

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 $\pi = 3,1415926$.
- (2) Implantez une **fonction** `surfaceTriangle` qui calcule la surface d'un triangle de base et de hauteur données.

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`.
- (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) 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 );
```

- (4) Implantez la fonction `puissance`.
- (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 ?

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

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.

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 ?
- (2) La documentation, le code et les tests sont-ils cohérents ?
- (3) Corrigez les anomalies éventuelles.

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

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 $9,8 \text{ m.s}^{-2}$ 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.

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 ($m = 3 \text{ kg}$) lâché à $t = 0$ depuis le 10^{ème} étage ($h_0 = 27 \text{ m}$), 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 ($V_0 = 80 \text{ m s}^{-1}$).

Indications :

— La hauteur s'exprime en fonction du temps par

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

où V_0 est la vitesse limite de chute de l'objet et $g = 9.81 \text{ m s}^{-2}$.

— 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).

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 mêmes 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 .