Les fonctions

1 Définition d'une fonction

- Une fonction est un algorithme indépendant.
- L'appel avec ou sans paramètres de la fonction déclenche l'exécution de son bloc d'instructions.
- Une fonction se termine en retournant ou non une valeur.

2 Définition d'une procédure

Une procédure est une fonction qui ne retourne aucune valeur.

3 Structure d'une fonction

```
Fonction nomFonction(liste des paramètres) : typeRetourné
Début
        Bloc d'instructions
Fin
```

Pour faciliter la lecture des algorithmes, on retiendra les principes suivant en pseudo langage :

- Le nom d'une fonction commence par une minuscule
- Le nom d'une fonction ne comporte pas d'espace
- Si le nom d'une fonction est composé de plusieurs mots, on fera commencer chaque mot par une majuscule (à l'exception bien sûr du 1er mot qui reste en minuscule) et on évitera les traits d'union dont l'utilisation est réservée au nom des algorithmes principaux. Exemple : calculDeRemise.

4 Exécution d'une fonction

- Le programme principal s'interrompt pour appeler la fonction.
- La fonction exécute son bloc d'instructions.
- Elle s'arrête dès qu'une instruction Retourne est exécutée.
- Sinon elle s'arrête à la fin de son bloc d'instructions.
- Le programme appelant reprend alors son exécution.

5 Pourquoi écrire des fonctions ?

- Le code de l'algorithme principal est plus simple et plus court s'il fait appel à des fonctions.
- Une fonction peut être appelée plusieurs fois.
- Une seule modification dans une fonction est automatiquement répercutée pour tous les algorithmes appelant cette fonction.
- Une fonction peut être conçue comme un outil mis à disposition d'autres développeurs.

6 Fonction sans valeur retournée

Ceci est signalé en précisant qu'elle retourne « vide ».

Exemple : fonction d'affichage du mot « Bonjour ».

```
Fonction afficheBonjour() : vide
Début
        Afficher(« Bonjour ») ;
Fin
```

Elle sera utilisée par un algorithme tel que celui-ci :

```
Algorithme utiliser-fonction-afficheBonjour
Début
        afficheBonjour() ;
Fin
```

7 Fonction avec valeur retournée

Une fonction peut retourner une valeur au programme appelant.
Cette valeur est unique et son retour signifie l'arrêt de la fonction.

Exemple :

```
Fonction lireNote() : entier
Variables : note : entier ;
Début
        Afficher(« Merci de saisir une note ») ;
        Saisir(note) ;
        Retourne(note) ;
Fin
```

Elle sera utilisée par un algorithme tel que celui-ci :

```
Algorithme utiliser-fonction- lireNote
Variables : Valeur : entier ;
Début
        Valeur ← lireNote() ;
        Afficher(Valeur)
Fin
```

8 Les paramètres

Un paramètre est une variable **locale** à une fonction.
Il possède dès le début de la fonction la valeur passée par le programme appelant.
Exemple de passage de paramètres :

```
Fonction maxDeDeuxValeurs (p1 : entier, p2 : entier) : entier
Variables : Résultat : entier
Début
        Si p1 < p 2 alors
            Résultat ← p2 ;
            Sinon
                Résultat ← p1 ;
        Fin si
        Retourne (résultat) ;
Fin
```

Cette fonction est appelée par l'algorithme suivant :

```
Algorithme utiliser-fonction-maxDeDeuxValeurs
Variables : valeur1, valeur2, maxi : entiers ;
Début
        Afficher(« Saisir une 1ère valeur ») ;
        Saisir(valeur1) ;

        Afficher(« Saisir une 2ème valeur ») ;
        Saisir(valeur2) ;

        maxi ← maxDeDeuxValeurs(valeur1, valeur2) ;

        Afficher (« La plus grande valeur est : », maxi) ;
Fin
```

On dissocie complètement ce que l'on appelle les environnements de données, à savoir les variables utilisées dans l'algorithme principal et celles qui le sont dans une fonction.
Elles n'ont aucune existence commune, celles utilisées dans la fonction n'ont d'existence qu'entre le début et la fin d'exécution de la fonction.

En mémoire, cela correspond à des adressages indépendants.
Il n'existe aucun moyen pour l'algorithme principal d'avoir accès aux variables de la fonction, ni à la fonction d'avoir accès aux variables de l'algorithme principal.

Les seuls échanges se font :
- De l'algorithme à la fonction en passant des valeurs grâce aux paramètres
- De la fonction vers l'algorithme en retournant une seule et unique valeur.

NB : il est donc possible que deux variables, l'une utilisée dans l'algorithme principal, l'autre dans une fonction, portent le même nom et ne soient pas du même type puisqu'elles sont utilisées de manière différente dans des environnements de données différents.

Au cœur d'une fonction on devra distinguer paramètres et variables.

- Les paramètres sont des valeurs passées par l'algorithme principal et connues dès le début de la fonction.
- Les paramètres sont déclarés dans la définition de la fonction, **il n'est donc pas nécessaire de les redéfinir dans la section des variables**.
- Il n'est pas obligatoire de les nommer comme les variables utilisées lors de l'appel de la fonction.

- Les variables d'une fonction sont locales à la fonction, donc inconnues du programme principal
- Les variables d'une fonction sont définies dans la section « Variables » de la fonction.
- Les variables du programme principal sont inconnues de la fonction.

9 Technique de création d'une fonction

On cherchera à mettre en place ce que l'on appelle la **signature** de la fonction, à savoir :

- Le nom de la fonction
- Les paramètres de la fonction : nom, type et ordre des paramètres
- Le type de valeur retournée

Cette signature est essentielle car elle permet le polymorphisme paramétrique : deux fonctions peuvent avoir le même nom et des paramètres différents, en nombre et/ou en type.

Le polymorphisme paramétrique garantit automatiquement l'exécution de la bonne fonction associée au bon nombre de paramètres et à leurs types. En effet les programmes identifient une fonction par sa signature et pas uniquement par son nom.

10 Récursivité simple

Une fonction est dite récursive si elle s'appelle elle-même.

Pour écrire une fonction récursive, il suffit d'utiliser la fonction que vous n'avez pas encore écrite en supposant qu'elle donne déjà un résultat.

Un algorithme récursif se compose de deux parties :
- Au moins une condition d'arrêt des appels récursifs, où les valeurs à déterminer sont immédiatement connues
- Un appel récursif : la fonction s'appelle elle-même dans un autre environnement.

Exemple : voici la fonction « Factorielle » notée n !

Elle se définit ainsi : Factorielle(n) = 1 x 2 x … x (n – 1) x n

Donc :
Factorielle(1) = 1
Factorielle(2) = 1 x 2 = 2
Factorielle(3) = 1 x 2 x 3 = 6
Factorielle(4) = 1 x 2 x 3 x 4 = 24
Mais aussi
Factorielle(4) = Factorielle(3) x 4 = 24

Et donc de manière générale :
Factorielle(n) = Factorielle (n – 1) x n

On écrit la fonction :

```
Fonction Factorielle (nb : entier) : entier
Variables : F : entier ;
Début
        Si nb = 1 alors
           F← 1 ;
          Sinon
             F ←  Factorielle (nb – 1) x nb ;
        Fin si
        Retourne(F) ;
Fin
```

Appelons la fonction à partir d'un programme principal qui affiche par exemple la factorielle de 3.

```
Algorithme utiliser-Factorielle
Début
        Afficher(Factorielle(3)) ;
Fin
```

Voici le détail de l'exécution :

Factorielle(3) = Factorielle(2) x 3
= (Factorielle(1) x 2) x 3
= 1 x 2 x 3
= 6

Notez que l'on peut encore simplifier l'écriture de notre fonction :

```
Fonction Factorielle (nb : entier) : entier
Début
        Si nb = 1 alors
            Retourne(1) ;
        Fin si
        Retourne (Factorielle (nb – 1) x nb) ;
Fin
```

Autre exemple : la suite de Fibonacci

C'est une suite récurrente dont chaque terme dépend des 2 précédents.
Elle est définie par :

$U_0 = 0$ et $U_1 = 1$
$U_n = U_{n-1} + U_{n-2}$ pour tout entier n supérieur ou égal à 2

Donc :

$U_2 = U_1 + U_0 = 1 + 0 = 1$
$U_3 = U_2 + U_1 = 1 + 1 = 2$
$U_4 = U_3 + U_2 = 2 + 1 = 3$
$U_5 = U_4 + U_3 = 3 + 2 = 5$ …

On désire calculer tout terme de la suite de Fibonacci de rang n, pour tout entier n donné.

On souhaite calculer tout terme de la suite de Fibonacci de rang n, pour tout entier donné.

Deux méthodes :

- Itérative avec une boucle qui calcule tous les termes jusqu'au rang n
- Récursive en supposant connues toutes le valeurs retournées par $U_{(n-1)}$ et $U_{(n-2)}$

Pour écrire Fibonacci(n), on suppose donc que les valeurs retournées par Fibonacci(n-1) et Fibonacci(n-2) sont connues.

L'algorithme avec la méthode récursive s'écrit ainsi :

```
Fonction Fibonacci (n : entier) : entier
Début
        Si (n = 0) alors
           Retourne(0) ;
        Sinon
        {
          Si (n = 1) alors
             Retourne(1) ;
             Sinon
                Retourne (Fibonacci (n – 1) + Fibonacci (n – 2)) ;
        }
Fin
```

La notion de récursivité ne s'acquiert pas forcément facilement. Voici donc quelques règles et techniques à maîtriser :

- Pas de boucle tant que dans une récursive
- Ne pas oublier la condition d'arrêt
- Ne pas hésiter à utiliser le résultat de la fonction que vous êtes en train d'écrire
- Ne pas mettre d'instruction retourne au milieu de la fonction récursive car les instructions suivantes ne seraient pas exécutées.

11 Récursivité terminale

Une fonction est dite récursive terminale si elle retourne sans autre calcul la valeur obtenue par son appel récursif.

La dernière ligne d'une telle fonction sera :

```
Retourne (fonction (paramètres)) ;
```

Revenons à la fonction Factorielle qui nous proposait :

```
Retourne (Factorielle (nb – 1) x nb) ;
```

Ce n'est pas de la récursivité terminale car pour le retour de valeur, on multiplie la valeur de la fonction par nb. La version récursivité terminale est donc :

```
Fonction Factorielle (nb : entier, résultat : entier) : entier
Début
        Si nb = 1 alors
           Retourne(résultat) ;
        Fin si
        Retourne (Factorielle (nb – 1, résultat x nb)) ;
Fin
```

Cette fonction est appelée en mettant initialement le paramètre résultat à 1 par :

```
Fonction Factorielle (nb : entier, 1) : entier
Début
        Retourne (Factorielle (nb, 1)) ;
Fin
```

Le paramètre résultat est calculé uniquement au fur et à mesure de l'appel récursif :
Factorielle (3, 1) retourne la valeur Factorielle (2, 3) qui retourne la valeur Factorielle (1, 6) qui retourne 6.