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) Implantez une **fonction** `surfaceDisque` qui calcule la surface d'un disque de rayon donné. On prendra $\pi = 3,1415926$.

Correction :

```
float surfaceDisque(float rayon) {
    return 3.1415926 * rayon * rayon;
}
```

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

Correction :

```
float surfaceTriangle(float base, float hauteur) {
    return base * hauteur / 2;
}
```

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`. Adaptez en conséquence la fonction `factorielle`.

Correction :

```
/** La fonction factorielle
 *  @param n un nombre entier positif
 *  @return n! comme un nombre à virgule flottante à double précision
 **/
double factorielle(int n) {
    double resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
    return resultat;
}
```

(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 ?

Correction : Le paramètre formel `a` est de type `double`, le paramètre formel `n` est de type `int`, la valeur de retour est de type `double`.

(3) Écrivez quelques exemples d'utilisation de la fonction `puissance`. Éditez-les sous forme de tests, en vous inspirant du test suivant pour la fonction `surfaceRectangle` :

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

Correction :

```
CHECK( puissance(2.5, 0) == 1 );
CHECK( puissance(35, 1) == 35 );
CHECK( puissance(-2, 4) == 16 );
CHECK( puissance(10, 5) == 100000 );
```

(4) Implantez la fonction `puissance`.

Correction :

```
double puissance(double a, int n) {
    double resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * a;
    }
    return resultat;
}
```

(5) Cherchez dans les notes de cours la sémantique *simplifiée* de l'appel d'une fonction.

(6) Exécutez pas à pas le programme suivant :

```
int k = 3;
double x = 6;
double a = 4;
double resultat = puissance(x - a, k) / factorielle(k);
```

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

Correction : Les trois premières lignes initialisent les variables `k`, `x` et `a`. Ensuite on calcule `puissance(x - a, k) / factorielle(k)`, qu'on stocke dans `resultat`.

Au moment du calcul, les expressions passées aux fonctions auxiliaires sont remplacées par leur valeur. Ainsi l'appel effectif est `puissance(2, 3) / factorielle(3)`.

Il ne reste plus qu'à exécuter pas à pas la fonction `puissance` avec 2 et 3 comme valeurs initiales pour ses paramètres formels `a` et `n` : on obtient 8.0 comme valeur de retour.

On fait de même avec `factorielle` avec 3 comme valeur initiale pour le paramètre formel `n`, ce qui donne 6.0.

Il ne reste qu'à faire la division `8.0 / 6.0`, ce qui donne environ 1.333, et c'est la valeur de `resultat`.

(7) ♣ Implantez les fonctions `factorielle` et `puissance` en récursif cette fois, puis refaites l'exécution pas à pas. Qu'est-ce qui change ?

Correction :

```
/** La fonction factorielle récursive
 *  @param n un nombre entier positif
 *  @return n ! comme un nombre à virgule flottante à double précision
 **/
double factorielle(int n) {
    double resultat = n;
    if ( n == 0 ) {
        resultat = 1;
    } else {
        resultat = resultat * factorielle(n-1);
    }
    return resultat;
}
```

Correction :

```
/** La fonction puissance récursive
 *  @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) {
    if ( n == 0 ) {
        return 1;
    } else {
        return a * puissance(a , n - 1);
    }
}
```

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

```
1 int i = 0;
2 int f(int j) {
3     i = i + j;
4     return i;
5 }
6
7 int resultat = f(1) + f(2) + f(3);
```

```
1 int f(int j) {
2     int i = 0;
3     i = i + j;
4     return i;
5 }
6
7 int resultat = f(1) + f(2) + f(3);
```

(1) Mettez en évidence les différences entre les deux programmes (par ex. au surligneur).
(2) Cherchez dans les notes de cours la sémantique *détaillée* de l'appel d'une fonction (formalisation suivant le modèle d'exécution).
(3) Exécutez 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 entre ces programmes, et retrouver dans les notes de cours le commentaire à ce propos.

Correction :

(1) `i` est une variable globale dans le premier programme, alors qu'elle est une variable locale dans le second programme car elle est définie dans le bloc de la fonction.
(2) Voir poly
(3) On exécute le premier programme, en réservant une case mémoire pour `i` qui est une variable globale, ainsi que `resultat`.

Pour des raisons de place, la représentation de la pile dans cette correction n'est pas exactement la même que dans le poly.

no de ligne	`i`	`resultat`	pile		
1	0	?			
7 : `f(1)`	0	?			
2	0	?	`j = 1`		
3	1	?	`j = 1`		
4	1	?	`ret = 1`		
7 : `f(2)`	1	?	`f(1) = 1`		
2	1	?	`f(1) = 1`	`j = 2`	
3	3	?	`f(1) = 1`	`j = 2`	
4	3	?	`f(1) = 1`	`ret = 3`	
7 : `f(3)`	3	?	`f(1) = 1`	`f(2) = 3`	
2	3	?	`f(1) = 1`	`f(2) = 3`	`j = 3`
3	6	?	`f(1) = 1`	`f(2) = 3`	`j = 3`
4	6	?	`f(1) = 1`	`f(2) = 3`	`ret = 6`
7	6	?	`f(1) = 1`	`f(2) = 3`	`f(3) = 6`
7	6	10			

On fait de même pour le deuxième programme. Cette fois-ci, `i` n'est pas une variable globale.

no de ligne	`resultat`	pile (du bas vers le haut)
7 : `f(1)`	?	
1	?	`j = 1`
2	?	`j = 1` | `i = 0`
3	?	`j = 1` | `i = 1`
4	?	`ret = 1`
7 : `f(2)`	?	`f(1) = 1`
1	?	`f(1) = 1` | `j = 2`
2	?	`f(1) = 1` | `j = 2` | `i = 0`
3	?	`f(1) = 1` | `j = 2` | `i = 2`
4	?	`f(1) = 1` | `ret = 2`
7 : `f(3)`	?	`f(1) = 1` | `f(2) = 2`
1	?	`f(1) = 1` | `f(2) = 2` | `j = 3`
2	?	`f(1) = 1` | `f(2) = 2` | `j = 3` | `i = 0`
3	?	`f(1) = 1` | `f(2) = 2` | `j = 3` | `i = 3`
4	?	`f(1) = 1` | `f(2) = 2` | `ret = 3`
7	?	`f(1) = 1` | `f(2) = 2` | `f(3) = 3`
7	6	

Pour le premier programme : les valeurs des variables `i` et `resultat` sont 6 et 10 respectivement. Pour le deuxième programme : la valeur de `resultat` est 6 et la valeur de `i` n'est pas définie à la fin du programme (`i` n'est pas déclarée hors de `f`).

(4) Une variable locale à une fonction n'existe que le temps d'exécution de la fonction ; sa valeur est donc perdue lors du retour au programme appelant et ne peut être récupérée lors d'un appel ultérieur.

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 :

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

(1) Est-ce que les tests passent ?

Correction : Oui, les tests passent.

(2) La documentation, le code et les tests sont-ils cohérents ?

Correction : Les tests et l'implantation ne sont pas cohérents avec la documentation car il y est spécifié que le résultat doit être exprimé en litres.

(3) Corrigez les anomalies éventuelles.
Correction : Le facteur de conversion devrait être de $10^3 = 1000$ et pas de 100.

Exercice ♣ 5.
Analysez 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 :

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

(1) Comment fait-on appel à cette fonction (quelle est sa *syntaxe*) ?

Correction : La signature de la fonction :

```
string mystere(int blop) {
```

nous indique que la fonction attend un seul paramètre entier (`blop`). On écrit donc `mystere(x)` où `x` est un entier pour appeler la fonction.

(2) Que fait cette fonction (quelle est sa *sémantique*) ?
Indications : pour les chaînes de caractères, l'opérateur `+` représente la concaténation (par exemple `"Cou" + "cou"` s'évalue `"Coucou"`) ; `x += expression` est un raccourci pour `x = x + expression` ; dans une chaîne de caractères, « `\n` » représente un saut de ligne.

Correction : La fonction crée une chaîne de caractères représentant un triangle d'étoiles ayant `blop` lignes et la renvoie. La boucle externe se charge des lignes, tandis que la boucle imbriquée se charge des colonnes.

Ainsi, `mystere(4)` renvoie la chaîne de caractères :

```
*
**
***
****
```

(3) Ré-écrivez la fonction en choisissant des noms pertinents pour la fonction et ses variables, et en la précédent de sa documentation.

Correction :

```
/** Triangle d'étoiles
 *  @param nbLignes un nombre entier positif
 *  @return une chaîne de caractère représentant un triangle d'étoiles
 *    ayant nbLignes lignes
 **/
string triangleEtoiles(int nbLignes) {
    string resultat = "";
    for ( int ligne = 1; ligne <= nbLignes; ligne++ ) {
```

```
        for ( int etoile = 1; etoile <= ligne; etoile++ ) {
            resultat += "*";
        }
        resultat += "\n";
    }
    return resultat;
}
```

Exercice ♣ 6.
Le but de cet exercice est de coder une fonction `point_de_chute` qui calcule l'abscisse x_c à laquelle tombe un projectile lancé en $x = 0$ avec une vitesse v suivant un angle α (exprimé en degrés par rapport à l'horizontale). Implantez la fonction `point_de_chute`. On commencera par écrire sa documentation ainsi que des tests (voir TD 1).

Rappels :
— 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).
— 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.

Correction :

```
/** Fonction qui calcule le point de chute d'un objet lancé
 *  @param v la vitesse de lancer en m/s
 *  @param angle l'angle de lancer en degrés, entre 0 et 180
 *  @return l'abscisse du point de chute comme un nombre à virgule flottante
 *          à double precision
 **/
double pointDeChute(int v, int angle) {
    double angleRadians = angle * 3.141592 / 180;
    return 2 * v * cos(angleRadians) * v * sin(angleRadians) / 9.8;
}


void pointDeChuteTest() {
    CHECK(pointDeChute(3, 0) == 0);
    CHECK(pointDeChute(3, 90) == 0);
    CHECK(pointDeChute(0, 45) == 0);
    CHECK(pointDeChute(3, 30) + pointDeChute(3, 150) == 0);
}
```

Exercice ♣ 7.
Le but de cet exercice est de calculer la hauteur en fonction du temps $z(t)$ à laquelle se trouve un pot de fleur ($m = 3\,\mathrm{kg}$) lâché à $t = 0$ depuis le 10 ème étage ($h_0 = 27\,\mathrm{m}$), en chute libre avec résistance de l'air ; puis de calculer le temps de chute.

(1) Implantez une fonction `chute_libre(t)` calculant $z(t)$ pour un V_0 donné ($V_0 = 80\,\mathrm{m\,s^{-1}}$).
Indications :
— La hauteur s'exprime en fonction du temps par

$$z(t) = h_0 - (V_0 t + \frac{V_0^2}{g} \ln\left(\frac{1}{2}\left(1 + e^{-2tg/V_0}\right)\right) ,$$

où V_0 est la vitesse limite de chute de l'objet et $g = 9.81\,\mathrm{m\,s^{-2}}$.
— La fonction logarithme népérien est prédéfinie sous la forme `log(arg)` dans `<cmath>`.

(2) Que se passe-t-il si on varie h_0 et V_0 ? Généralisez votre fonction pour prendre en paramètres additionnels la hauteur initiale h_0 et la vitesse limite de chute V_0. Pour la gravité, définir une variable globale g. Bonus : définir cette variable globale comme une constante (nous irons sur Mars une autre fois).

Correction :

```
double g = 9.81;

/** Calcul de la hauteur d'un objet en chute libre
 *  @param time: le temps de chute
 *  @param h0: le hauteur au début de la chute
 *  @param V0: la vitesse limite
 *  @return la hauteur de l'objet au temps time
 **/
double chute_libre(double t, double h0, double Vo) {
    return ho - Vo*t - Vo**2/g*log( (1 + exp(-2*t*g/Vo)) / 2 );
}
```

Écrivez les appels à la fonction précédente pour calculer $z(t)$ pour différentes valeurs de V_0 (10, 40, 60, 120, 180).

(3) Écrivez une fonction `temps_de_chute` qui prend les même paramètres que précédemment et utilise `chute_libre` de façon répétée pour déterminer une approximation de la durée t_c de la chute du pot de fleur jusqu'au sol.

(4) La vitesse limite peut être obtenue en fonction de la masse volumique de l'air ρ, du coefficient de résistance aérodynamique C_x et de la section de l'objet S à l'aide de la formule $V_0 = \sqrt{\frac{2mg}{C_x \rho S}}$. Implantez une fonction `vitesse_limite` pour calculer cette formule. Puis implantez de nouvelles fonctions utilisant les précédentes pour calculer $z(t)$ et le temps de chute t_c en fonction des paramètres h_0, S, et m.