Exercices C++ Avancés

Templates (Patrons)

Préalable : extraire la version complète de la classe **CFract** version flux C++ (avec les méthodes définies *inline*) dans un fichier **Fract.h** qui sera inclus dans les différents projets si nécessaire.

Exercice 1: Patrons de fonction

A. Concevoir une fonction générique **MaxVal** (générique par rapport à une classe **TObj**) qui retourne la plus grande valeur entre deux objets de type **TObj** reçus comme paramètre.

B. Testez-la avec deux **int**, deux **float**, un **int** et un **double**. Quelle est la conclusion ?

C. Quelle sont les demandes implicites pour la classe **TObj** ? Peut-on les réduire ?

D. Peut-on utiliser **MaxVal** avec deux objets de type **CFract** ? Mais un objet **CFract** et un objet double ?

Exercice 2: Patrons de fonction

A. Concevoir un patron de fonction **ShowVect** qui affiche vers un flux de sortie le type, la taille et le contenu d'un tableau d'objets **TObj** qu'il reçoit en argument ainsi que sa taille.

B. Concevoir un patron de fonction **VectSomme** qui effectue la somme d'un tableau (vecteur) d'objets **TObj** qu'il reçoit en argument ainsi que sa taille. Il doit retourner la somme ainsi calculée. Quelles sont les demandes que votre patron de fonction impose à la classe **TObj** ?

C. Tester le patron **VectSomme** pour des tableaux (globaux, locaux et dynamiques) d'objets de type **short**, **unsigned long**, **double** et **CFract**. Utiliser **les flux C++** pour l'affichage.

D. Réécrire **VectSomme** pour qu'il demande un peu moins à la classe **TObj**. Tester la nouvelle version.

E. Réécrire **VectSomme** en enlevant l'argument taille de la fonction et en le rajoutant comme argument du *template*. Est-ce possible ? Tester : marche-t-il avec les 3 types de tableaux ? Pourquoi ?

F. Ecrire un patron de fonction qui affiche le contenu de n'importe quel tableau d'objets. Idem pour un patron de fonction qui retourne la taille d'un tableau d'objets (de quelle nature ?). Idem pour des tableaux de tableaux (2D).

Exercice 3: Patrons de fonction

A. Concevoir une fonction **TriMontant** qui va trier du plus petit au plus grand les objets **TObj** pointés par un tableau de pointeurs reçu en argument. Sa taille arrive comme deuxième argument.

B. Quelles sont les demandes que votre patron de fonction impose à la classe **TObj** ?

C. Testez la fonction générique **TriMontant** en utilisant un tableau contenant des pointeurs vers de **int** ou vers des **CFract** ou vers des **CCard** (cartes de jeu).

Exercice 4: Patrons de classe

A. Modifier la classe **TriPascal** pour qu'elle soit un patron de classe par rapport au type de l'élément **TElem**.

B. Faites la liste des demandes vis-à-vis de la classe **TElem**.

C. Testez-la en la particularisant pour des **unsigned short**, **long** ou **unsigned __int64**. Utiliser **les flux C++** pour l'affichage. Tester les limitations en fonction du type choisi.

Exercice 5: Patrons de classe

A. Concevez une classe générique **Mat2D** qui encapsule une matrice bidimensionnelle d'éléments **CElem**, de la taille voulue par l'utilisateur, passée comme paramètre de *template*, au moment de sa création. Il y aura un constructeur sans paramètres qui ne fait rien, un autre qui demande un **CElem** et remplit toute la matrice avec sa valeur. Prévoyez aussi un constructeur copie, un opérateur d'attribution à partir d'une autre matrice de même type ou à partir d'un élément **CElem**. Prévoir un opérateur **parenthèses** à deux paramètres (ligne et colonne) permettant l'accès à un élément de la matrice. Prévoir une méthode **Rand** qui remplit chaque élément de la matrice par une valeur pseudo-aléatoire entre deux limites spécifiées comme paramètres.

B. Ecrire la méthode **Show** de la classe générique qui affiche les informations (taille, type d'élément, adresse) et le contenu de la matrice (une ligne de matrice par ligne de sortie) vers un flux C++ arrivé en paramètre. Prévoir un deuxième paramètre de type **size_t** qui signifie la largeur d'affichage pour chaque élément. Quand il est utilisé à l'appel,

sa dernière valeur est mémorisée dans un membre statique de la classe (dont la valeur d'origine est 10). S'il n'est pas utilisé à l'appel de Show alors on utilise comme largeur la valeur précédemment mémorisée dans le membre statique. Prévoir un setter public **SetW** qui permet de modifier la valeur du membre statique sans être obligé d'appeler **Show** avec 2 paramètres. Même démarche pour un 3ème paramètre qui désigne la précision (après virgule). Le membre statique qui lui est associé a comme valeur originale 3 et son setter est **SetPrec**. Prévoir aussi un opérateur << avec le même rôle que **Show** avec un seul paramètre (le flux C++).

C. Testez la matrice générique en utilisant comme éléments des **flot** ou des **char**. Déclarez une variable de la taille voulue, initialisez toute la matrice avec une valeur, puis l'élément au milieu de la matrice avec une autre valeur. Enfin, afficher le contenu de la matrice.

D. Rajouter une méthode de **Transposition** (résultat dans une nouvelle matrice) et l'opérateur associé "~". Rajouter les opérateurs d'addition et de multiplication avec une autre matrice ou un seul **CElem**. Tester avec des tailles différentes et énumérez les différences observées par rapport à une approche "non-template".

E. Rajouter comme méthode un opérateur * entre 2 objets **Mat2D** de même type d'éléments et de tailles correctes mathématiquement. Quels sont les nouveaux degrés de liberté que rajoute cet opérateur ? Prévoir donc, non pas une méthode, mais un patron de méthode avec le(s) degré(s) de liberté nécessaire(s). Tester !

F. On souhaite pouvoir additionner une matrice NxN de **int**s avec une matrice NxN de **float**s. Peut-on faire ce type d'opération actuellement ? Proposer une démarche assez générique : dans le cas précédent, c'est l'utilisateur qui doit faire le choix sur le résultat attendu (matrice de int ou float). D'une manière générale, on souhaite pouvoir transformer n'importe quelle matrice d'un type donné vers une autre de même taille mais avec un type d'élément compatible. Proposer une solution et tester. Attention aux degrés de liberté !

Exercice 6: Modélisation d'un polynôme à coefficients réels.

A. Concevoir 2 versions d'une classe **CPoly** capable de modéliser un polynôme d'ordre quelconque à coefficients réels (mémoriser seulement l'ordre du polynôme, inférieur à 256 et ses coefficients de type **float** dans un tableau). Une version utilise comme attribut un tableau de taille fixe **MAX_SZ** (égale à 256 est définie comme constante dans la classe) et l'une version utilise un tableau de taille variable alloué dynamiquement. Il n'y aura aucune différence de l'interface publique entre les 2 classes (donc à utiliser les mêmes tests).

B. Interface :

- Un constructeur à 5 paramètres permettant la création explicite d'un polynôme d'ordre entre 0 et 4 (attention, si par exemple l'utilisateur ne fournit pas les coefficients d'ordre 4 et 3 on crée un polynôme seulement d'ordre 2).
- Un constructeur à 2 paramètres qui signifient l'ordre du polynôme et le tableau des coefficients de type **float** à copier.
- Seulement s'il est nécessaire : constructeur copie, destructeur, opérateur de copie.
- Les opérateurs arithmétiques suivants : **+=** , **+** , ***=** et ***** (entre des polynômes). Attention aux prototypes, on veut le même comportement que pour les opérateurs arithmétiques classiques. Attention à la gestion de mémoire dynamique et aux différents cas : les polynômes opérandes n'ont pas forcement le même ordre.
- Une méthode **Show** qui prend en argument un *stream* de sortie de type **ostream** et qui affiche la formule du polynôme en commençant par les puissances les plus faibles.

C. Définir un opérateur << pour afficher le polynôme **CPoly** dans un *stream* de sortie de type **ostream**. On ne veut pas utiliser **friend**.

D. Faites le nécessaire pour pouvoir multiplier à droite ou à gauche un polynôme de type **CPoly** avec un **float** (ou **int**).

E. Surcharger l'opérateur parenthèses avec un seul argument (**x** de type **float**) pour qu'il retourne la valeur du polynôme pour **x**.

F. Tester chaque élément de l'interface (utilisez, entre autres, les lignes de test fournies) et afficher les résultats en utilisant l'opérateur << .

G. En utilisant le code déjà écrit, concevoir une classe *template* **TPoly/TPolyDyn** qui modélise un polynôme d'ordre quelconque à coefficients de type **TCoef**. Testez la classe **TPoly** en la particularisant pour **TCoef** de type **float**, **double** ou **long**. Ensuite écrire un patron de fonction de test et appelez-le pour des **float**, **int**, **unsigned char**, **CFract, complex**. Corriger les éventuels défauts, en spécialisant la méthode d'affichage.

H. Concevoir une nouvelle version *template* **TPoly** de la classe qui modélise le polynôme (dans un autre *namespace*) qui prend aussi l'ordre du polynôme comme paramètre générique. Comment cela change le code de la classe ? Et le code généré à l'exécution ?

Exercice 7: Utilisation de la classe/template vecteur (std::vector)

A. Utiliser la classe **vector** disponible dans bibliothèque STL en incluant l'entête <**vector**>. Quelles sont les différences entre les méthodes **size**(), **max_size**() et **capacity**() ? A quoi serve **reserve**() ? Est-ce absolument nécessaire ?

B. Rajouter dans un vecteur d'entiers les 10 premiers carrés (1, 4, 9, 16 etc.) et afficher leur contenu en utilisant l'opérateur "[ ]" ou les itérateurs.

C. Ecrire un patron d'une fonction et d'opérateur << capable d'afficher vers un flux C++ toutes les informations (taille, type, adresse) et le contenu du vecteur passé en 2ème argument.

Exercice 8: Utilisation de la classe/template std::string (STL)

Utiliser la classe **string** disponible dans bibliothèque STL en incluant l'entête <**string**>. Ecrire des fonctions de test pour tester l'ensemble des méthodes et des opérateurs de la classe. Idem pour **wstring**. Peut-on écrire des fonctions de test génériques ? Quelle est la différence d'approche entre **string** / **wstring** et les APIs **TextOutA** / **TextOutW** ?

Exercice 9: Modélisation d'une permutation

On souhaite modéliser en C++ des permutations d'un ordre donné. On utilise la notation/représentation sur une ligne/tableau (en.wikipedia.org/wiki/Permutation) mais avec des indexes qui commencent à 0. Par exemple, il y a maximum **3! = 6** permutations d'ordre 3, dans l'ordre : {0,1,2} {0,2,1} {1,0,2} {1,2,0} {2,0,1} et {2,1,0}. La 1ère est une identité car elle ne permute rien entre l'entrée et la sortie, la 2ème place le 3ème élément d'entrée (2) sur la 2ème place en sortie et le 2ème élément d'entrée (1) sur la 3ème place, etc. Pour optimiser la taille des futurs objets et leur manipulation, on envisage de mémoriser non pas ce tableau de permutation mais leur index dans la famille (entre **0** pour la 1ère et **ordre!-1** pour la dernière).

Un algorithme simple permet pour une permutation d'ordre **Ord** de passer de l'index **IPerm** de la permutation à son tableau de permutation **Tab** :

- Initialiser **Tab** par la 1ère permutation : {0, 1, 2 ... ordre-1}
- Du début à la fin, pour chaque case **i** du tableau **Tab** :
  - Calculer **ofs = IPerm / (Ord-i-1)!**
  - Copier dans **Tmp** la case **Tab[i+ofs]**
  - Décaler les cases **Tab[i]** à **Tab[i+ofs-1]** à droite (case voisine supérieure)
  - Copier **Tmp** dans **Tab[i]**
  - Actualiser **IPerm = Iperm % (Ord-i-1)!**

Un autre algorithme permet de passer d'un tableau de permutation **Tab** de taille **Ord** à l'index de permutation **IPerm** :

- Initialiser **IPerm = 0**
- Du début à la fin, pour chaque case **i** du tableau **Tab** :
  - rajouter à **IPerm** : **Tab[i]*(Ord-i-1)!**
  - pour toutes les cases de **i+1** à la fin, les décrémenter si leur valeur **> Tab[i]**

A. Parmi les types entiers que vous connaissez, quel est le mieux adapté pour mémoriser l'index de la permutation **IPerm** ? Et quel est le meilleur type pour les cases du tableau **Tab** ? Faire le choix puis écrire en commentaire votre réponse et préciser les limitations envisagées pour votre modèle. Quel est le meilleur endroit pour placer ces déclarations de types et les utiliser ensuite quand cela est nécessaire ?

B. Concevoir un modèle de classe **TPerm** qui modélisera n'importe quel type de permutation d'ordre **ORD** connu à la compilation. Chaque classe devra mémoriser son index dans la famille ainsi que le nombre maximum (constant) de membres de la famille (pas de tableau !). Prévoir au moins :

- Un constructeur avec 1 paramètre, par défaut 0, qui initialise l'index de la permutation. Bien sur, la valeur par défaut, donne naissance à la 1ère permutation (identité).
- Un autre constructeur qui prend un tableau de permutation de la taille correcte et initialise l'index de la permutation en conséquence.
- Destructeur, constructeur de copie, opérateur de copie seulement si nécessaire.
- Une méthode **GetIdx** qui retourne l'index de la permutation et une méthode **MaxIdx** qui retourne le nombre total des permutations d'un ordre donné.
- Une méthode (... comment ?) **Tab2Idx** qui prend un tableau de permutation de taille correcte et retourne un index de permutation.
- Une méthode (... comment ?) **Idx2Tab** qui prend un index de permutation comme 1èr paramètre et qui copie dans un tableau **Tab** passé en 2ème paramètre la traduction de la permutation.
- Une méthode **Next** qui ne retourne rien mais passe à la permutation suivante. Après la dernière on repasse à la première. Définir les opérateurs de pré- et post-incrémentation (++) à l'aide de cette méthode.

- Une méthode **Show** ainsi qu'un opérateur << (méthode ou fonction ?) qui affichent vers un flux C++ l'ordre puis l'index et la forme tableau de la permutation.
- Un opérateur *= qui applique la permutation de gauche P_L à la permutation de droite P_R et écrit le résultat dans celle de gauche. Définir un opérateur * à l'aide du premier. Equivalent mathématique et informatique: $P_L*P_R=\{P_L[P_R[i]]\}_{i=0..ORD-1}$. La multiplication est-elle commutative ? Exemple : {4, 2, 1, 3, 0} * {3, 0, 1, 4, 2} => {3, 4, 2, 0, 1}. Ici le $2^{ème}$ élément de sortie (4) est obtenu ainsi : c'est le $2^{ème}$ élément de la permutation de droite (0) qui est utilisée comme index dans celle de gauche et donc c'est (4).

Conseil : écrire une petite fonction ou méthode appelée **Fact** qui calcule la factorielle de l'argument. Elle vous servira plusieurs fois !

Tester l'ensemble des méthodes, fonctions et opérateurs pour des permutations d'ordre 5 et 10.
Comment peut-on déterminer le plus grand ordre supporté par notre modèle (*template*) de classe ?

C. Ecrire une fonction qui cherche pour les permutations d'ordre 4, dans l'ordre, si telle ou telle permutation est une racine d'ordre **N** de la permutation identité (c'est à dire qu'en la multipliant un nombre de fois **N** par elle-même, on arrive à la permutation identité).

D. Ecrire une fonction qui prend en arguments un **vector** et une permutation **TPerm** et retourne un nouveau vecteur qui représente l'original permuté. Tester la compatibilité de taille et provoquer une exception dans le mauvais cas.

Exercice 10: Modélisation de carrés magiques

A. En utilisant les composants déjà développés précédemment (**Mat2D**, **TPerm**), on souhaite concevoir un patron de classe **TMagic** qui modélise n'importe quel carré magique de taille N. On rappelle qu'un carré magique est une matrice carrée qui contient que des nombres entiers entre 1 et N et qu'il n'a jamais deux nombres identiques sur la même ligne ni sur la même colonne.

Prévoir un constructeur qui initialise la matrice : première ligne 1, 2, 3, ... N puis chaque diagonale garde les mêmes valeurs (donc décalage circulaire à droite d'une ligne sur la suivante). Le constructeur prend 2 arguments (avec des valeurs par défaut nulles) qui représentent les permutations des lignes et respectivement des colonnes à appliquer sur la matrice initiale, pour avoir un nouveau carré magique. (On rappelle aussi que la permutation de lignes ou de colonnes d'un carré magique gardent sa propriété "magique").

Tester pour différents ordres et différents indices de permutation.

B. Sur le même principe, créer une classe (*template*) **TSudoko** qui modélise n'importe quelle grille de Sudoku (complètement remplie) en partant de la grille suivante (à gauche) et en lui appliquant un nombre de permutations (combien ?) d'ordre 3. Par exemple, on peut créer une nouvelle grille à partir de l'initiale et en permutant entre elles les 3 premières lignes. Si vous trouvez toutes les permutations à appliquer vous pouvez aussi calculer le nombre de grilles (même si parfois on peut tomber sur les mêmes).

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8	9	1
5	6	7	8	9	1	2	3	4
8	9	1	2	3	4	5	6	7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5

Cette grille est basée sur 3x3 sous-carrés de taille 3x3. Dans l'état elle pourrait être une classe basée sur les *templates* déjà développés. Mais **TSudoku** généralise ce principe en utilisant son seul paramètre générique pour designer la taille des sous-carrés. Ainsi, une grille classique correspond à **TSudoku<3>**, mais on peut avoir des grilles de 4x4 modélisées par **TSudoku<2>** comme de 2x2 sous-carrés de taille 2x2 ou grilles de 16x16 modélisées par **TSudoku<4>** comme de 4x4 sous-carrés de taille 4x4. Pour obtenir toute la variété possible, le constructeur doit recevoir un seul paramètre entier qui englobe (mathématiquement comment ?) les index des permutations possibles.

Tester votre *template* **TSudoku** en affichant le contenu à la console pour les instances suivantes :

TSudoku<3>(0) TSudoku<3>(10) TSudoku<3>(1000) TSudoku<3>(1000000)

TSudoku<2>(0) TSudoku<2>(10) TSudoku<3>(100)

TSudoku<4>(0) TSudoku<4>(10) TSudoku<4>(1000) TSudoku<4>(1000000)

n7l 10 w nn2l kl da 3la visual 2017 b3d mn3ml nettoyage llsolution w nfix l warnings li htb2a errors