Les tableaux

Déclaration

La déclaration d'un tableau se fait dans la `DATA DIVISION`.

Pour préciser qu'une variable est un tableau, il faut utiliser le mot-clef `OCCURS` suivi d'un nombre représentant la taille du tableau. Il contient obligatoirement un nombre fini d'éléments représentant la taille du tableau. Ces éléments peuvent être des entiers, des réels, des chaînes, etc.

Nous allons voir les différents tableaux possibles à travers quelques exemples.

Tableau unidimensionnel

Voici l'exemple le plus classique que vous allez trouver :

```
      * En tête...
      WORKING-STORAGE SECTION.

      01 tableau.
         02 entier PIC 9 OCCURS 10.
```

Résumons avec un schéma :

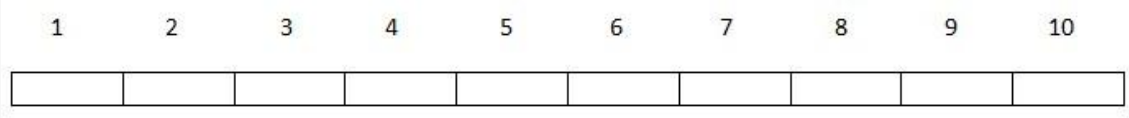

Tableau d'entiers à 1 dimension et 10 entrées

Vous voyez que sur le schéma, mes cellules sont numérotées de 1 à 10, et non de 0 à 9. **Contrairement à la majorité des langages, en Cobol un tableau commence à l'indice 1 et non 0.**

Tableau multidimensionnel

Voilà un exemple de tableau bidimensionnel :

```
01 tab.
   02 ligne OCCURS 3. * Le nombre de lignes
      03 cellule PIC 9 OCCURS 5.* Le nombre de colonnes
```

Vous pouvez créer jusqu'à 6 dimensions en imbriquant le mot clé `OCCURS` à chaque ligne, mais au-delà de 3, gérer son tableau devient une tâche difficile. Voilà à quoi cela ressemble :

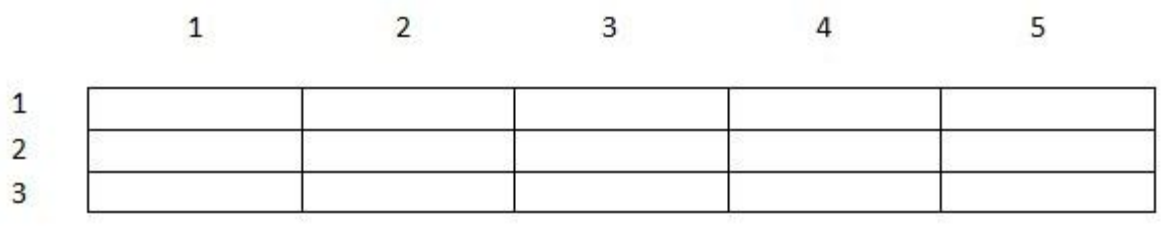

Tableau à 2 dimensions

Tableau de structure

```
01 tab.
   02 ligne-carre OCCURS 3.
      03 cellule OCCURS 5.
           04 prenom PIC x(30).
           04 nom PIC x(30).
```

Dans cet exemple, chaque cellule contient une structure à la place d'une simple variable ; pour faire simple, c'est comme si la cellule était divisée en plusieurs parties contenant chaque information. Encore une fois, rien ne vaut une bonne illustration :

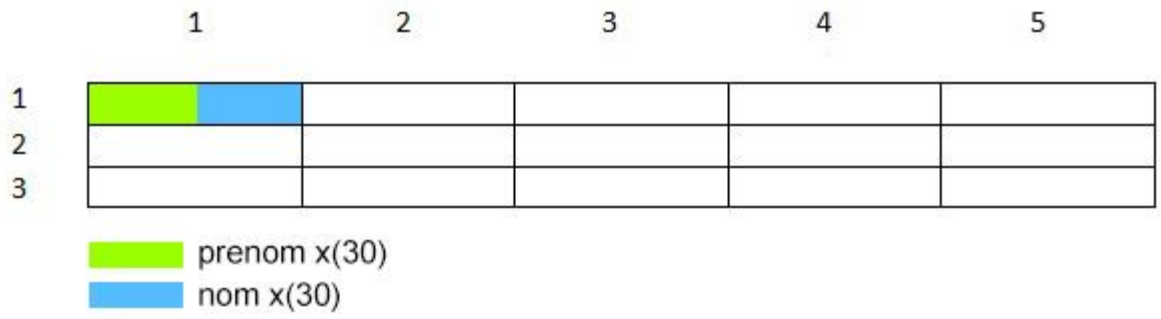

Tableau de structure à 2 dimensions

Affectation

Bon, déclarer nos tableaux c'est bien, mais les utiliser c'est mieux !

On va prendre l'exemple le plus simple avec une dimension, nous allons demander à l'utilisateur jusqu'à combien il veut compter et nous remplirons le tableau selon ce qu'il désire.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID. Tableau.

      DATA DIVISION.
      WORKING-STORAGE SECTION.

      77 n PIC 99.
      77 i PIC 99.

      01 tab.
         02 entier PIC 99 OCCURS 99.

      SCREEN SECTION.

      01 pls-n.
         02 BLANK SCREEN.
         02 LINE 5 COL 8 VALUE 'Valeur de n : '.
         02 PIC 99 TO n REQUIRED.

      01 pla-tab.
         02 BLANK SCREEN.
         02 LINE 2.
         02 OCCURS 99.
              03 LINE + 1 COL 5 PIC zz FROM entier.

      PROCEDURE DIVISION.
      INITIALIZE tab.

      DISPLAY pls-n.
      ACCEPT pls-n.

      PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
         MOVE i TO entier(i)
      END-PERFORM.

      DISPLAY pla-tab.

      STOP RUN.
```

Sortez vos loupes !

On va analyser ce code de manière un peu plus approfondie.

Pour la partie située dans la `WORKING-STORAGE SECTION`, les explications sont données dans la partie traitant la déclaration donc je ne vais pas y revenir, il vous suffit de lire le début du chapitre.

Voyons plutôt du côté de `pla-tab` :

```
      01 pla-tab.
         02 BLANK SCREEN.
         02 LINE 2.
         02 OCCURS 99.
              03 LINE + 1 COL 5 PIC zz FROM entier.
```

OCCURS permet de choisir le nombre de ligne que l'on veut afficher lors de l'affichage de notre tableau. Donc là, on va afficher le tableau complet, soit 99 lignes.

La ligne suivante, permet de récupérer ligne par ligne le contenu de la variable **entier**. Vous pouvez également constater la petite subtilité qui permet de sauter une ligne à chaque affichage avec `LINE + 1`.

Passons au `PROCEDURE` :

```
      PROCEDURE DIVISION.
      INITIALIZE tab.

      DISPLAY pls-n.
      ACCEPT pls-n.

      PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
         MOVE i TO entier(i)
      END-PERFORM.

      DISPLAY pla-tab.

      STOP RUN.
```

Une fois que l'utilisateur a choisi le nombre qu'il veut, nous remplissons le tableau jusqu'à la valeur saisie. Nous faisons un `TEST AFTER` afin d'arriver jusqu'à l'indice n, sans lui nous irions seulement jusqu'à l'indice n-1 (rappelez-vous le schéma dans le chapitre sur les boucles).

Ensuite, nous ajoutons le numéro de l'itération dans le tableau grâce à `MOVE i TO entier(i)`.

J'ai utilisé `MOVE` pour attribuer une valeur à `entier(i)`, mais on peut aussi le faire avec `COMPUTE`.

Je ne comprends pas très bien là… Pourquoi `entier(i)` et pas `pla-tab(i)` comme dans les autres langages ?

Voilà une autre particularité du Cobol : on accède au contenu d'un tableau via le nom de la cellule, et non celui du tableau ! Donc entier(1) vaut 1, entier(2) vaut 2, etc.

Ensuite `DISPLAY pla-tab` pour afficher notre tableau, voici le résultat pour n = 10 :

```
1
2
3
4
5
6
7
8
9
10
```

Autre méthode

Jusqu'à maintenant nous avons afficher notre tableau d'un bloc, mais vous pouvez aussi le faire ligne par ligne comme ceci :

```
      * En tete et déclaration...
      01 pla-ligne.
         02 LINE i.
         02 COL 5 PIC zz FROM entier(i).

      PROCEDURE DIVISION.
      INITIALIZE tab.

      DISPLAY pls-n.
      ACCEPT pls-n.

      PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
          MOVE i TO entier(i)
          DISPLAY pla-ligne
      END-PERFORM.

      STOP RUN.
```

Le résultat est exactement le même mais c'est simplement pour vous montrer une autre manière de procéder.

Taille des tableaux

Pour l'instant je vous ai uniquement montré comment faire des tableaux à taille fixe, mais il est également possible de faire des tableaux dont la taille varie.

```
      01 tab.
         02 default PIC 99.
         02 val PIC 9 OCCURS 1 TO 50 DEPENDING default.
```

De manière plus globale, la syntaxe peut se représenter de la manière suivante : `OCCURS x TO y DEPENDING z`. x symbolise la taille minimum du tableau, y la taille maximum et z la taille par défaut.

Donc z doit se situer entre x et y, sous forme plus mathématique cela se traduit par :

$$x <= z <= y$$ x<=z<=y

Je ne l'ai pas vraiment abordé de manière précise, mais pour les tableaux à plusieurs dimensions, l'accès à une cellule se fait comme ceci : `cellule(i,j,k)`

Opérations

Certaines fonctions permettent d'effectuer des opérations sur les tableaux, nous allons en voir deux.

Faisons le tri !

La première opération que nous allons voir c'est comment trier nos tableaux. Nous allons reprendre l'exemple de tout à l'heure en le modifiant un petit peu :

```
       PROCEDURE DIVISION.
       INITIALIZE tab.

       MOVE 15 TO n.
       PERFORM TEST AFTER VARYING i FROM 1 BY 1 UNTIL i = n
           MOVE i TO entier(i)
       END-PERFORM.

       SORT entier DESCENDING.
       DISPLAY pla-tab.
```

En exécutant notre programme, l'affichage sera :

```
15
14
13
12
11
10
9
8
7
6
5
4
3
```

```
142
151
```

Je pense que vous l'aurez compris, `SORT entier DESCENDING` permet de trier les éléments de notre tableau par ordre décroissant. Ici nous utilisons `DESCENDING` car nos éléments ont été ajoutés par ordre croissant.

Mais dans le cas où les entrées sont dans un ordre aléatoire et que l'on veut afficher notre tableau dans l'ordre, il suffit d'utiliser l'opération inverse, à savoir `ASCENDING`.

La recherche

L'un des avantages des tableaux, c'est que l'on peut facilement rechercher l'élément qui nous intéresse grâce à une fonction toute faite.

Pour pouvoir faire une recherche il faut que notre tableau soit indexé, lors de sa déclaration nous allons ajouter ceci :

```
      01 tab.
         02 entier PIC 99 OCCURS 99 INDEXED BY indice.
```

Et notre recherche dans `PROCEDURE DIVISION` se fait comme ceci :

```
      SET indice TO 1.
      SEARCH entier
         AT END
             DISPLAY "Element introuvable..."
         WHEN entier(indice) = 5
             DISPLAY "Element " entier(indice) " trouve ! ".
```

Pour pouvoir commencer notre recherche correctement, il faut initialiser l'indice de départ. Donc avec `SET indice TO 1` nous positionnons notre curseur de recherche à la cellule 1. Ensuite il faut indiquer dans quelle cellule on veut chercher notre élément, ce qui correspond à `SEARCH entier`.

La clause `AT END` est optionnelle, mais je la mets ici, pour signaler à notre utilisateur si la recherche ne renvoie aucun résultat, ce qui affichera "Element introuvable…" à l'écran.

Ensuite, à la manière d'une condition multiple avec le mot `WHEN`, nous allons énoncer nos conditions. Ici je cherche si 5 se trouve dans notre tableau entier. Si un élément est trouvé alors on affiche "Element 05 trouve !", sachant qu'*indice* joue le rôle de variable d'itération.

Nous avons fait le tour en ce qui concerne les tableaux en COBOL, comme d'habitude, on se retrouve au prochain chapitre.

Les fonctions intrinsèques

Les fonctions intrinsèques sont des fonctions toute prête à l'emploi, cela évite de réécrire bêtement des fonctions qui existent déjà.

Voyons ça avec un exemple d'utilisation :

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. fonction.

       WORKING-STORAGE SECTION.
       01 mot PIC A(30).
       01 nombre PIC 9.
       01 racine PIC 9.

       SCREEN SECTION.
       01 plg-aff-titre.
          02 BLANK SCREEN.
          02 LINE 1 COL 10 'Utilisation des fonctions.'.

       01 plg-saisie.
          02 LINE 3 COL 1 'Tapez un mot en minuscule : '.
          02 PIC A(30) TO mot REQUIRED.
          02 LINE 4 COL 1 'Entrez un nombre entre 0 et 9 : '.
          02 PIC z TO nombre REQUIRED.

       01 plg-res.
          02 LINE 7 COL 1 'Voici votre mot en majuscule : '.
          02 PIC A(30) FROM mot.
          02 LINE 8 COL 1 'La racine carre du nombre est : '.
          02 PIC 9 FROM racine.

       PROCEDURE DIVISION.
          DISPLAY plg-aff-titre plg-saisie.
          ACCEPT plg-saisie.

          MOVE FUNCTION UPPER-CASE (mot) TO mot.
          MOVE FUNCTION SQRT (nombre) TO racine.

          DISPLAY plg-res.

       GOBACK.
```

Dans cet exemple, on demande à l'utilisateur d'entrer un mot en minuscule et un entier entre 0 et 9. Ensuite, on utilise la fonction `UPPER-CASE` pour mettre toutes les lettres du mot en majuscule. Et, on utilise la fonction `SQRT` pour obtenir la racine carré du nombre entré. Si le résultat n'est pas juste, alors le résultat est tronqué.

Donc, on peut voir que pour utiliser une fonction, c'est très simple. Il suffit d'écrire le mot `FUNCTION` suivi du nom de la fonction en question.

```
Utilisation des fonctions.

Tapez un mot en minuscule : canard
Entrez un nombre entier entre 0 et 9 : 9

Voici votre mot en majuscule : CANARD
La racine carre du nombre est : 3
```

Quelques fonctions utiles

Nous vous avons préparé quelques tableaux de fonctions intrinsèques, mais nous ne détaillerons pas leur utilisation précise.

Fonctions mathématiques

Nom	Description
COS	Fonction cosinus
TAN	Fonction tangente
LOG	Fonction Logarithme naturel
LOG10	fonction Log base 10
SIN	Fonction Sinus
FACTORIAL	Factoriel n
SQRT	Fonction racine
SUM	Fait la somme de plusieurs valeurs
MOD	Modulo 2: renvoit le reste d'une division (10 module 3 = 1 par exemple)
RANDOM	Génère un nombre entre 0 et 1 exclu

Fonctions sur les caractères

Nom	Description
LENGTH	Retourne la longueur d'une chaine de caractères sous forme d'un entier
UPPER-CASE	Permet de passer une chaine de caractères en majuscules
LOWER-CASE	Permet de passer une chaine de caractères en minuscules

Nom	Description
REVERSE	Inverse une chaine de caractères

Les dates

Nom	Description
CURRENT-DATE	Retourne la date actuelle sous la forme AAAAMMJJHHMMSSCC
WHEN-COMPILED	Donne la date de compilation

Création d'une fonction

Forme générale d'une fonction

L'utilisation des fonctions prédéfinies c'est bien, mais écrire les siennes c'est encore mieux. Une fonction possède des paramètres formels. Une fois que la fonction a été appelée, celle-ci donne un résultat.

Voici la forme générale d'une fonction.

```
      $set repository "update on"
      FUNCTION-ID. nomFonction.
      [file-control.]
      [data division.]
      [file section.]
      [working-storage section.]

      LINKAGE SECTION.

      *> ici déclaration des paramètres en entrées
      *> puis déclaration du paramètre résultat

      [screen section.]

      PROCEDURE DIVISION [USING liste des paramètres en entrée] GIVING para

      *> ici toutes les instructions

      END FUNCTION nomFonction.
```

Tout d'abord, j'ai fait exprès de mettre entre crochets les sections qui n'ont pas d'importance ici. Pour écrire une fonction, on doit utiliser une nouvelle section : la `LINKAGE SECTION`, qui se trouve dans la `DATA DIVISION`.

Dans la `LINKAGE SECTION`, il faut mettre toutes les variables qui seront les paramètres formels de la fonction. Les variables déclarées dans cette section seront liées entre la fonction, et le programme appelant cette fonction, bien souvent le programme principal.

Une fonction peut avoir un ou plusieurs paramètres en entrée, et un seul résultat en sorti. Une fonction peut ne pas avoir de paramètres en entrée, mais renverra obligatoirement un résultat.

Sinon, vous pouvez déclarer vos variables comme vous savez déjà le faire. Il faut juste que les variables définies dans la `LINKAGE SECTION` soient identiques à celles des variables originales définies dans le programme appelant.

Attention ! La clause `VALUE` ne peut pas être utilisée pour l'initialisation des variables déclarer dans la `LINKAGE SECTION`.

Ensuite, il y a deux nouvelles clauses, `USING` et `GIVING`, qui apparaissent sur la même ligne que la `PROCEDURE DIVISION`.

On écrit la clause `USING` suivie du ou des paramètres en entrée nécessaire(s) pour la fonction. Vous remarquerez qu'il y a des crochets entourant la clause `USING` et ses paramètres d'entrée. Je pense que vous avez deviné pourquoi.

Oui, on n'a pas besoin de l'écrire si la fonction n'a pas de paramètre en entrée. De plus, lorsque la clause `USING` est utilisée, l'ordre dans lequel est écrit les variables est important : il doit correspondre à l'ordre utilisé lors de l'appel de la fonction.

La seconde clause `GIVING` est suivie du nom de la variable résultat qui a été déclarée dans la `LINKAGE SECTION`.

Mais c'est quoi ce truc `\$set repository "update on"` tout en haut ?

J'allais justement y venir !

Pour qu'une fonction puisse être utilisable, elle doit être compilée et enregistrée dans un répertoire, d'où le **repository**.

L'enregistrement de la fonction s'effectue grâce à la commande suivante : `\$set repository "update on"`. Mais cette commande va enregistrer par défaut la fonction dans le répertoire courant.

Si vous voulez enregistrer la fonction dans un autre endroit, il faut utiliser une commande supplémentaire : `\$set rdfpath "_chemin complet vers le répertoire_"`. Cette commande est à mettre avant la commande précédente. On obtient donc :

```
      $set rdfpath "chemin complet vers le répertoire"
      $set repository "update on"
      *> définition du reste de la fonction
```

Voici la fonction. Je préfère enregistrer la fonction dans le répertoire courant.
À compiler en premier !!

```
$set repository "update on"
FUNCTION-ID. calculSomme.

LINKAGE SECTION.
01 param1 pic 99.
01 param2 pic 99.
01 paramRes pic 99.

PROCEDURE DIVISION USING param1 param2 GIVING paramRes.

COMPUTE paramRes = param1 + param2.

END FUNCTION calculSomme.
```

Et voilà le programme principal utilisant la fonction.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. progPrinc.
REPOSITORY.
FUNCTION calculSomme.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 entier1 PIC 99.
01 entier2 PIC 99.
01 res PIC 99.

SCREEN SECTION.
01 plg-aff-titre.
    02 BLANK SCREEN.
    02 LINE 1 COL 20 'Somme de deux entiers'.

01 plg-saisie.
    02 LINE 4 COL 1 'Entrez le nombre 1 : '.
    02 PIC zz TO entier1 REQUIRED.
    02 LINE 5 COL 1 'Entrez le nombre 2 : '.
    02 PIC zz TO entier2 REQUIRED.

01 plg-resultat.
    02 line 8 col 1 'La somme des deux entiers est : '.
    02 PIC 99 FROM res.

PROCEDURE DIVISION.

INITIALIZE entier1 entier2 res.
DISPLAY plg-aff-titre plg-saisie.
ACCEPT plg-saisie.

MOVE FUNCTION calculSomme(entier1,entier2) TO res.
DISPLAY plg-resultat.

GOBACK.
```

```
37      END PROGRAM progPrinc.
```

Les sous-programmes

Dans cette partie, nous allons voir la création de sous-programmes. Dans un projet important, au lieu de réécrire plusieurs fois le même code, on fera plutôt plusieurs appels à un même sous-programme. C'est le principe même de la programmation procédurale. Un sous-programme possède des paramètres qui, comme pour une procédure, permettent d'échanger des données avec un programme appelant.

Écrire un sous-programme n'est pas très compliqué, c'est même très proche de l'écriture des fonctions, puisque l'on y retrouve le même principe de construction. À la différence qu'un sous-programme peut avoir un point d'entrée principal, et un ou plusieurs points d'entrées secondaires.

Je vais vous introduire les différentes notions petit à petit.

Définition générale d'un sous-programme

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. nom du sous-programme.
3
4       [file-control.]
5       [data division.]
6       [file section.]
7
8       [working-storage section.]
9       *> Possibilité de déclarer une variable
10
11
12      LINKAGE SECTION.
13      *> déclaration des paramètres en entrée
14      *> sans oublier le paramètre résultat
15
16
17      [screen section.]
18
19      *>point d'entrée principal
20
21PROCEDURE DIVISION [USING BY REFERENCE ou BY VALUE paramètre en entrée] [RE
22
23
24      *>point de sortie
25      GOBACK
26      ou
27      EXIT PROGRAM [GIVING variable déclaré dans la WSS]
28
29
30
31      *>point d'entrée secondaire
32      ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée
```

```
33      END PROGRAM nom du sous-programme.
```

On retrouve bien sûr la `LINKAGE SECTION` qui garde le même rôle que pour les fonctions. À savoir qu'il n'y a pas besoin d'enregistrer un sous-programme dans un répertoire comme on devait le faire avec les fonctions. Il suffit juste de le compiler avant de l'utiliser.

À partir du point d'entrée principal, vous pouvez voir cette ligne :

```
1PROCEDURE DIVISION [USING BY REFERENCE ou BY VALUE paramètre en entrée] [RE
```

Passage de paramètre

On retrouve la clause `USING` qui garde ici aussi le même rôle. Il y a une nouveauté qui apparaît, c'est le passage des paramètres. Il peut être envisagé par adresse ou par valeur.

`BY REFERENCE` : c'est-à-dire par adresse. Un paramètre transmis par adresse peut être partagé entre le programme appelant et le sous-programme appelé. Ce qui veut dire que toute modification du paramètre envoyé au sous-programme est répercutée au niveau du paramètre du programme appelant. Lors de l'appel, le paramètre du sous-programme se voit attribuer une adresse qui est celle du paramètre du programme principal.

`BY VALUE` : c'est-à-dire par valeur. Lors de l'appel, l'adresse du paramètre du programme appelant sera différente de celle du paramètre du sous-programme. Dans ce cas, au moment de l'appel, la valeur du paramètre du programme principal est directement affectée au paramètre du sous-programme comme une copie. Donc, contrairement au passage par adresse, le passage par valeur permet de modifier une variable sans que cela n'affecte la variable utilisée dans le programme principal. De plus, dans ce type de transfert de paramètres, la variable ne doit pas dépasser 2 octets.

Si rien n'est spécifié au niveau du mode de passage de paramètres, alors c'est la transmission par adresse qui sera effectuée par défaut. D'ailleurs, aujourd'hui il est beaucoup moins courant de préciser le passage de paramètres. Tout simplement parce qu'on n'est plus à l'époque où l'on manquait de mémoire…

Après le passage de paramètres, on peut voir l'instruction `RETURNING`. Cette instruction fait la même chose que `GIVING` sauf que celle-ci concerne les sous-programmes.

Il se pourrait que vous ayez à utiliser juste la phrase : `PROCEDURE DIVISION RETURNING` *variable* .
Ce genre de phrase n'est à utiliser que dans les sous-programmes. En effet, utiliser cette phrase dans le programme principal de votre projet pourrait entraîner des résultats imprévisibles.

Appel d'un sous-programme

```
CALL id-sous-programme [USING BY REFERENCE ou BY CONTENT paramètre en

[EXCEPTION instruction-impérative1]

[NOT EXCEPTION instruction-impérative2]

[RETURNING variable]

END-CALL
```

Au moment de l'appel d'un sous-programme, on peut décider de la méthode du passage des paramètres comme on l'a vu précédemment.

Mais c'est quoi `BY CONTENT` ?! Encore un autre méthode de passage de paramètres ?

Non, pas du tout. Vous n'avez pas besoin de vous inquiéter, en fait cela correspond à la même chose que l'explication que je vous ai donnée pour `BY VALUE` plus haut.

En 2002 est apparue une nouvelle norme avec l'adaptation du COBOL vers le langage orienté objet.
Pour ceux qui ont déjà programmé avec des langages orientés objet comme le Java, ou le C# entre autres, le concept de la gestion des exceptions est similaire.

Je ne vais pas m'étendre sur l'explication du fonctionnement des exceptions. Il faut juste savoir qu'il peut arriver que l'appel du sous-programme échoue. Cela peut se produire si ce dernier ne peut pas se charger dynamiquement, ou s'il n'y pas assez de place en mémoire. Dans ce cas-là, une exception est levée, et donc le code se trouvant après le mot-clé `EXCEPTION` sera exécuté.

Si dans le cas contraire, l'appel du sous-programme a fonctionné, alors les instructions écrites après le mot-clé `NOT EXCEPTION` seront exécutées, et ensuite la main sera passée au sous-programme.

Sachez que la gestion des exceptions en COBOL n'est en général pas très utilisée. Personnellement, je n'ai jamais eu besoin de les utiliser, mais vous savez que cela existe maintenant.

L'instruction `RETURNING` peut aussi être utilisée dans la phrase de `CALL`. Cela permet tout simplement de recevoir le résultat du sous-programme. Dans le cas où le sous-programme est quitté avec une instruction du type `EXIT PROGRAM GIVING` *variable*.

Lorsque l'appel est terminé, il ne faut pas oublier le mot clé `END-CALL`.

Point d'entrée secondaire

Dans la définition générale d'un sous-programme, on peut voir qu'il peut y avoir un ou plusieurs points d'entrées secondaires. Un point d'entrée secondaire est défini comme suit : `ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée].`

On utilise le mot clé `ENTRY` suivi du nom du point d'entrée entre apostrophes. On peut utiliser aussi la clause `USING` et préciser le mode de passage des paramètres comme on l'a vu précédemment avec le point d'entrée principal.

À chaque fois que vous définissez un point d'entrée secondaire, il ne faut pas oublier d'écrire l'instruction `GOBACK` pour indiquer la sortie du sous-programme.

Exemple :

```
      ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée
            instruction 1
            instruction 2

            GOBACK.

      ENTRY 'id-entrée' [USING BY REFERENCE ou BY VALUE paramètre en entrée
            instruction 1
            instruction 2

            GOBACK.
```

On n'a pas le droit d'utiliser le mot-clé `RETURNING` dans un point d'entrée secondaire.

Sortie d'un sous-programme

Que l'on se trouve au niveau du point d'entrée principal, ou d'un point d'entrée secondaire, on peut quitter un sous-programme de la même manière.

Pour sortir d'un sous-programme, on peut utiliser l'instruction `GOBACK`. On peut aussi utiliser l'instruction `EXIT PROGRAM` en rajoutant si cela est nécessaire la clause `GIVING` pour renvoyer le résultat du sous-programme.

Il est tout de même préférable de quitter un sous-programme en utilisant `GOBACK`.

```
      GOBACK ou EXIT PROGRAM [GIVING variable]
```

Exemple 1

Voici un premier exemple de sous-programme très simple qui effectue la somme de deux entiers. J'ai choisi de faire un sous-programme n'utilisant que le point d'entrée principal pour le moment.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID.somme.

      DATA DIVISION.
```

```
      WORKING-STORAGE SECTION.
      01 res PIC 999.

      LINKAGE SECTION.
      01 entier1 PIC 99.
      01 entier2 PIC 99.

      PROCEDURE DIVISION USING entier1 entier2.

         COMPUTE res = entier1 + entier2

      EXIT PROGRAM GIVING res.
      END PROGRAM somme.
```

Dans la première ligne surlignée, j'ai seulement utilisé la clause `USING` pour le passage de paramètres sans préciser le mode de passage.

Dans la seconde ligne surlignée, comme je n'ai pas utilisé la clause `RETURNING` au niveau du point d'entrée principal, j'ai dû utiliser l'instruction `EXIT PROGRAM GIVING` *res* pour que le sous-programme donne le résultat au programme principal. *res* étant la variable qui a été déclarée dans la WSS.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID.progPrinc.

      WORKING-STORAGE SECTION.
      01 entier1 PIC 99.
      01 entier2 PIC 99.
      01 res PIC 999.

      SCREEN SECTION.
      01 plg-aff-titre.
         02 BLANK SCREEN.
         02 LINE 1 COL 10 'Utilisation d''un sous-programme'.

      01 plg-saisie.
         02 LINE 3 COL 1 'Entrez un nombre entier : '.
         02 PIC zz TO entier1 REQUIRED.
         02 LINE 4 COL 1 'Entrez un nombre entier : '.
         02 PIC zz TO entier2 REQUIRED.

      01 plg-res.
         02 LINE 7 COL 1 'La somme des deux entiers est : '.
         02 PIC 999 FROM res.

      PROCEDURE DIVISION.

         INITIALIZE entier1 entier2.

         DISPLAY plg-aff-titre plg-saisie.
         ACCEPT plg-saisie.

```

```
31          CALL 'somme' USING entier1 entier2 RETURNING res END-CALL
32
33          DISPLAY plg-res.
34
35      GOBACK.
36      END PROGRAM progPrinc.
```

Dans le programme principal, j'ai surligné la ligne la plus importante : l'appel du sous-programme. Vous pouvez voir que je n'ai rien fait de compliqué. Je n'ai ni précisé le mode de transmission des paramètres ni utilisé les exceptions. Et pourtant, ce code marche très bien. Grâce à l'instruction `RETURNING`, le résultat est directement envoyé dans la variable *res*.

Il est quand même à noter que le nom du sous-programme appelé doit être mis entre apostrophes (*simple quote* pour les habitués).

Exemple 2

On pourrait imaginer un sous-programme calculant les caractéristiques d'un rectangle. Pour cet exemple, j'ai choisi de créer un sous-programme où l'on passera seulement par deux points d'entrées secondaires. Un point d'entrée permettrait de calculer le périmètre d'un rectangle, et le second calculerait la surface.

Voici le sous-programme :

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID.caractRect.
3
4       LINKAGE SECTION.
5       01 longueur PIC 99.
6       01 largeur PIC 99.
7       01 perimetre PIC 9(3).
8       01 surface PIC 9(4).
9
10      PROCEDURE DIVISION.
11      GOBACK.
12
13      ENTRY 'perimetre' USING longueur largeur perimetre.
14
15           COMPUTE perimetre = ( longueur + largeur ) * 2
16           GOBACK.
17
18      ENTRY 'surface' USING longueur largeur surface.
19
20           COMPUTE surface = longueur * largeur
21           GOBACK.
22
23      END PROGRAM caractRect.
```

Le code n'a rien de compliqué, il faut juste veiller à bien mette l'instruction `GOBACK` pour indiquer la sortie du sous-programme pour chaque point d'entrée créé.

Et voici le programme principal :

```
      PROGRAM-ID. progPrinc.

      WORKING-STORAGE SECTION.
      01 longueur PIC 99.
      01 largeur PIC 99.
      01 perimetre PIC 9(3).
      01 surface PIC 9(4).

      SCREEN SECTION.
      01 plg-aff-titre.
         02 BLANK SCREEN.
         02 LINE 1 COL 10 'Caracteristique rectangle'.

      01 plg-saisie.
         02 LINE 3 COL 1 'Entrez la longueur du rectangle : '.
         02 PIC zz TO longueur.
         02 LINE 4 COL 1 'Entrez la largeur du rectangle : '.
         02 PIC zz TO largeur.

      01 plg-res.
         02 LINE 8 COL 1 'Le perimetre du rectangle est : '.
         02 PIC z(3) FROM perimetre.
         02 LINE 9 COL 1 'La surface du rectagle est : '.
         02 PIC z(4) FROM surface.

      PROCEDURE DIVISION.
         DISPLAY plg-aff-titre plg-saisie.
         ACCEPT plg-saisie.

         CALL 'caractRect'.

         CALL 'perimetre' USING longueur largeur perimetre END-CALL.
         CALL 'surface' USING longueur largeur surface END-CALL.

         DISPLAY plg-res.

      END PROGRAM progPrinc.
```

Quand vous voulez utiliser un sous-programme qui a un ou plusieurs points d'entrées secondaires, il faut d'abord appeler le point d'entrée principal du sous-programme, comme je l'ai fait ici au niveau de la ligne surlignée. Après seulement vous pouvez appeler le sous-programme à partir des points d'entrée secondaire comme vous savez le faire.

On a fait le tour des fonctions et des sous-programmes, de leur création à leur utilisation. Vous êtes maintenant parés pour faire de gros projets en COBOL.