

Examen, INFO702, Session 1

Durée: 1h30

Documents autorisés : tous documents du cours/td/tp, notes manuscrites (nb: pas de livres)

Les exercices sont indépendants. Le barème est indicatif. La durée est 2h pour les étudiants bénéficiant d'un 1/3 temps.

1 Programmation générique (/4)

En utilisant potentiellement des structures de données classiques de la STL, écrivez une méthode générique `number_of_distinct_elements` qui détermine pour un intervalle (*range*) de données, le nombre de ces éléments distincts. On pourrait l'utiliser ainsi:

```
int t[ 5 ] = { 3, 6, 3, 2, 2 };
std::vector< char > u { 'f', 'u', 'z', 'z', 'i', 'n', 'e', 's', 's' };
std::list< std::string > v { "le", "boucher", "et", "le", "charcutier" };
int nb_t = number_of_distinct_elements( t, t + 5 ); // 3
int nb_u = number_of_distinct_elements( u.begin(), u.end() ); // 7
int nb_v = number_of_distinct_elements( v.begin(), v.end() ); // 4
```

NB: on peut récupérer le type pointé par un itérateur de type `It` avec le code `typename It::value_type`. On supposera que le type pointé est `LessThanComparable`.

2 Polymorphisme, échecs en C++ (/8)

On vous donne le code suivant, qui permet d'afficher un échiquier (vide ici).

```
#include <iostream>

std::string console( std::string code )
{ return "\033[" + code + "m"; }
auto normal = console("0"); // revient à normal
auto rouge = console("1;31"); // rouge gras
auto bleu = console("1;34"); // bleu gras
auto fnoir = console("40"); // fond noir
auto fgris = console("47"); // fond gris

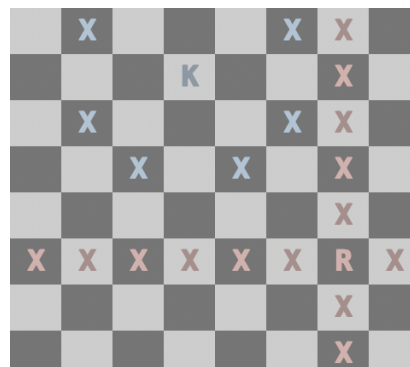
struct Piece; // forward declaration
class Square {
    bool m_color; // couleur
    Piece* m_piece;
public:
    Square() : m_piece( nullptr ) {}
    Square( bool color )
        : m_color( color ), m_piece( nullptr ) {}
    void change( Piece* p )
    { m_piece = p; }
    void display() const
    {
        std::cout << normal
            << ( m_color ? fgris : fnoir )
            << " " << normal;
    }
};

struct ChessBoard {
    Square squares[8][8];
    ChessBoard()
    {
        for ( int i = 0; i < 8; i++ )
            for ( int j = 0; j < 8; j++ )
                squares[ i ][ j ] = Square((i+j)%2 == 0 );
    }
    void display() const
    {
        for ( int i = 0; i < 8; i++ ) {
            for ( int j = 0; j < 8; j++ )
                squares[ i ][ j ].display();
            std::cout << std::endl;
        }
    }
};

int main()
{
    ChessBoard D;
    D.display();
    std::cout << std::endl;
    return 0;
}
```

On veut maintenant placer des pièces dans l'échiquier, que l'on va modéliser sous forme d'une classe abstraite `Piece` et de classes concrètes dérivées `Knight` (Cavalier), `Rook` (Tour), etc. On écrit aussi une classe dérivée `Move` qui matérialise un mouvement possible. **Ecrivez le code de ces classes**, afin que le programme suivant affiche l'échiquier avec les 2 pièces indiquées et leurs mouvements possibles (les blancs en bleu et les noirs en rouge):

```
int main()
{
    ChessBoard D;
    Knight* Kw = new Knight(true, 1, 3); // K
    Rook* Rb = new Rook(false, 5, 6); // R
    D.place( Kw );
    D.place( Rb );
    for ( int i = 0; i < 8; i++ )
        for ( int j = 0; j < 8; j++ )
        {
            if ( Kw->isMovableTo( i, j ) )
                D.place( new Move(Kw->color,i,j) ); // X
            if ( Rb->isMovableTo( i, j ) )
                D.place( new Move(Rb->color,i,j) ); // X
        }
    D.display();
}
```



3 Tableaux multi-dimensionnels en C++ (/13,5)

Les tableaux multi-dimensionnels sont ubiquitaires en informatique, notamment en calcul scientifique (vecteurs et matrices), en imagerie (images 2D, 3D, voire plus), et en apprentissage profond (les blocs de calculs transforment les matrices nD en matrices $n'D$, où n peut être différent de n'). Nous allons développer une classe générique `NDArray` pour créer de tels tableaux. Notez que, quelle que soit la dimension N choisie pour le tableau, les données seront stockées de façon contiguë à l'aide d'un `std::vector`.

```
NDArray<2,int> T2( Index<2>({8,10}) ); // 8 colonnes, 10 lignes, avec 8*10=80 entiers dedans
NDArray<3,float> T3( Index<3>({4,3,2}) ); // 4 colonnes, 3 lignes, 2 profondeurs, avec 4*3*2=24 nombres à virgule flottante dedans.
```

On voit que l'indiciage de ces tableaux est la difficulté principale, car on veut rester générique. Le principe suivi est : avancer le premier indice déplace à la case suivante, le deuxième à la ligne suivante, le troisième à la profondeur suivante, etc. On va donc d'abord gérer les indices via une classe générique `Index`, puis ensuite développer la classe `NDArray`.

```
template <int N>
struct Index { // Indice = tableau de N coordonnées
    int _pos[ N ]; // dans l'ordre x, y, z, etc.
};
```

- (/0,5) Ecrivez le constructeur par défaut de `Index` de façon à le placer à l'index $0, \dots, 0$.
- (/0,5) Faut-il réécrire le constructeur par copie et le destructeur ? Justifiez en 1 ligne.
- (/1) Surchargez ses opérateurs `[]` pour qu'ils retournent en lecture et en lecture/écriture la k -ème coordonnée de l'indice.
- (/2) Ecrivez maintenant sa méthode `int number(const Index& shape) const` qui calcule le numéro de l'indice courant dans un tableau multidimensionnel de taille `shape` donné en paramètre.
Ex: si l'`Index idx` est `(3, 1, 2)`, et que `shape` vaut `(6, 8, 5)`, alors son numéro est $(2 * 8 + 1) * 6 + 3$. Le premier numéro dans ce tableau de taille `shape` est 0 pour l'indice `(0, 0, 0)` et le premier numéro en dehors est $240 = 6 * 8 * 5$, pour l'indice `(0, 0, 5)`.
- (/1) Ecrivez maintenant un constructeur `Index(std::initializer_list<int> L)`. Cela permet de créer facilement des indices multi-dimensionnels, e.g. `Index<3> idx({3,1,2})` crée l'indice `(3, 1, 2)`. `L` est un *range* (intervalle) énumérable avec des itérateurs ou avec `for (auto v : L)...`, avec `v` qui prend successivement les valeurs `3,1,2` (ici).
- (/2) Ecrivez maintenant la classe générique `NDArray` avec un constructeur par défaut qui crée un tableau N -dimensionnel vide et un constructeur `NDArray(const Index<N>& size)` qui construit un tel tableau avec la taille spécifiée.
NB: on stockera les données dans un `std::vector` de la bonne taille. On stockera aussi les tailles selon chaque coordonnée sous forme d'un `Index`.
- (/1) Ecrivez maintenant les accesseurs simples suivants:

```
int dim() const; // la dimension du tableau
int size() const; // le nombre d'éléments total du tableau
int size( int k ) const; // la taille selon la k-ème coordonnée
```
- (/1) Ecrivez la méthode `int number(const Index<N>& idx) const` qui retourne le numéro/indice dans le tableau de données correspondant à l'index `idx`. Vaut-il mieux passer cet index par valeur ou en référence constante ?
NB: N'oubliez pas de vous servir de la méthode `Index<N>::number`.
- (/1) Vous pouvez maintenant surcharger l'opérateur `[]` pour faire les accesseurs à chaque donnée, en passant un `Index` en paramètre.
- (/2) On veut maintenant faire une coupe dans le tableau multidimensionnel. Cette coupe a donc une dimension de moins. Pour simplifier on ne considère qu'une coupe selon la plus grande coordonnée. Ecrivez cette méthode `lastSlice(int k) const` qui retourne la k -ème coupe selon la plus grande coordonnée.

```
NDArray<3,float> T( { 4,3,5 } ); // 4 colonnes, 3 lignes, 5 profondeurs, avec 60 éléments
auto S2 = T.lastSlice( 2 ); // 4 colonnes, 3 lignes, coupe au milieu de T.
```
- (/0,5) *Reformer* un tableau multidimensionnel (*reshape*) consiste à changer les tailles selon les différents axes, tout en conservant exactement le même nombre d'éléments, sans déplacer les valeurs. Par exemple, c'est parfois pratique de voir une matrice comme un vecteur de toutes ses valeurs. Ecrivez donc une méthode `reshape` qui fait cela, si les nouvelles tailles sont compatibles.

```
NDArray<2,float> I( Index<2>( { 100, 100 } ) ); // Image 2D 100 x 100
NDArray<2,float> I1 = I;
I1.reshape( Index<2>( { 10000, 1 } ) ); // Image 2D 10000 x 1
auto V = I1.slice( 0 ); // le vecteur 1D correspondant
```
- (/1) Comment écriveriez-vous de la façon la plus simple possible des itérateurs pour ces tableaux multidimensionnels ?