cpp/container/vector. Exemple ci-dessous sur Moodle.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v1; // Construction à vide
    for (int i(0); i<10; i++)
          v1.push_back(i);
    std::vector<int> v2(v1); // Construction par recopie
    // Parcours par indice
    for (std::vector<int>::size_type i(0); i<v1.size(); ++i)
          std::cout << v1[i];
    // Parcours par itérateur
    for (std::vector<int>::const_iterator i(v1.begin()); i!=v1.end(); ++i)
          std::cout << *i;
    // Parcours et modification par itérateur
    for (std::vector<int>::iterator i(v1.begin()); i!=v1.end(); ++i)
          *i = *i +1;
    // Parcours et modification par itérateur
    for (auto i(v1.begin()); i!=v1.end(); ++i)
          *i = *i +1;
    // Parcours par boucle for d'intervalle (valeur)
    for (auto i : v1)
          std::cout << i;
```

```
    // Parcours par boucle for d'intervalle (référence)
    for (auto & i : v1)
        i *= 2;
    // Utilisation de l'opérateur d'affectation.
    v2 = v1;
    return 0;
}
```

1. Écrire une fonction permettant de saisir des valeurs dans un vecteur d'entiers, le vecteur étant « retourné » par la fonction. Réfléchir aux différences entre l'utilisation d'un vecteur comme « valeur de retour » et un vecteur passé comme paramètre, par référence.
2. Écrire une fonction retournant le maximum parmi toutes les valeurs entières d'un vecteur d'entiers : Écrire 3 variantes de la fonction, en utilisant (1) un indice, (2) un iterateur, (3) une boucle `for` d'intervalle.

4. Séquence de couleurs (4/5) : vecteur

Classe, Liste d'initialisation, std::vector.

La classe écrite en « Séquence de couleurs (3) » nous a demandé de définir un constructeur par défaut, un constructeur par recopie, un destructeur, et manipuler des pointeurs. Ceci nous a permis de gérer une structure de données de taille variable, avec une certaine efficacité (avec le redimensionnement au besoin, et non pas systématique), au prix d'une certaine « complexité » du code, source d'erreurs.

1. Écrire une classe `sequence` ayant exactement la même interface que celle écrite dans la partie précédente de la suite d'exercices (et qui devrait fonctionner avec le `main` donné plus haut sans aucune modification), dans laquelle les éléments de la séquence sont représentés sous la forme d'un `std::vector`.

Le `std::vector` étant un conteneur de taille variable, il est évidemment mieux adapté que `std::array`, et plus simple à utiliser qu'une zone allouée dynamiquement. Il sera toujours préférable de trouver la meilleure structure de données offerte par la bibliothèque standard plutôt que réinventer la roue. De plus, il permet d'appliquer la « règle des 0 » : si vous ne l'avez pas appliquée, analysez votre code afin de voir si vous pouvez l'appliquer.

5. Jeu de la vie

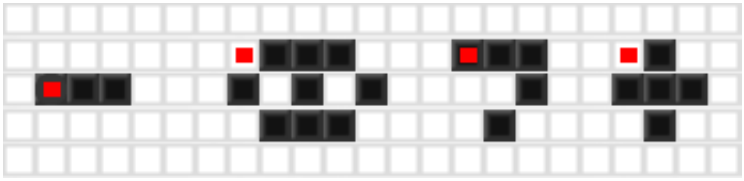
Classe, std::vector, Méthodes, Encapsulation.

On veut définir une classe permettant de programmer le jeu de la vie de Conway. *https://fr.wikipedia.org/wiki/Jeu_de_la_vie*

1. L'état du jeu est représenté par une grille à deux dimensions d'une taille fixée (par des paramètres du constructeur de la classe), chaque case de la grille contient une valeur d'un type énuméré `etatcellule` pouvant prendre les valeurs `vivant` ou `mort`. Définir un type `coordonnee`, et une classe `grille`, représentant un état de grille. Le constructeur prendra comme paramètre un couple de coordonnées représentant la taille de la grille (largeur, hauteur), et il est conseillé d'utiliser une classe de la bibliothèque standard (`std::array` ? `std::vector`) pour représenter l'état de la grille plutôt que d'utiliser une allocation dynamique d'une zone.
2. Rajouter à la classe `grille` les méthodes suivantes :
   `vider` supprimant toutes les cellules vivantes de la grille.
   `vivante` prenant comme paramètre un couple de coordonnées et retournant `true` si la cellule correspondante est vivante, `false` sinon.
   `generer` prenant comme paramètre un couple de coordonnées et générant une cellule vivante dans la grille aux coordonnées passées.
   `tuer` tuant la cellule vivante située aux coordonnées passées.

`afficher` affichant à l'écran, ligne par ligne, le contenu de la grille : le caractère `*` sera utilisé pour les cellules vivantes et l'espace pour les cellules mortes. Quand vous testez cette méthode, assurez-vous que votre terminal à l'écran a plus de lignes que la grille, sinon, vous ne verrez pas la grille en entier (idem pour les colonnes).

3. Certaines structures ont des particularités d'évolution intéressantes. Définir une énumération fortement typée `structure` définissant 4 symboles pour ces structures : `oscillateurligne`, `floraison`, `planeur`, `oscillateurcroix`. Écrire une méthode `ajouterstructure` dans `grille` prenant comme paramètre une `structure` et un couple de coordonnées, et rajoutant à la grille la structure correspondante aux coordonnées passées. Les structures sont définies ci-contre. Le point rouge sur ces figures correspond au couple de coordonnées passé en paramètre, la structure sera créée dans la grille à partir de ce point (qui est une cellule morte dans le cas d'une `floraison`).



oscillateurligne floraison planeur oscilateurcroix

4. Écrire dans `grille` une méthode `vivantes` prenant comme paramètre un couple de coordonnées et retournant le nombre de cellules vivantes dans les 8 cases entourant la case de la cellule. Attention aux bords de la grille. Une solution pour faciliter l'écriture du code serait de définir `coordonnee` comme un type `signed`.
5. Écrire une méthode `evolution` de `grille` prenant comme paramètre une autre `grille` et calculant dans la `grille` passée en paramètre l'évolution de la `grille` sur laquelle la méthode a été appelée. L'évolution est calculée ainsi :
   Une cellule morte prend vie si elle est entourée de 3 cellules vivantes (exactement).
   Une cellule vivante meurt d'isolement si elle est entourée de 0 ou 1 cellule vivante, et meurt d'étouffement si elle est entourée de 4 (ou plus) cellules vivantes.
6. Écrire un programme principal dans lequel après avoir placé une structure au milieu de la grille, l'évolution de la grille est calculée et affichée, et attend un appui sur Entrée pour calculer et afficher l'état suivant. Pour lecture au clavier, vous utiliserez `std::getline(std::cin, s)` (`s` étant une `std::string`)[1] plutôt que `std::cin >> s` car `getline` permet de lire n'importe quelle chaîne (y compris vide, y compris contenant des caractères séparateurs tels que l'espace) alors que l'opérateur `>>` attend la saisie d'un mot non vide (et refuse donc les espaces considérés comme des séparateurs de mots, ainsi que la chaîne vide : cette fonction continue d'attendre si on appuie directement sur Entrée). Pour profiter au mieux de l'application, choisissez le même nombre de lignes/colonnes dans le terminal et pour la `grille`.

1 http://en.cppreference.com/w/cpp/string/basic_string/getline