Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2022-2023**
Examen du 12 décembre 2022 (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 réponses sont à donner, autant que possible, sur le sujet ; sinon, demander un intercalaire et mettre un renvoi.
Les enseignants collecteront votre copie à votre place.

Exercice 1 (Question de cours : fichiers (10 points))**.**

(1) Écrivez un fragment de programme C++ qui ouvre un fichier `toto.txt` en lecture :

Correction :

```
ifstream fichier;
fichier.("toto.txt");
```

(2) Écrivez le code pour afficher un message d'erreur si l'ouverture du fichier a échoué (♣ ou mieux, déclencher une exception !)

Correction :

```
if ( not fichier )
    throw runtime_error("Le fichier n'a pas pu être ouvert");
```

(3) On suppose que `toto.txt` ne contient que des entiers, séparés par des espaces. Écrivez le code pour compter le nombre d'entiers dans le fichier. Bien fermer le fichier.

Correction :

```
int i, compteur;
while ( fichier >> i )
    compteur ++;
fichier.close();
```

(4) Donnez la définition de flux entrant.

Correction : Un flux entrant est un dispositif où l'on peut lire des données, successivement l'une après l'autre.

Exercice 2 (Exponentiation rapide : fonctions, tests, documentation, boucles (18 points))**.**

(1) Spécifiez (par une documentation) une fonction `puissanceNaive` qui prend en entrée deux entiers positifs x et n et qui calcule x^n.

```
/** Fonction puissanceNaive
 *  @param x un nombre entier
 *  @param n un nombre entier positif
 *  @return n-ième puissance x^n de x
 **/
```

(2) Donnez un exemple d'utilisation de cette fonction déclarant une nouvelle variable p, calculant 3^4 et affectant le résultat à p :

Correction :

```
int p = puissanceNaive(3, 4);
```

(3) Implantez cette fonction.

```
int puissanceNaive( int x, int n ) {
    int y = 1;

    for ( int k = 1; k <= n; k++ ) {
        y = y * x;
    }

    return y;
}
```

Dans une bibliothèque, vous trouvez la fonction `puissanceRapide` suivante, avec la même spécification que pour `puissanceNaive` :

```
1  int puissanceRapide( int x, int n ) {
2      int y = 1;
3      while ( n > 0 ){
4          if ( n % 2 == 0 ) {
5              x = x * x;
6              n = n / 2;
7          } else {
8              y = y * x;
9              n = n - 1;
10             return y;
11         }
12     }
13 }
```

Cette fonction est munie des tests suivants :

```
CHECK( puissanceRapide( 2, 2 ) ==  4 );
CHECK( puissanceRapide( 2, 3 ) ==  8 );
```

(4) Les tests et la spécification sont ils cohérents entre eux ?

Correction : oui.

(5) Exécutez pas à pas l'appel à `puissanceRapide(2,2)` du premier test :

x	y	n
2	1	2
4	1	1
4	4	0

Donnez la valeur renvoyée : 4

Le test passe-t'il ? oui

(6) Même question pour l'appel à `puissanceRapide(2,3)` du second test :

x	y	n
2	1	3
2	2	2

Donnez la valeur renvoyée : 2

Le test passe-t'il ? non

(7) Proposez deux nouveaux tests pour cette fonction, tous deux cohérents avec la spécifications, dont le premier passe et le second ne passe pas.

```
CHECK( puissanceRapide( 2, 1 ) ==  2 ); //bon
CHECK( puissanceRapide( 2, 5 ) == 32 ); //mauvais
```

(8) Quelle(s) ligne(s) de la fonction `puissanceRapide` faut-il modifier pour qu'elle corresponde à sa spécification ?

Correction : Déplacer le `return y` de la ligne 11 à la ligne 14.

Exercice 3 (Tableaux à une ou deux dimensions, fichiers (20 points))**.**

(1) (4 points) Écrivez le code d'une fonction `affiche` qui prend un tableau de mots en paramètres et affiche ces mots séparés par des espaces.

Correction :

```
void affiche(vector<string> l) {
    for (int numColonne = 0; numColonne < l.size(); numColonne++) {
        if ( numColonne > 0 )
            cout << " ";
        cout << l[numColonne];
    }
}
```

On supposera maintenant définis la fonction `diagonale` documentée comme suit :

```
/** renvoie la diagonale d'un tableau 2D.
* @param t tableau carré à deux dimensions contennant des mots.
* @return un tableau à une dimension contenant une copie des mots
* de la première diagonale du tableau à deux dimensions.
*/ vector<string> diagonale(vector<vector<string>> t);
```

ainsi que le tableau `tdiago` ci-dessous :

```
vector<vector<string>> tdiago = { {"ici", "ouf", "une", "rue", "boa"},
    {"bon", "ton", "est", "non", "oui"}, {"raz", "ile", "job", "epi", "net"},
    {"lac", "rez", "sas", "est", "but"}, {"des", "des", "ont", "pas", "bon"}
};
```

(2) (2 points) On considère l'expression :

```
affiche(diagonale(tdiago));
```

Indiquez ce qui est renvoyé et ce qui est affiché par l'évaluation de cette expression :

Correction : Cet appel ne renvoie rien.

Cet appel affiche : `ici ton job est bon`

(3) (4 points) En vous inspirant de l'exemple précédent, donnez un test pour la fonction `diagonale` :

Correction :

```
CHECK( diagonale(tdiago) ==
        vector<string>({ "ici", "ton", "job", "est", "bon"}));
```

La fonction `nnMotsLongueurEtalon` prend en paramètres `nomFichier`, le nom d'un fichier contenant des mots de différentes longueurs, n, un entier et `etalon`, un mot. On note k la longueur de ce mot. La fonction construit un tableau carré de taille n par n initialisé avec les $n \times n$ premiers mots de longueur k trouvés dans le fichier ; si le fichier ne contient pas assez de mots de longueur k, alors la fin du tableau est initialisée avec le mot `etalon`. Enfin, la fonction renvoie ce tableau.

(4) (10 points) Écrivez le code de la fonction `nnMotsLongueurEtalon`.

Correction :

```
vector<vector<string>> nnMotsLongueurEtalon( string nomFichier, int n, string e
    ifstream flux;          // déclaration du flux
    flux.open(nomFichier) ;  // ouverture du fichier
    string mot;
    // declaration
    vector<vector<string>> t;
    // allocation
    t = vector<vector<string>>(n); // allocation
    for (int numLigne = 0; numLigne < n; numLigne++) {
        // allocation des sous-tableaux
        t[numLigne] = vector<string>(n);
        // initialisation
        for ( int j = 0; j < n; j++ ) {
            while ( ( flux >> mot ) and ( mot.size()  != etalon.size() ) );
            if ( flux )
                t[numLigne][j] = mot;
            else
                t[numLigne][j] = etalon;
        }
    }
    flux.close();          // fermeture du fichier
    return t;
}
```

Exercice 4 (Pile, tas, tableaux 2D (12 points))**.**
On considère le fragment de programme suivant :

```
int main() {
    vector<vector<int>> t;
    t = vector<vector<int>>(3);
    t[0] = { 0, 1, 2, 3 };
    t[1] = vector<int>(3);
    vector<int> c = t[0];
    t[0][2] = 5;
    // ICI
    t[2][0] = 5;
    return 0;
}
```

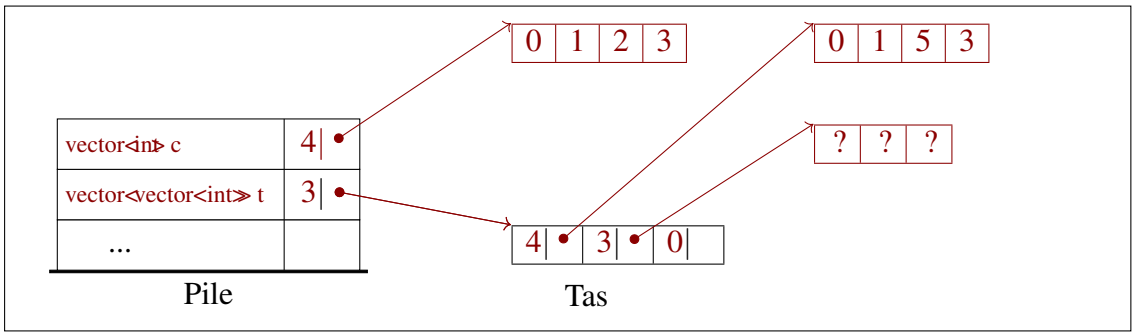
(1) Annotez le programme ci-dessus avec, pour chaque ligne du programme contribuant à la construction du tableau 2D `t`, l'étape(s) de la construction à laquelle elle contribue.

(2) Que manque-t'il pour que la construction soit complète ?

Correction : (a) Le sous-tableau `t[1]` n'est pas complètement initialisé.

(b) Le sous-tableau `t[2]` n'est pas alloué.

(3) Sur votre brouillon, exécutez pas-à-pas le programme. En suivant les conventions du cours, complétez ci-dessous le dessin de la pile et du tas au moment où l'exécution atteint la ligne marquée `ICI`.



(4) Quel est le comportement du programme lors de l'exécution du la ligne suivante ? Pourquoi ?

Correction : Le comportement du programme est indéfini dans la norme C++ ; dans le meilleur des cas, il y aura une erreur de segmentation : en effet, `t[2][0]` accède à la première case d'un tableau qui n'a pas été alloué (plus précisément, avec les `vector` de C++ : qui a été alloué par défaut avec une taille de 0).

(5) Par quoi pourrait-on remplacer `t[2][0]` pour que le problème soit signalé par une exception ?

Correction : `t.at(2).at(0)`

Exercice 5 (Sudoku (40 points))**.**
Le but du Sudoku est de compléter une grille de 81 cases (9x9) avec des chiffres de 1 à 9. Cette grille est divisée en 9 blocs de 3x3. Le joueur doit compléter la grille avec la contrainte de ne jamais avoir deux fois le même chiffre sur une ligne, une colonne ou un bloc de 3x3.
Au départ, la grille de Sudoku est partiellement remplie. La Figure 1 présente un exemple de Sudoku à remplir.

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

FIGURE 1 – Un exemple de Sudoku partiellement rempli en début de jeu

Dans ce problème, on s'intéresse à l'écriture d'un programme permettant de compléter une grille de Sudoku en respectant les règles de jeu. On représente une telle grille par un tableau 2D d'entiers, les cases vides étant encodées par l'entier 0.

(1) **Validité d'une grille** :

(a) Implantez une fonction `elementsDeBloc` qui prend en paramètres une grille de Sudoku, ainsi que les indices de la ligne et de la colonne d'un bloc de 3x3 (0, 1 ou 2), et qui renvoie les entiers **non-nuls** contenus dans ce bloc. Ainsi, si l'on prend l'exemple de la Figure 1, l'appel de `elementsDeBloc(grille, 2, 0)` doit renvoyer le tableau `{7, 9, 3}`.

```
/** Renvoie les valeurs non-nulles d'un bloc d'indice
 * ligne et colonne
 * @param grille une grille de sudoku
 * @param indice de la ligne du bloc (entre 0 et 2)
 * @param indice de la colonne du bloc (entre 0 et 2)
 * @return un tableau d'entier
 **/
vector<int> elementsDeBloc(Grille g, int ligne, int colonne) {
    vector<int> res;
    int element;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            element = g[ligne * 3 + i][colonne * 3 + j];
            if (element != 0) {
                res.push_back(element);
            }
        }
    }
    return res;
}
```

(b) On considère qu'il existe une fonction `contient` qui prend en paramètres un entier et un tableau d'entiers et renvoie `true` si l'entier est présent dans le tableau, `false` sinon. Vous avez déjà implanté une fonction très similaire pendant le semestre, lors du

TP sur le jeu de Yams. Implantez la fonction `sansDoublon` ci-dessous, (en utilisant la fonction `contient` si vous le souhaitez) :

```
/** Teste si un tableau d'entiers n'a pas de doublons entre 1 et 9
 * NB : on ne considère pas les doublons de 0
 * @param un tableau d'entier
 * @return blooléen: True si contient un doublon, False sinon
 **/
bool sansDoublon(vector<int> t) {
    vector<int> vus = {};
    for (int i = 0; i < t.size(); i++) {
        if (contient(t[i], vus)) {
            return false;
        }
        if (t[i] != 0) {
            vus.push_back(t[i]);
        }
    }
    return true;
}
```

(c) Écrivez au moins trois tests pertinents de la fonction `sansDoublon` :

```
void sansDoublonTest() {
    CHECK(sansDoublon({0, 1, 2, 3, 0}));
    CHECK(not sansDoublon({0, 1, 0, 2, 2, 0}));
    CHECK(sansDoublon({0, 0, 0}));
    CHECK(sansDoublon({}));
}
```

(d) De manière analogue à la fonction `elementsDeBloc`, on suppose que l'on a des fonctions `elementsDeLigne` et `elementsDeColonne` qui extraient les entiers non-nuls contenus dans une ligne (resp. une colonne) d'indice entre 0 et 8 donné. En réutilisant les fonctions des questions (a) et (b), spécifiez (par une documentaion) et écrivez une fonction `estValide` qui prend en paramètre une grille quelconque (remplie ou partiellement remplie) et renvoie `true` si la grille est valide c'est à dire qu'elle ne présente pas de doublons sur ses lignes, ses colonnes et ses blocs.

```
/** Teste si une grille quelconque est valide c'est-à-dire qu'elle respecte les co
 * @param grille une grille de sudoku
 * @return un booléen
 **/
```

```
bool estValide(Grille g) {
    for (int i = 0; i < 9; i++) {
        if (sansDoublon(elementsDeLigne(g, i)) == false or
            sansDoublon(elementsDeColonne(g, i)) == false) {
            return false;
        }
    }
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (sansDoublon(elementsDeBloc(g, i, j)) == false) {
                return false;
            }
        }
    }
    return true;
}
```

(2) **Vers la recherche automatique de solution** :

Nous nous intéressons à présent au processus de remplissage des cases en vue d'automatiser la recherche de solution.

(a) Implantez une fonction `valeursPossibles` qui prend en paramètres une grille et les indices d'une case de la grille et renvoie les valeurs compatibles pour cette case c'est à dire celles ne créant pas de doublons avec la ligne, la colonne ou le bloc considéré. **Indice :** pensez à réutiliser les fonctions `contient`, `elementsDeLigne`, `elementsDeColonne` et `elementsDeBloc`.

```
vector<int> valeursPossibles(Grille g, int i, int j) {
    vector<int> res;
    for (int k = 1; k <= 9; k++) {
        if (not (contient(k, elementsDeLigne(g, i)) or
                 contient(k, elementsDeColonne(g, j)) or
                 contient(k, elementsDeBloc(g, i / 3, j / 3)) ) ) {
            res.push_back(k);
        }

    }
    return res;
}
```

(b) On considère la fonction `resolutionNaive` qui, à partir d'une grille, remplit tour à tour les cases vides n'ayant qu'une seule valeur possible et renvoie une grille vide si la complétion n'est pas possible. Autrement dit, la fonction `resolutionNaive` renvoie un Sudoku complété au maximum permis par l'algorithme ou une grille vide si le Sudoku n'adment pas de solution. La fonction `resolutionNaiveBoguee` suivante implante cet algorithme, mais contient quatre bogues. On suppose que la fonction `grilleVide` qui ne prend aucune entrée et renvoie une grille vide a déjà été implémentée.

Pour chaque bogue, **identifiez le numéro de la ligne fautive, décrivez le problème et proposez une correction**.

```
1  Grille resolutionNaiveBoguee(Grille grille) {
2      bool aChange = false;
3      while ( aChange ) {
4          aChange = false;
5          for (int ligne = 0; ligne < 9; ligne++) {
6              for (int colonne = 0; colonne > 9; colonne++) {
7                  if ( grille[ligne][colonne] = 0 ) {
8                      vector<int> vals =
9                          valeursPossibles(grille, ligne, colonne);
10                     if ( vals.size() == 0 ) {
11                         return grilleVide();
12                     }
13                     if ( vals.size() == 1 ) {
14                         grille[ligne][colonne] == vals[0];
15                         aChange = true;
16                     }
17                 }
18             }
19         }
20     }
21     return grille;
22 }
```

(i) Ligne 2 : on n'entre jamais dans la boucle ;
correction : `bool aChange = true ;`

(ii) Ligne 6 : inégalité inversée ;
correction : `colonne < 9`

(iii) Ligne 7 : affectation au lieu d'un test d'égalité
correction : `if (grille[ligne][colonne] == 0)`

(iv) Ligne 14 : test d'égalité au lieu d'affectation ;
correction : `grille[ligne][colonne] = vals[0];`

(c) ♣ Proposez un algorithme permettant de déterminer toutes les solutions d'un problème de Sudoku. On pourra se contenter d'en donner les grandes lignes.

Correction : On considère la grille obtenue par complétion naïve. On choisit une position avec plusieurs valeurs possibles, et on essaie récursivement de compléter la grille en choisissant tour à tour ces valeurs.