Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2022-2023**
Examen mi-semestre du vendredi 28 octobre 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 (Cours (10 points))**.**

(1) Rappelez la syntaxe (3 points) et la sémantique (4 points) de la boucle `for` en C++.

Syntaxe :

Correction :

```
for ( initialisation ; condition ; incrementation ) {
   bloc d'instructions
}
```

Sémantique :

Correction :

(a) Exécution de l'instruction d'initialisation

(b) Évaluation de la condition

(c) Si sa valeur est `true` :

(i) Exécution du bloc d'instruction

(ii) Exécution de l'instruction d'incrémentation

(iii) On recommence en 1b

(2) Dans quel cas faut-il préférer une boucle `for` à une boucle `while` ? Pourquoi ? (3 points)

Correction : On préfère la version « for » pour une boucle basée sur l'utilisation d'un **compteur**. En effet la boucle « for » exprime l'intention d'utiliser un compteur. De plus, toute la gestion du compteur est centralisée sur une seule ligne, ce qui est plus lisible.

Exercice 2 (Fonctions simples (10 points))**.**

(1) Spécifiez (sous forme de documentation) et implantez (sous forme de code) une fonction `sommeTroisEntiers` qui prend en paramètres trois entiers et renvoie leur somme.

Correction :

```
/** calcule la somme de trois entiers
 * @param a un entier
 * @param b un entier
 * @param c un entier
 * @return la somme de ces entiers
**/
int sommeTroisEntiers(int a, int b, int c) {
    return a + b + c;
}
```

(2) Écrivez un test permettant de vérifier le bon fonctionnement de `sommeTroisEntiers`.

Correction :

```
CHECK( sommeTroisEntiers(4, 2, 9) == 15 );
```

(3) Utilisation : écrivez les lignes de code qui permettent de déclarer et initialiser trois variables avec les valeurs de votre choix pour représenter les trois entiers, puis de stocker leur somme dans une nouvelle variable. Il est obligatoire d'utiliser un **appel** à votre fonction `sommeTroisEntiers`.

Correction :

```
int p = 3;
int q = 7;
int r = 1;
int resultat = sommeTroisEntiers(p, q, r);
```

Exercice 3 (Booléens (8 points))**.**

(1) Implantez la fonction dont la documentation et les tests sont donnés ci-dessous :

```
/** Fonction multipleAvecContrainte
 *  @param a un entier
 *  @param b un entier
 *  @return true si a est un multiple de b et l'un des
        deux entiers est supérieur ou égal à 10, false sinon
 **/
```

```
CHECK( multipleAvecContrainte(50, 5) );
CHECK( multipleAvecContrainte(370, 10) );
CHECK( not multipleAvecContrainte(8, 2) );
CHECK( not multipleAvecContrainte(7, 49) );
```

Correction :

```
bool multipleAvecContrainteNaif(int a, int b) {
    if ((a % b == 0) and (a >= 10 or b >= 10)) {
        return true;
    } else {
        return false;
    }
}
```

(2) Si ce n'est pas déjà le cas, ré-implantez votre fonction pour qu'elle ne contienne ni`if` ni aucune autre structure de contrôle :

Correction :

```
bool multipleAvecContrainte(int a, int b) {
    return ((a % b == 0) and (a >= 10 or b >= 10));
}
```

Exercice 4 (Boucles (12 points))**.**

(1) Écrivez le code permettant d'afficher tous les nombres impairs compris entre 0 et 100.

Correction :

```
for ( int i = 0; i <= 100; i++ ) {
    if ( i % 2 == 1)
        cout << i << endl;
}
```

Dans les deux questions suivantes, vous devez utiliser des appels à la fonction `puissance` dont la documentation est donnée ci-dessous :

```
/** La fonction puissance
 *  @param x un nombre entier
 *  @param n un nombre entier positif
 *  @return la n-ième puissance x^n de x
**/
```

Cette fonction est considérée déjà définie, vous n'avez pas à écrire son code.

(2) Implantez une fonction `sommePuissancesImpaires` qui prend en paramètre un entier n et un entier k et qui renvoie la somme des k premières puissances impaires de n. Par exemple, pour $k = 3$, la fonction doit renvoyer $n^1 + n^3 + n^5$.

Correction :

```
int sommePuissancesImpaires(int n, int k) {
    int resultat = 0;
    for ( int i = 1; i <= k; i++ ) {
        resultat = resultat + puissance(n, 2*i-1);
    }
    return resultat;
}
```

(3) Implantez une fonction `plusGrandExposant` qui prend en paramètres deux entiers $n > 1$ et $m \geq 1$ et qui renvoie le plus grand entier $k \geq 0$ tel que $n^k \leq m$

Correction :

```
int plusGrandExposant(int n, int m) {
    for ( int k = 0; puissance(n, k) <= m; k++ ) {
        k++;
    }
    return puissance(n, k-1);
}
```

Exercice 5 (Tableaux (10 points))**.**

(1) Implantez une fonction nommée `majore` qui prend en paramètres un tableau d'entiers `t` et un entier `m` et qui renvoie `true` si `m` est *strictement* plus grand que tous les éléments de `t` et `false` sinon (sans utiliser de boucle *pour tout*). La fonction devra passer les tests suivants :

```
CHECK(    majore( { 1,2,  3}, 5 ) );
CHECK( not majore( { 4,5,  6}, 2 ) );
CHECK(    majore( {10, 46, 32}, 50 ) );
```

```
bool majore(vector<int> t, int m) {
    for ( int i = 0; i < t.size(); i++ ) {
        if ( t[i] >= m )
            return false;
    }
    return true;
}
```

(2) ♣ Réimplantez la fonction précédente en utilisant une boucle *pour tout* `for (int v:t){...}`.

```
bool majorePourTout(vector<int> t, int m) {
    for ( int v: t ) {
        if ( v >= m )
            return false;
    }
    return true;
}
```

(3) Implantez une fonction nommée `positionImpair` qui prend en paramètre un tableau d'entiers `t`, et qui renvoie un nouveau tableau contenant les indices des éléments impairs de `t`. La fonction devra passer les tests suivants :

```
CHECK( positionImpair( {1, 2, 3} ) == {0, 2} );
CHECK( positionImpair( {8, 2, 4} ) == {} );
```

```
vector<int> positionImpair(vector<int> t) {
    vector<int> positions = {};
    for ( int i = 0; i < t.size(); i++) {
        if (t[i] % 2 == 1)
            positions.push_back(i);
    }
    return positions;
}
```

Exercice 6 (Pile et Tas (12 points))**.**
On considère le fragment de programme suivant :

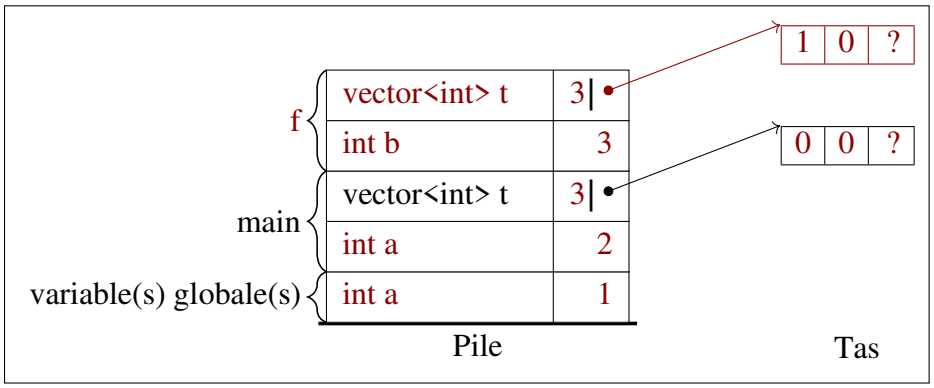
```
int a = 0;

int f(int b, vector<int> t) {
    a = a + 1;
    b = b + 1;
    t[0] = a;
    // ICI
    return b;
}

int main() {
    int a = 2;
    vector<int> t;
    t = vector<int>(3);
    t[0] = 0;
    t[1] = 0;
    t[2] = f(a, t);
    // LÀ
    cout << a << " " << t[2] << endl;
    return 0;
}
```

(1) Soulignez la ou les déclaration(s) de paramètre(s) formel(s).

(2) Encadrez d'un rectangle la ou les déclaration(s) de variable(s) locale(s).

(3) Entourez d'un rond la ou les déclaration(s) de variable(s) globale(s).

(4) 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`.



(5) Quelles sont les valeurs des variables suivantes au moment où l'exécution du programme atteint la ligne marquée `LÀ` ?
— variable globale `a` : 1
— variable locale `a` : 2
— variable locale `t` : `{0, 0, 3}`

Exercice 7 (Jeu de Nim (38 points))**.**
Toutes les questions sont indépendantes

Le jeu de Nim est un jeu à deux joueurs qui se joue avec des jetons. Avant de commencer la partie, on répartit les jetons en plusieurs tas (on peut en faire autant que l'on veut). Les joueurs jouent à tour de rôle. Chaque joueur doit à son tour choisir un tas qui contient des jetons et peut décider d'en retirer autant qu'il veut de ce tas (au moins un jeton à chaque fois). Le joueur qui retire le dernier jeton a gagné.

On représente un état du jeu de Nim par un tableau d'entiers, chaque case du tableau représentant le nombre de jetons dans un tas (on numérote donc les tas en commençant à zéro).

(1) Implantez la fonction `afficheNim` qui prend en paramètre un état du jeu de Nim et affiche le nombre de jetons dans chaque tas de sorte que les nombres de jetons soient écrits sur une ligne et séparés par le signe "-", comme dans l'exemple ci-dessous.

```
afficheNim({2, 0, 3, 4});
```

```
2-0-3-4
```

```
void afficheNim(vector<int> etat) {
    for ( int i = 0; i < etat.size()-1; i++ ) {
        cout << etat[i] << "-";
    }
    cout << etat[etat.size()-1] << endl;
}
```

(2) La partie est finie lorsque tous les tas sont vides. Complétez la fonction suivante dont on vous donne la documentation et les tests.

```
/** Teste si le jeu est terminé
 * @param etat, un tableau d'entiers représentant l'état actuel du j
 * @return True si le tableau ne contient plus de jetons, False sinc
 */
bool fini(vector<int> etat) {

    for ( int i = 0; i < etat.size(); i++ ) {
        if ( etat[i] > 0 ) {
            return false;
        }
    }
    return true;

}
```

```
CHECK( fini({0, 0, 0}) );
CHECK( not fini({0, 1, 3, 0}));
```

(3) À chaque tour de, le joueur choisit un tas et retire au moins un jeton de ce tas. Implantez la fonction `tour` spécifiée par la documentation et les tests suivants.

```
/** Jouer un tour au jeu de Nim
 * @param etat un tableau d'entiers représentant l'état actuel du jeu
 * @param tas, le numéro du tas à modifier
 * @param nbJetons, le nombre de jetons à retirer du tas
 * @return l'état du jeu après avoir retiré les jetons du tas
 *         correspondant si le coup est valide (numéro de tas
 *         et nombre de jetons à retirer valides); sinon
 *         renvoie l'état du jeu non modifié.
**/

vector<int> tour(vector<int> etat, int tas, int nbJetons) {
    if ( tas >= etat.size() or tas < 0
            or nbJetons <= 0 or etat[tas] < nbJetons) {
        return etat;
    }
    etat[tas] -= nbJetons;
    return etat;
}
```

```
/* test des coups invalides */
CHECK( tour({0, 0}, 0, 1) == vector<int>({0, 0}) );
CHECK( tour({2, 1, 3}, 4, 1) == vector<int>({2, 1, 3}) );
CHECK( tour({2, 1, 3}, -1, 1) == vector<int>({2, 1, 3}) );
CHECK( tour({2, 1, 3}, 1, -1) == vector<int>({2, 1, 3}) );
/* test des coups valides */
CHECK( tour({2, 1, 3, 4}, 2, 2) == vector<int>({2, 1, 1, 4}) );
CHECK( tour({1, 2, 3}, 0, 1) == vector<int>({0, 2, 3}) );
```

(4) On souhaite à présent tester différentes stratégies de jeu. On va pour cela implanter des fonctions `strategie` qui prennent en paramètre l'état du jeu et renvoient un coup possible à jouer sous forme de tableau à deux valeurs : la première case du tableau est un numéro de tas et la deuxième case un nombre de jetons.

On va se contenter d'une stratégie naïve (et mauvaise) qui consiste à choisir le tas avec le maximum de jetons et à les prendre tous. Implantez la fonction `strategieMax` dont on vous donne la documentation et les tests :

```
/** strategieMax
 * @param etat un tableau d'entiers représentant l'état actuel du jeu
 * @return un tableau d'entiers de taille 2 donnant le coup à jouer
 *         la première case est le numéro de tas avec le plus de jetons
 *         la deuxième case est le nombre de jetons de ce tas
**/
```

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

```
vector<int> strategieMax(vector<int> etat) {

    int tasMax = 0;
    for( int i = 0; i < etat.size(); i++ ) {
        if ( etat[i] > etat[tasMax] ) {
            tasMax = i;
        }
    }
    return {tasMax, etat[tasMax]};

}
```

(5) Observez la fonction suivante.

```
int partie(vector<int> etat) {
    int joueur = 2;
    while ( not fini(etat) ) {
        if ( joueur == 1 ) {
            joueur = 2;
        } else {
            joueur = 1;
        }
        vector<int> strat = strategieMax(etat);
        etat = tour(etat, strat[0], strat[1]);
    }
    return joueur;
}
```

(a) Exécutez pas à pas le programme `partie({3,2,1})`.

(b) Écrivez la documentation de la fonction

Correction :

```
/** Une partie complète selon la strategieMax
 * @param etat un tableau d'entiers représentant l'état de départ
 * @return numéro du joueur qui gagne la partie
**/
```

(c) Écrivez le test correspondant au tableau `{3,2,1}` ainsi qu'un second test avec une valeur de retour différente.

Correction :

```
CHECK( partie({3,2,1}) == 1);
CHECK( partie({3,4}) == 2);
```

(6) ♣ Il existe une variante du jeu de Nim qui s'appelle le jeu de Grundy. Cette fois, on choisit un tas que l'on **sépare** en deux tas de **tailles différentes strictement positives**. Par exemple, si je pars avec les tas `2,3,4` je peux choisir de séparer le tas de taille 3 en `1 + 2` et j'obtiens `2,1,2,4`. Le jeu se termine quand on ne peut plus séparer aucun tas.

Implantez la fonction `tourGrundy` dont on vous donne la documentation et les tests. Attention, l'ordre des tas doit être respecté (le nouveau tas se place juste après le tas dont il a été séparé). Tout comme pour la fonction `tour`, si le coup joué est invalide, alors l'état du jeu n'est pas modifié.

```
/** Jouer un tour au jeu de Grundy
 * @param etat un tableau d'entiers représentant l'état actuel du jeu
 * @param tas le numéro du tas à modifier
 * @param nbJetons le nombre de jetons à séparer
 * @return l'état de jeu après avoir séparé le tas choisi
**/

vector<int> tourGrundy(vector<int> etat, int tas, int nbJetons) {
    if ( tas >= etat.size() or tas < 0
            or nbJetons <= 0 or etat[tas] <= nbJetons
            or etat[tas] - nbJetons == nbJetons) {
        return etat;
    }
    vector<int> nouvelEtat(etat.size()+ 1 );
    for (int i = 0; i < tas ; i++){
        nouvelEtat[i] = etat[i];
    }
    nouvelEtat[tas] = etat[tas]-nbJetons;
    nouvelEtat[tas+1] = nbJetons;
    for (int i = tas+1; i < etat.size() ; i++){
        nouvelEtat[i+1] = etat[i];
    }
    return nouvelEtat;
}
```

```
/* test des coups invalides */
CHECK( tourGrundy({1, 2, 3}, 0, 1) == vector<int>({1, 2, 3}) );
CHECK( tourGrundy({1, 2, 3}, 1, 1) == vector<int>({1, 2, 3}) );
CHECK( tourGrundy({1, 4, 3}, 1, 2) == vector<int>({1, 4, 3}) );
CHECK( tourGrundy({1, 2, 3}, 4, 1) == vector<int>({1, 2, 3}) );
CHECK( tourGrundy({1, 2, 3}, -1, 1) == vector<int>({1, 2, 3}) );
CHECK( tourGrundy({1, 2, 3}, 2, -1) == vector<int>({1, 2, 3}) );

/* test des coups valides */
CHECK( tourGrundy({2, 1, 3, 4}, 2, 1) == vector<int>({2, 1, 2, 1, 4}) );
CHECK( tourGrundy({2, 1, 3, 4}, 3, 1) == vector<int>({2, 1, 3, 3, 1}) );
CHECK( tourGrundy({2, 3, 4}, 1, 2) == vector<int>({2, 1, 2, 4}) );
```