TD 4 : Des fonctions, des tests et de la documentation

Exercice 1 (Premières fonctions)**.**
Voici une fonction qui calcule la surface d'un rectangle :

```
float surfaceRectangle(float longueur, float largeur) {
    return longueur * largeur;
}
```

(1) Implanter une **fonction** `surfaceDisque` qui calcule la surface d'un disque de rayon donné. On prendra $\pi = 3.1415926$.

(2) Implanter une **fonction** `surfaceTriangle` qui calcule la surface d'un triangle de base b et de hauteur h.

Exercice 2 (En route vers l'exponentielle)**.**

(1) Nous avons vu en cours et en TP une fonction `factorielle(n)` qui calcule la factorielle d'un entier positif n. Pour un exercice du TP à venir, et pour éviter les problèmes de dépassement de capacité, il est souhaitable que les calculs intermédiaires et le résultat soient des `double`. Adapter en conséquence la fonction `factorielle`.

(2) On considère la fonction dont la documentation et l'entête sont donnés ci-dessous :

```
/** La fonction puissance
 *  @param a un nombre à virgule flottante en double précision
 *  @param n un nombre entier positif
 *  @return la n-ième puissance a^n de a
 **/
double puissance(double a, int n) {
```

Quels sont les types de ses paramètres formels et de sa valeur de retour ?

(3) Écrire quelques exemples d'utilisation de la fonction puissance. Les mettre sous forme de tests, en vous inspirant du test suivant pour la fonction `surfaceRectangle` :

```
surfaceRectangle(4, 5) == 20
```

(4) Implanter la fonction `puissance`.

(5) Exécuter pas à pas le programme suivant :

```
int n = 3;
float x = 4;
float a = 2;
float resultat = puissance( x-a, n ) / factorielle(n);
```

Quel est la valeur de la variable `resultat` à la fin ?

(6) ♣ Réimplanter les fonctions `factorielle` et `puissance` en récursif, puis refaire l'exécution pas à pas.

Exercice 3 (Variables locales / globales, pile et exécution pas à pas)**.**
On considère les deux programmes suivants :

```
int i = 0;
int f(int j) {
    i = i + j;
    return i;
}

int resultat = f(1) + f(2) + f(3);
```

```
int f(int j) {
    int i = 0;
    i = i + j;
    return i;
}

int resultat = f(1) + f(2) + f(3);
```

(1) Surligner les différences entre les deux programmes.

(2) Chercher dans le poly de cours la sémantique de l'appel d'une fonction.

(3) Exécuter pas à pas les deux programmes en décrivant au fur et à mesure l'état de la mémoire (pile). Quelle est la valeur des variables `i` et `resultat` à la fin de l'exécution ?

(4) Décrire la différence de comportement et retrouver dans les notes de cours le commentaire à ce propos.

Exercice 4 (La trilogie code, documentation, tests)**.**
Analyser la fonction `volumePiscine` suivante :

```
/**Calcule le volume d'une piscine parallélépipédique
 * @param profondeur la profondeur de la piscine (en mètres)
 * @param largeur la largeur de la piscine (en mètres)
 * @param longueur la longueur de la piscine (en mètres)
 * @return le volume de la piscine (en litres)
 **/
double volumePiscine(double profondeur, double largeur, double longueur) {
     return 100 * profondeur * largeur * longueur;
}
```

Munie des tests :

```
    ASSERT( volumePiscine(5, 12, 5) == 30000 );
    ASSERT( volumePiscine(1,  1, 5) == 500   );
```

(1) Est-ce que les tests passent ?

(2) Est-ce que la documentation, le code et les tests sont cohérents ?

(3) Corriger les anomalies éventuelles.

Exercice ♣ 5.
Analyser la fonction `mystere` suivante :

```
string mystere(int blop) {
    string schtroumpf = "";
    for ( int hip=1; hip <= blop; hip++ ) {
        for ( int hop=1; hop <= hip; hop++ ) {
            schtroumpf += "*";
        }
        schtroumpf += "\n";
    }
    return schtroumpf;
}
```

Munie des tests :

```
    ASSERT( mystere(0) == ""             );
    ASSERT( mystere(1) == "*\n"          );
    ASSERT( mystere(2) == "*\n**\n"      );
    ASSERT( mystere(3) == "*\n**\n***\n" );
```

(1) Comment appelle-t-on cette fonction (quelle est sa *syntaxe*) ?

(2) Que fait cette fonction (quelle est sa *sémantique*) ?
Indications : la notation x += expression est un raccourci pour x = x + expression ; dans une chaîne de caractères, « `\n` » représente un saut de ligne.

(3) Choisir un bon nom pour cette fonction et ses variables et en écrire la documentation.

Exercice ♣ 6.
Le but de cet exercice est de coder une fonction `pointDeChute` qui calcule l'abscisse x_c à laquelle retombe un projectile lancé en $x = 0$ avec une vitesse v suivant un angle α (exprimé en degrés par rapport à l'horizontale). On rappelle que l'abscisse est donnée par la formule : $x_c = (2v_x v_y)/g$ où $v_x = v\cos(\alpha)$, $v_y = v\sin(\alpha)$ et g est l'accélération gravitationnelle (environ $9,8 m/s^2$ sur la planète Terre).

Implanter la fonction `pointDeChute`. On commencera par écrire sa documentation et des tests (voir TD 1).

Rappel : en C++, les fonctions mathématiques sinus et cosinus sont implantées par les fonctions prédéfinies `sin(arg)` et `cos(arg)` dans `<cmath>`, où l'angle `arg` est exprimé en radians.

Exercice ♣ 7.
En vous inspirant de votre implantation de la fonction exponentielle, documenter puis implanter les fonctions sinus et cosinus, en exploitant les formules trigonométriques usuelles pour réduire le code et accélérer la convergence. Quelles stratégies peut-on adopter pour tester ces fonctions ?

Voir aussi : `https://en.wikipedia.org/wiki/Trigonometric_functions#Computation`