NOM :

PRÉNOM :

PLACE :

Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2023-2024**
Examen du lundi 18 décembre 2023 (deux heures)

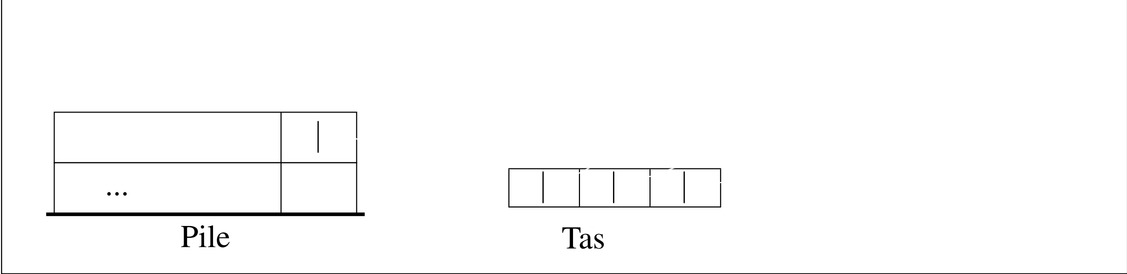
Calculatrices, téléphones mobiles et tout appareil électronique non autorisé doivent être éteints et déposés avec vos affaires personnelles. Seul document autorisé : une feuille au format A4 avec, au recto, le résumé de la syntaxe C++ de la fiche de TD 2 et, au verso, des notes manuscrites. Pour les étudiants inscrits en Français Langue Étrangère, un dictionnaire est autorisé.
Les exercices sont indépendants les uns des autres ; il n'est pas nécessaire de les faire dans l'ordre ; ceux marqués d'un ♣ sont plus difficiles mais font partie du barème sur 100. Le barème est indicatif et sujet à ajustements.
Les copies seront scannées en niveau de gris. Les réponses sont à donner sur le sujet, à l'intérieur des cadres prévus à cet effet ; en cas de besoin, utiliser la dernière page et mettre un renvoi. Pour assurer la lisibilité, nous recommandons un stylo à encre sombre (bleu ou noir). L'utilisation de crayon, d'encre de couleur claire, d'encre effaçable par friction est à vos risques et périls.

Exercice 1 (Cours : tableaux à deux dimensions (15 points)).

(1) Illustrez les étapes de la construction d'un tableau à deux dimensions par un fragment de programme construisant un tableau `t` avec trois lignes et quatre colonnes, initialisé de sorte que la valeur à la ligne i et colonne j soit $i + j$. Indiquez par des commentaires les étapes de la construction.

(2) En suivant les conventions du cours, complétez ci-dessous le dessin de la pile et du tas juste après l'exécution de votre fragment de programme immédiatement suivi par les instructions suivantes :

```
t[2] = t[1];
t[2][1] = 0;
```



PLACE :

Exercice 2 (fonctions, tests, conditionnelles, récursivité (23 points)).
Toutes les questions de cet exercice sont à faire seulement à l'aide des fonctions documentées ci-dessous et des méthodes `length()` ou `size()` de la classe `string`.

```
/** premiereLettre : renvoie la première lettre du mot **/
string premiereLettre(string mot);

/** trouve : indique l'indice de la première lettre
 * de la première occurence de chaine dans le texte t
 * @retour -1 si chaine ne figure pas dans le texte;
 * texte.lentgh() si chaine est vide; l'indice entre 0 et t.length()-1
 * de la première occurence de la première lettre de chaine dans t
 **/
int trouve (string t, string chaine);
/** suffixeMajuscule : copie depuis position en majuscule sans ponctuation
 *  Exemple : suffixe( 17, "Ici TOUT VA Bien, Va") vaut "VA"
 *     suffixe( 0, "évidence,      TOUTE simple .") vaut "EVIDENCE TOUTE SIMPLE"
 **/
string suffixeMajuscule (int position, string texte);
```

(1) Voici une première implantation d'une fonction `cite`. Entourez le ou les tests qui ne passent pas.

```
/// indique si un prénom p est cité dans un texte t (version 1).
bool cite (string t, string p){
  return (trouve( suffixeMajuscule(0,t), suffixeMajuscule(0,p) )>=0 ); }
```

```
CHECK ( cite( "HIER ABDEL ET ALEX ONT BIEN RI", "ALEX") );
CHECK ( not (cite( "ALEXANDRA JOUE DU PIANO", "ALEX") ) );
```

(2) La seconde implantation ci-dessous de la fonction `cite` ne compile pas pourquoi ?

```
/** indique si un prénom p est cité dans un texte t. (version 2) */
bool cite (string t, string p) {
    t = suffixeMajuscule(0, t);
    string recherche = suffixeMajuscule(0, p);
    int indice = trouve ( t, recherche );
    if ( (indice >= 0) and ( t.length() > indice + p.length() ) ) {
        recherche = recherche + " ";
    }
    if (indice > 0) {
        recherche = " " + recherche;
    }



}
```

(3) Ajoutez une ligne de code de votre choix à cette seconde implantation pour que, d'une part, elle compile et d'autre part qu'elle passe tous les tests et vérifie sa spécification.

(4) ♣ Parfois un prénom n'est pas cité en clair dans une phrase mais y est dissimulé : qui cherche le prénom peut le voir apparaître en omettant la lecture des autres lettres de la phrase. Compléter la fonction récursive `est_dissimule` ci-dessous pour qu'elle passe son test. Une instruction suffit.

```
/** est_dissimule: indique si un prénom se dissimule dans un texte.
 * @return vrai si le prénom peut être lu dans le texte en masquant
 *         les autres lettres et sans en changer l'ordre
 **/
bool est_dissimule (string texte, string prenom);
```

```
CHECK ( est_dissimule( "HIER ABDEL ET ALEX ONT BIEN RI", "ALAIN") );
CHECK ( not est_dissimule( "ALEXANDRA LIT", "RANDALL") );
CHECK ( not est_dissimule( "ALEXIS LIT", "NARUTO") );
```

```
bool est_dissimule (string phrase, string prenom) {
    if ( prenom.length() == 0)
        return true;
    int i = trouve(suffixeMajuscule(0, phrase), premiereLettre(prenom));






}
```

(5) Proposez un test pour la fonction `accrostiche` ci-dessous tel que le résultat est le mot `"INFO"`.

```
/** accrostiche : renvoie la première lettre de chacun des mots de la phrase.
 *  Exemple: accrostiche("CA VA") vaut "CV"   **/
string accrostiche(string phrase);
```

(6) Implantez la fonction `est_accrostiche` documentée ci-dessous.

```
/** est_accrostiche : indique si un poème est un ACCROSTICHE
 *  Exemple : est_accrostiche( "ALEX JOUE AU YOYO", "AJOY") vaut false
 *            est_accrostiche( "IBIS NICHE FIN OCTOBRE", "INFO") vaut true
 **/
bool est_accrostiche(string poeme, string message);
```

PLACE :

Exercice 3 (tableaux, fichiers, fonctions, boucles (20 points))**.**
On réalise un programme permettant de déterminer le temps de trajet entre deux stations de train sur une ligne.

On représente une ligne de train par un fichier avec une ligne pour chaque tronçon (espace entre deux stations). Pour chaque tronçon, on écrit sa **longueur** en kilomètres (donc la distance séparant les deux stations), puis la **vitesse** à laquelle vont les trains sur ce tronçon. Cette vitesse est exprimée en *nombre de minutes nécessaires pour parcourir un kilomètre*. Pour un fichier contenant n tronçons, il y a $n+1$ stations que l'on numérote de 0 à n. Le fichier suivant représente une ligne avec 5 stations. Les deux premières y sont séparées de 30 km. Par ailleurs, les trains mettent 3 minutes pour parcourir un kilomètre sur le tronçon entre les deux dernières stations.

train.txt

```
30 5
40 2
50 1
5 3
```

(1) Complétez le fragment de programme suivant pour construire et initaliser les tableaux `distances` et `vitesses` en ouvrant et lisant `"train.txt"`, de manière à faire passer les tests. Le fichier est au format indiqué ci-dessus, mais pourrait contenir d'autres valeurs, auquel cas votre code doit rester correct en adaptant juste les tests.

```
vector<int> distances;
vector<int> vitesses;









```

```
CHECK( distances == vector<int>({30, 40, 50, 5}) );
CHECK( vitesses  == vector<int>({5, 2, 1, 3})    );
```

(2) Implantez la fonction `tempsTotal` dont la documentation et les tests sont donnés :

```
/** Renvoie le temps total nécessaire pour parcourir toute la ligne
 * @param distances : tableau contenant la longueur de chaque tronçon
 * @param vitesses : tableau contenant le nombre de minutes nécessaires
 *     pour parcourir un kilomètre pour chaque tronçon
 * @return : le temps total en minutes pour parcourir tous les tronçons
 *     en prenant en compte les vitesses
 **/
int tempsTotal(vector<int> distances, vector<int> vitesses) {






}
```

PLACE :


```
CHECK( tempsTotal(vector<int>({100, 50}),
                  vector<int>({1, 2})) == 200 );
CHECK( tempsTotal(vector<int>({30, 40, 50, 5}),
                  vector<int>({5, 2, 1, 3})) == 295 );
```

(3) Implantez la fonction `sousTableau` dont la documentation et les tests sont donnés :

```
/** Renvoie un sous-tableau de tab aux bornes données
 * @param tab : un tableau d'entiers
 * @param debut : un indice de tab
 * @param fin : un indice de tab supérieur ou égal à debut
 * @return : un tableau contenant
 *   les éléments {tab[debut], tab[debut+1], ..., tab[fin]}
 **/
vector<int> sousTableau(vector<int> tab, int debut, int fin) {
```

```
CHECK( sousTableau(vector<int>({1, 2, 3, 4}), 0, 3)
       == vector<int>({1, 2, 3, 4}) );
CHECK( sousTableau(vector<int>({1, 2, 3, 4}), 1, 2)
       == vector<int>({2, 3}) );
```

(4) Implantez la fonction `tempsTrajet` dont la documentation et les tests sont donnés ci-dessous. **Rappel** : s'il y a n tronçons, il y a $n+1$ stations.

```
/** Renvoie le temps nécessaire à aller d'une gare à l'autre
 * @param distances : tableau contenant la longueur de chaque tronçon
 * @param vitesses : tableau contenant le nombre de minutes nécessaires
 *     pour parcourir un kilomètre pour chaque tronçon
 * @param depart : numéro de gare
 * @param arrivee : numéro de gare strictement supérieur à depart
 * @return temps nécessaire pour aller de la gare depart
 *     à la gare arrivee
 */
int tempsTrajet(vector<int> distances, vector<int> vitesses,
                int depart, int arrivee) {



}
```

```
CHECK( tempsTrajet(vector<int>({30, 40, 50, 5}),
                   vector<int>({5, 2, 1, 3}), 0, 4) == 295 );
CHECK( tempsTrajet(vector<int>({30, 40, 50, 5}),
                   vector<int>({5, 2, 1, 3}), 1, 2) == 80  );
```

PLACE :

Tous les exercices et toutes les questions qui suivent sont indépendantes et concernent des manipulations de tableaux à 1 ou 2 dimensions : le jeu Puissance 4 ne sert qu'à donner le contexte. Vous n'avez besoin ni de savoir jouer ni de connaître le détail des règles.

`Puissance 4` est un jeu où deux joueurs font tomber chacun à son tour un pion dans une grille verticale comptant 6 rangées et 7 colonnes, dans le but d'aligner quatre pions.

Dans les exercices suivants, on s'intéresse à l'écriture d'un programme pour jouer à Puissance 4. Par convention, on représentera par 1 le joueur 1 ou un de ses pions, par 2 le joueur 2 ou un de ses pions et par 0 une case vide. La grille de jeu sera représentée par un `vector<vector<int>>`, de sorte que `grille[3]` représente la quatrième rangée de la grille en partant du haut.

Pour tester les fonctions des exercices 4 et 5, on définit la variable globale suivante :

```
vector<vector<int>> grille = {
    {1, 0, 0, 0, 0, 2, 1},
    {1, 1, 0, 2, 0, 2, 2},
    {2, 1, 1, 2, 0, 2, 2},
    {2, 1, 1, 1, 1, 2, 1},
    {2, 2, 1, 2, 2, 1, 2},
    {1, 2, 2, 1, 2, 1, 1}
};
```

Exercice 4 (Affichage et sauvegarde : tableaux 2d, entrées-sorties, fichiers (12 points))**.**

La fonction suivante a été proposée pour afficher la grille de jeu :

```

/** Affiche la grille de puissance 4 sur le terminal
 * @param grille une grille de puissance 4
 **/
void afficheGrille(vector<vector<int>> grille) {
    for ( int i = 0; i < grille.size(); i++ ) {
        for ( int j = 0; j < grille[i].size(); j++ ) {
            cout << grille[i][j];

        }

    }

}
```

Lorsqu'on effectue l'appel suivant :

```
afficheGrille(grille);
```

on obtient l'affichage :

```
100002111020222112022211112122122121221211
```

(1) Retouchez la fonction ci-dessus (vous pouvez corriger directement sur le code) pour que la grille soit affichée ligne à ligne en séparant les colonnes par un espace. L'appel ci-dessus devrait donc donner :

```
1 0 0 0 0 2 1
1 1 0 2 0 2 2
2 1 1 2 0 2 2
2 1 1 1 1 2 1
2 2 1 2 2 1 2
1 2 2 1 2 1 1
```

PLACE :

(2) On voudrait pour pouvoir sauvegarder et charger des parties en cours. On utilise un format de fichier très simple : le fichier contient le mot « Puissance4 » (pour identifier le jeu) sur la première ligne puis une grille de jeu écrite ligne à ligne comme pour l'affichage. Ainsi, notre exemple `grille` sera représenté par le fichier :

```
Puissance4
1 0 0 0 0 2 1
1 1 0 2 0 2 2
2 1 1 2 0 2 2
2 1 1 1 1 2 1
2 2 1 2 2 1 2
1 2 2 1 2 1 1
```

Écrivez la fonction `chargePartie` qui permet de charger une partie précédemment sauvegardée dans un fichier suivant le format décrit. La fonction prend en paramètre le nom du fichier et renvoie la grille correspondante lue dans le fichier.

```
/** Charge une partie sauvegardée dans un fichier
 * La première ligne du fichier contient uniquement "Puissance4"
 * Le fichier contient ensuite la grille ligne par ligne en représentant
 * les pions par les numéros des joueurs
 * @param nomFichier le nom du fichier
 * @return la grille de puissance 4 du fichier (6 lignes et 7 cols)
 **/
vector<vector<int>> chargePartie(string nomFichier) {

}
```

PLACE :

Exercice 5 (Trouver le gagnant : tableaux 1d et 2d (30 points))**.**
Dans cet exercice, nous allons écrire différentes fonctions permettant de tester si un joueur a gagné en alignant quatre pions sur une ligne, colonne ou diagonale. **Les questions sont indépendantes.** La variable `grille` utilisée pour les tests est celle définie précédemment.

(1) On s'intéresse d'abord à l'alignement en ligne. Pour cela, on a écrit la fonction suivante permettant de tester un tableau à une dimension. Compléter les tests fournis avec, pour chacune des situations indiquées en commentaire, deux tests correspondant au résultats attendus pour les deux joueurs.

```
/** Teste si 'tableau' contient 'valeur' dans au moins quatre cases consécutives
 *  @param tableau un tableau d'entiers à une dimension
 *  @param valeur  un entier
 *  @return un booléen
**/
bool estGagnant1D(vector<int> tableau, int valeur) {
    int compteur = 0;
    for ( int i = 0; i < tableau.size(); i++ ) {
        if ( tableau[i] == valeur ) {
            compteur = compteur + 1;
            if ( compteur == 4 ) {
                return true;
            }
        } else {
            compteur = 0;
        }
    }
    return false;
}
```

```
CHECK(     estGagnant1D({2, 2, 1, 1, 1, 1, 0}, 1));
CHECK( not estGagnant1D({2, 2, 1, 1, 1, 1, 0}, 2));
// un tableau vide



// un tableau de taille 7 contenant une série de quatre 2 alignés



// un tableau de taille 7 contenant quatre 1 mais non alignés


```

(2) On a donné le projet Puissance4 à des étudiants en informatique ; l'un d'eux a écrit la fonction suivante mais son jeu ne fonctionne pas.

```
/** Teste si au moins une ligne de la grille contient
    au moins 4 pions alignés du 'joueur'
 * @param grille une grille de puissance 4
 * @param joueur un entier 1 ou 2 désignant le joueur
 * @return un booleen
 **/
```

PLACE :


```
bool ligneGagnanteBug(vector<vector<int>> grille, int joueur) {

    for ( int i = 0; i < grille.size(); i++ ) {
        if ( estGagnant1D(grille[i], joueur) ) {
            return true;
        } else {
            return false;
        }
    }

}
```

(a) Entourez le ou les tests ci-dessous qui ne passent pas avec la fonction telle qu'elle est écrite par l'étudiant.

```
CHECK(     ligneGagnanteBug(grille, 1) ); // j1 gagne ligne 3
CHECK( not ligneGagnanteBug(grille, 2) ); // j2 n'a pas de ligne gagnante
```

(b) Modifiez la fonction pour qu'elle réponde à sa documentation et que les tests passent (corrigez directement sur le code).

(3) Pour tester les colonnes, on écrit d'abord une fonction permettant **d'extraire** une colonne de la grille sous forme d'un tableau à une dimension. Complétez la fonction ci-dessous dont on vous fournit la documentation et les tests.

```
/** Extraction d'une colonne
 * Renvoie la colonne j sous forme d'un tableau 1D
 * @param grille une grille de puissance 4
 * @param j l'indice de la collone à extraire
 * @return la colonne d'indice j sous forme d'un vector
 **/
```

```
CHECK( extractionColonne(grille, 0) == vector<int>({1, 1, 2, 2, 2, 1}));
CHECK( extractionColonne(grille, 1) == vector<int>({0, 1, 1, 1, 2, 2}));
CHECK( extractionColonne(grille, 2) == vector<int>({0, 0, 1, 1, 1, 2}));
CHECK( extractionColonne(grille, 3) == vector<int>({0, 2, 2, 1, 2, 1}));
CHECK( extractionColonne(grille, 4) == vector<int>({0, 0, 0, 1, 2, 2}));
CHECK( extractionColonne(grille, 5) == vector<int>({2, 2, 2, 2, 1, 1}));
CHECK( extractionColonne(grille, 6) == vector<int>({1, 2, 2, 1, 2, 1}));
```

PLACE :

1	0	**0**	0	0	2	1
1	1	0	**2**	0	2	2
2	1	1	2	**0**	2	2
2	1	1	1	1	**2**	1
2	2	1	2	2	1	**2**
1	2	2	1	2	1	1

(4) ♣ Implantez la fonction suivante permettant d'extraire une diagonale en allant vers le bas à droite à partir d'une case de coordonnées données. Par exemple, dans la figure à droite, la diagonale de notre grille commençant par la case de coordonnées (0,2) est mise en gras.

```
/** Extraction des diagonales vers bas-droite
 * Extrait les valeurs de la grille en partant des indices
 * i et j et en descendant en diagonale vers la droite
 * @param grille une grille de puissance 4
 * @param i,j  les indices de la ligne et colonne de départ
 * @return la diagonale sous forme de tableau 1D
 **/
```

```
CHECK(extractionDiagonale(grille, 0, 0) == vector<int>({1, 1, 1, 1, 2, 1}));
CHECK(extractionDiagonale(grille, 0, 1) == vector<int>({0, 0, 2, 1, 1, 1}));
CHECK(extractionDiagonale(grille, 0, 2) == vector<int>({0, 2, 0, 2, 2}));
CHECK(extractionDiagonale(grille, 0, 3) == vector<int>({0, 0, 2, 1}));
CHECK(extractionDiagonale(grille, 1, 0) == vector<int>({1, 1, 1, 2, 2}));
CHECK(extractionDiagonale(grille, 2, 0) == vector<int>({2, 1, 1, 1}));
```

Utilisez cet encadré si vous avez manqué de place ailleurs dans l'examen.