Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2021-2022**
Examen mi-semestre du mercredi 27 octobre 2021 (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). (1) Quelles sont les trois étapes pour construire un tableau en C++ ?

Correction : Un tableau se construit en trois étapes :

(a) Déclaration

(b) Allocation

(c) Initialisation

(2) Illustrer votre réponse avec la construction d'un tableau de 100 entiers tous égaux à 1 :

Correction :

```
vector<int> t;
t = vector<int>(100);
for ( int i = 0; i < 100; i++ ) {
    t[i] = 1;
}
```

(3) Quelle(s) conséquence(s) si on oublie la première étape ?

Correction : Le compilateur donnera une erreur : le tableau n'est pas déclaré.

(4) Quelle(s) conséquence(s) si on oublie la deuxième étape ?

Correction : Les conséquences ne sont pas définies par la norme. Au mieux, le programme s'arrête avec une erreur de Segmentation. Au pire, la mémoire est corrompue.

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

(1) Écrire une fonction `aireRectangle` qui prend en paramètres deux entiers représentant la longueur et la largeur d'un rectangle, et renvoie son aire. Précéder la fonction de sa documentation, indiquant ses paramètres et ce qu'elle renvoie.

Correction :

```
/** calcule l'aire d'un rectangle
@param a longueur du rectangle
@param b largeur du rectangle
@return aire du rectangle
**/
int aireRectangle(int a, int b) {
    return a*b;
}
```

(2) Écrire un test permettant de vérifier le bon fonctionnement de `aireRectangle`.

Correction :

```
    CHECK( aireRectangle(3, 5) == 15 );
```

(3) Utilisation : écrire les lignes de code qui permettent de déclarer et initialiser deux variables avec les valeurs de votre choix pour représenter les dimensions d'un rectangle, puis de stocker son aire dans une nouvelle variable. Il est obligatoire d'utiliser un **appel** à votre fonction `aireRectangle`.

Correction :

```
    int p = 2;
    int q = 8;
    int resultat = aireRectangle(p, q);
```

Exercice 3 (Booléens)**.**

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

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

```
    ASSERT( unPasLautre(15, 9) );
    ASSERT( unPasLautre(7, 23) );
    ASSERT( not unPasLautre(12, 16) );
    ASSERT( not unPasLautre(8, 6) );
```

Correction :

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

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

Correction :

```
bool unPasLautre(int a, int b) {
    return a >= 10 and b < 10 or a < 10 and b >= 10;
}
```

Exercice 4 (Boucles (15 points))**.**

(1) Écrire les lignes de code permettant d'afficher tous les multiples de 7 de 0 à 1000.

Correction :

```
    for ( int i = 0; i < 1000; i+=7){
        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 n un nombre entier
 *  @param p un nombre entier positif
 *  @return la p-ième puissance n^p de n
 **/
```

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

(2) Écrire une fonction nommée `sommePuissancePonderee` qui prend en paramètre un entier n et un entier p, et qui renvoie la somme des puissances k^e de n multipliées par k pour k allant de 1 à p. Par exemple, `sommePuissancePonderee(5, 4)` calcule : $1 \times 5^1 + 2 \times 5^2 + 3 \times 5^3 + 4 \times 5^4$.

Correction :

```
int sommePuissancePonderee(int n, int p){
    int resultat = 0;
    int puissance = 1;
    for ( int k = 1; k <= p; k++ ){
        resultat = resultat + k*puissance(n,k);
    }
    return resultat;
}
```

(3) Écrire une fonction nommée `affichePuissances` qui prend en paramètres un entier n et un entier p, et qui affiche les entiers k tels que $n^k \leq p$. Par exemple :
— `affichePuissances(2, 15)` affiche `0,1,2,3`
— `affichePuissances(2, 16)` affiche `0,1,2,3,4`
— `affichePuissances(5, 4)` affiche `0`
♣ : afficher les puissances sur une seule ligne et séparées par des virgules, comme ci-dessus.

Correction :

```
void affichePuissances(int n, int p){
    cout << 0;
    int k = 1;
    while (puissance(n, k) <= p){
        cout << "," << k;
        k++;
    }
    cout << endl;
}
```

Exercice 5 (Tableaux (12 points))**.**

(1) Écrire une fonction nommée `tousDifferents` qui prend en paramètre un tableau d'entiers `t` et un entier `m` et qui renvoie `true` si tous les éléments de `t` sont différents de `m`, `false` sinon (sans utiliser de boucle *for each*). La fonction devra passer les tests suivants :

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

```
    CHECK(     tousDifferents( {2, 8, -6}, 5 ) );
    CHECK( not tousDifferents( {2, 8, -6}, 2 ) );
    CHECK( not tousDifferents( {2, 0, 6}, 6 ) );
```

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

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

(3) Écrire une fonction nommée `sansPremier` qui prend en paramètre un tableau d'entiers `t`, et qui renvoie un nouveau tableau contenant une case de moins que `t`. Ce nouveau tableau doit contenir les mêmes éléments que `t`, sauf son premier élément. La fonction devra passer les tests suivants :

```
    CHECK( sansPremier( {4, 6, 3} ) ==  vector<int>({6, 3}) );
    CHECK( sansPremier( {7, 1, 15, -3} ) ==  vector<int>({1, 15, -3}) );
```

```
vector<int> sansPremier(vector<int> t) {
    vector<int> res;
    res = vector<int>(t.size()-1);
    for ( int i = 0; i < res.size(); i++ )
        res[i] = t[i+1];
    return res;
}
```

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

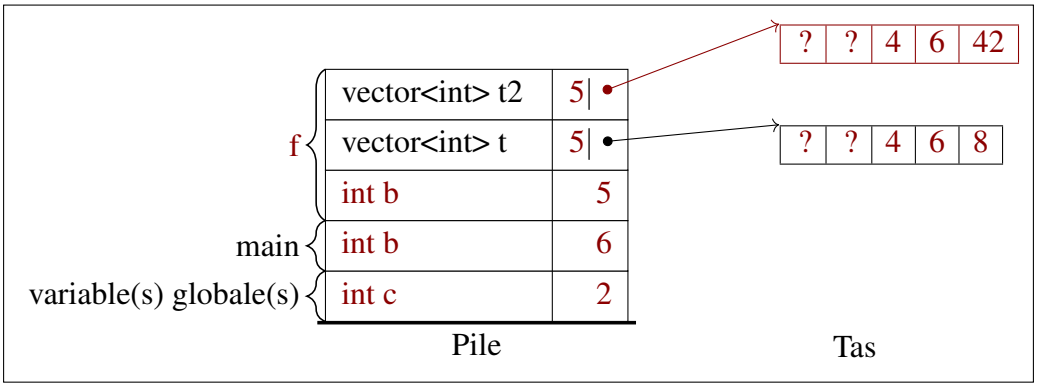
```
int c = 4;

int f(int b) {
    b = b - 1;
    vector<int> t;
    t = vector<int>(b);
    for ( int i = 2; i < t.size(); i++ ) {
        t[i] = c * i;
    }
    vector<int> t2;
    t2 = t;
    t2[4] = 42;
    // ICI
    return t[4];
}

int main() {
    int b = 6;
    c = c - 2;
    c = f(b);
    cout << c << 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) Quelle est la valeur de `c` à la fin de l'exécution du programme ?

Correction : 8

Exercice 7 (Bataille Fluviale (35 points))**.**
La Bataille Navale est un jeu de société où deux joueurs placent des bateaux sur une grille secrète et cherchent à couler ceux de l'adversaire. À tour de rôle, chaque joueur va choisir une position sur la grille. Si cette position correspond à un bateau adverse, ce navire sera touché. Si toutes les cases d'un navire ont été touchées, celui-ci est coulé. Le gagnant est le premier joueur qui réussit à couler toute la flotte de son adversaire.

On se propose d'implanter une variante de la Bataille Navale, la Bataille Fluviale, prenant place non pas sur une mer en deux dimensions mais sur une rivière en une dimension. Le but est de découvrir le plus rapidement possible l'ensemble des bateaux placés par l'ordinateur.

Dans cette implantation, la rivière sera représentée par un tableau d'entiers de taille 15. Dans ce tableau, un 0 symbolisera une position vide, et un entier i représentera une case du bateau i. Par simplicité, on considérera trois bateaux, de taille 2, 3 et 4 nommés respectivement 2, 3 et 4. Par exemple, le tableau ci-dessous représente une rivière où le bateau de longueur 4 commence à la position 1, le bateau de longueur 2 à la position 5 et le bateau de longueur 3 à la position 12.

0	4	4	4	4	2	2	0	0	0	0	0	3	3	3

(1) On s'intéresse tout d'abord au placement aléatoire de bateaux dans la rivière. Pour ceci, on souhaite implanter deux fonctions, `positionBateauValide` qui vérifiera si une certaine position permet d'accueillir un bateau dans la rivière, et `genereRiviere` qui s'occupera de générer une nouvelle rivière.

(a) Compléter le code de la fonction `positionBateauValide`, dont la documentation est donnée ci-dessous :

```
/** Vérifie si il est possible d'ajouter dans la rivière un bateau
 *  d'une certaine longueur à une certaine position
 * @param rivière un tableau d'entiers représentant la rivière
 * @param debut un entier représentant une position de la rivière
 * @param longueur un entier représentant la longueur du bateau
 * @return un booléen indiquant si l'ajout est possible ou non
 **/
```

```
bool positionBateauValide(vector<int> riviere, int debut, int longueur) {
    if (debut + longueur >= riviere.size())
        return  false ;
    bool valide =  true ;
    for (int i = debut; i < debut + longueur; i++) {
        if (riviere[i]  != 0)
            valide =  false ;
    }
    return valide;
}
```

(b) Écrire trois tests pour la fonction `positionBateauValide`, en considérant des rivières de taille 4 :

```
    CHECK(    positionBateauValide({0, 0, 0, 0}, 0, 4));
    CHECK(not positionBateauValide({0, 0, 0, 0}, 2, 3));
    CHECK(not positionBateauValide({0, 3, 3, 3}, 0, 2));
```

(c) Compléter le code de la fonction `genereriviere`, dont la documentation est donnée ci-dessous. On considère que l'on a déjà une fonction `int aleaint(int a, int b)` qui renvoie un entier aléatoire compris entre a et b inclus.

```
/** Génère aléatoirement un placement de bateaux de taille 2, 3
 *  et 4 dans une rivière de taille 15
 * @return un tableau d'entiers représentant la rivière
 **/
```

```
vector<int> genereRiviere() {
    vector<int> riviere;
    riviere = vector<int>(15);
    for (int i=2; i<=4; i++) {
        int pos;
        pos = aleaint(  0, 14  );
        while (  not positionBateauValide(riviere, pos, i) )
            pos = aleaint(  0, 14  );
        for (  int j = pos ; j < pos + i; j++ )
            riviere[j] = i;
    }
    return riviere;
}
```

(2) Afin de vérifier si une position a déjà été attaquée par le joueur, on souhaite implanter une fonction `tirDansHistorique`.

(a) Observez les tests suivants et déduisez-en la documentation (rôle, entrées et sortie) de la fonction `tirDansHistorique`. Écrire la documentation en dessous des tests :

```
    CHECK(    tirDansHistorique(0, {0, 2, 3, 5}));
    CHECK(not tirDansHistorique(1, {2, 4, 4})   );
    CHECK(    tirDansHistorique(2, {0, 2, 2, 5}));
    CHECK(not tirDansHistorique(3, {})          );
```

```
/** Verifie si une position a déjà été attaquée par le joueur
 * @param position un entier
 * @param historiqueTirs un tableau d entiers, l historique des positions attaquée
 * @return un booléen, vrai si la position a été attaquée par le joueur
 **/
```

(b) Implanter la fonction `tirDansHistorique` :

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

(3) Pour vérifier l'état d'avancement de la partie, on souhaite implanter deux fonctions, `afficheRiviere` qui affichera les bateaux découverts par le joueur, et `victoire` qui vérifiera si la partie est gagnée pour le joueur.

(a) Compléter le code de la fonction `afficheRiviere` dont la documentation est donnée

```
/** Affiche la rivière et les bateaux découverts par le joueur.
 * Les positions non dévoilées sont marquées par le symbole "-",
 * les bateaux dévoilés par leur numéro et les cases vides dévoilées
 * par le symbole "~".
 * @param rivière un tableau d'entiers, le placement des bateaux
 *        dans la rivière
 * @param historiqueTirs un tableau d'entiers, l'historique des
 *        positions attaquées par le joueur
 **/
```

```
void afficheRiviere(vector<int> riviere, vector<int> historiqueTirs) {
    for (int i=0; i<riviere.size(); i++) {
        if (  tirDansHistorique(i, historiqueTirs)  ) {
            if (  riviere[i] == 0  ) {
                cout << "~";
            } else {
                cout << riviere[i];
            }
        } else {
            cout << "-";
        }
    }
    cout << endl;
}
```

(b) À partir de la documentation de la fonction `victoire` donnée ci-dessous, écrire trois tests pour cette fonction, en considérant une rivière de taille 4 :

```
/** Vérifie si le joueur a gagné la partie
 * @param riviere un tableau d'entiers, le placement des bateaux
 *        dans la rivière
 * @param historiqueTirs un tableau d'entiers, l'historique des
 *        positions attaquées par le joueur
 * @return un booléen, vrai si toutes les positions de bateaux ont
 * été attaquées par le joueur
 **/
```

```
    CHECK(    victoire({1, 0, 2, 2}, {0, 2, 3}));
    CHECK(not victoire({2, 2, 0, 0}, {0, 2, 4}));
    CHECK(    victoire({0, 0, 0, 0}, {}) );
    CHECK(not victoire({4, 4, 4, 4}, {1, 2, 3}));
    CHECK(not victoire({1, 0, 0, 0}, {}) );
    CHECK(    victoire({0, 0, 0, 1}, {3}) );
```

(c) Implanter la fonction `victoire` :

```
bool victoire(vector<int> riviere, vector<int> historiqueTirs) {
    for (int i = 0; i < riviere.size(); i++) {
```

```
        if (riviere[i] != 0 and not tirDansHistorique(i, historiqueTirs))
            return false;
    }
    return true;
}
```

(4) On suppose déjà implantées les fonctions `entrerPositionTir` qui demande au joueur où il souhaite tirer dans la rivière et renvoie cette position, et `resultatTir` qui affiche si le dernier tir du joueur rate, touche un bateau, ou coule un bateau. Compléter le code de la fonction principale de la Bataille Fluviale présentée ci-dessous :

```
int main() {
    srand(time(0)); // Initialisation de la fonction rand

    vector<int> riviere =                    ;
    vector<int> historiqueTirs =             ;

    afficheRiviere(  riviere, historiqueTirs  );

    int tir;
    while (  not victoire(riviere, historiqueTirs) ) {
        tir = entrerPositionTir();
        resultatTir(tir, historiqueTirs, riviere);
        historiqueTirs.push_back(tir);
        afficheRiviere(  riviere, historiqueTirs  );
    }

    cout << "Félicitations, vous avez gagné en ";
    cout <<   historiqueTirs.size()  << " coups !" << endl;

    return 0;
}
```