EXERCICES COBOL

Du Novice à l'Expert

■ Novice	■ Débutant	■ Intermédiaire	■ Avancé	■ Expert
Ex. 1–3	Ex. 4–6	Ex. 7–9	Ex. 10–12	Ex. 13–15

Chaque exercice inclut : objectif, contexte, instructions pas-à-pas, exemple de code, résultat attendu et conseils pratiques.

Compatible GNU COBOL 4.0 + Docker MainFreem | Exercices SQL : ocesql + PostgreSQL (voir Dockerfile)

github.com/samarha-dev/MainFreem

Table des matières

PARTIE 1 — Les Bases du COBOL

Exercice 1 — Hello, Monde !

Niveau : ■ Novice

■ Objectif

Créer votre premier programme COBOL qui affiche un message à l'écran.

■ Contexte

Le COBOL est structuré en 4 DIVISIONS : IDENTIFICATION, ENVIRONMENT, DATA et PROCEDURE. Ce premier exercice vous familiarise avec cette structure de base et la compilation avec GNU COBOL.

■ Instructions

1. Créer un fichier hello.cobol avec les 4 divisions obligatoires.
2. Dans la PROCEDURE DIVISION, utiliser DISPLAY pour afficher 'Bonjour, le monde !'
3. Terminer avec STOP RUN.
4. Compiler : cobc -x hello.cobol puis exécuter : ./hello

■ Exemple / Structure de départ

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO.

       ENVIRONMENT DIVISION.

       DATA DIVISION.

       PROCEDURE DIVISION.
           DISPLAY '???'.
           STOP RUN.
```

■ Résultat attendu

Le terminal affiche : Bonjour, le monde !

■ Conseils

- Colonnes 1-6 : numéros de ligne. Colonne 7 : zone A. Colonne 12+ : zone B.
- Chaque instruction se termine par un point (.).
- GNU COBOL est moins strict sur les colonnes en format 'free' : cobc -x -free hello.cobol

---

Exercice 2 — Variables et DISPLAY

Niveau : ■ Novice

■ Objectif

Déclarer des variables dans la WORKING-STORAGE SECTION et les afficher.

■ Contexte

PIC (PICTURE) définit le type : PIC 9 = chiffre, PIC X = alphanumérique. MOVE affecte une valeur à une variable.

■ Instructions

1. Déclarer 3 variables : WS-NOM (PIC X(20)), WS-AGE (PIC 9(3)), WS-VILLE (PIC X(30)).
2. Initialiser les variables avec MOVE.
3. Afficher chaque variable avec DISPLAY et un libellé (ex: 'Nom : ' WS-NOM).
4. Compiler et tester dans le Docker.

■ Exemple / Structure de départ

```
       WORKING-STORAGE SECTION.
       01 WS-NOM    PIC X(20).
       01 WS-AGE    PIC 9(3).
       01 WS-VILLE  PIC X(30).

           MOVE 'Jean Dupont' TO WS-NOM.
           MOVE 25            TO WS-AGE.
           DISPLAY 'Nom : ' WS-NOM.
```

■ Résultat attendu

Les 3 variables s'affichent avec leur libellé.

■ Conseils

- VALUE permet d'initialiser dès la déclaration : PIC X(20) VALUE 'Jean'.
- MOVE SPACES efface une variable alphanumérique. MOVE ZEROS remet un numérique à 0.

Exercice 3 — Opérations Arithmétiques

Niveau : ■ Novice

■ Objectif

Réaliser des opérations de base : addition, soustraction, multiplication, division.

■ Instructions

1. Déclarer WS-A (PIC 9(5)), WS-B (PIC 9(5)), WS-RESULT (PIC 9(10)), WS-RESTE (PIC 9(5)).
2. Initialiser WS-A = 150 et WS-B = 40.
3. Calculer et afficher la somme, la différence et le produit.
4. Calculer la division entière + reste avec DIVIDE ... GIVING ... REMAINDER ...

■ Exemple / Structure de départ

```
           ADD WS-A TO WS-B GIVING WS-RESULT.
           SUBTRACT WS-B FROM WS-A GIVING WS-RESULT.
           MULTIPLY WS-A BY WS-B GIVING WS-RESULT.
           DIVIDE WS-A BY WS-B GIVING WS-RESULT
               REMAINDER WS-RESTE.
```

■ Résultat attendu

Somme=190, Différence=110, Produit=6000, Division=3, Reste=30

■ Conseils

- COMPUTE permet des expressions complexes : COMPUTE WS-RESULT = WS-A * WS-B + 10.
- Attention au dépassement : PIC 9(5) * PIC 9(5) peut nécessiter PIC 9(10) pour le résultat !

PARTIE 2 — Structures de Contrôle

Exercice 4 — Conditions avec IF / ELSE

Niveau : ■ Débutant

■ Objectif

Utiliser les structures conditionnelles IF/ELSE/END-IF pour contrôler le flux du programme.

■ Contexte

La condition peut porter sur des valeurs, des comparaisons, ou des 88-levels (valeurs conditionnelles nommées).

■ Instructions

1. Déclarer WS-NOTE (PIC 9(3)) et WS-MENTION (PIC X(15)).
2. Initialiser WS-NOTE à la valeur de votre choix.
3. Écrire un IF imbriqué : >=16 Très Bien, >=14 Bien, >=12 Assez Bien, >=10 Passable, <10 Insuffisant.
4. Afficher la note et la mention.

■ Exemple / Structure de départ

```
IF WS-NOTE >= 16
    MOVE 'Tres Bien'    TO WS-MENTION
ELSE IF WS-NOTE >= 14
    MOVE 'Bien'         TO WS-MENTION
ELSE IF WS-NOTE >= 10
    MOVE 'Passable'     TO WS-MENTION
ELSE
    MOVE 'Insuffisant'  TO WS-MENTION
END-IF.
```

■ Résultat attendu

La mention correcte s'affiche selon la note.

■ Conseils

- Toujours fermer avec END-IF pour éviter les ambiguïtés.
- Les 88-levels créent des conditions nommées : 88 NOTE-ADMISE VALUES 10 THRU 20.

Exercice 5 — Boucles PERFORM

Niveau : ■ Débutant

■ Objectif

Maîtriser les différentes formes de PERFORM pour répéter des traitements.

■ Instructions

1. Écrire un programme qui calcule et affiche la table de multiplication d'un nombre saisi.
2. Utiliser PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 10.
3. Bonus : afficher les carrés des 10 premiers entiers avec PERFORM TIMES.
4. Bonus 2 : extraire le calcul dans un paragraphe séparé appelé par PERFORM.

■ Exemple / Structure de départ

```
       01 WS-NOMBRE  PIC 9(2) VALUE 7.
       01 WS-I       PIC 9(2).
       01 WS-RESULT  PIC 9(5).

           PERFORM VARYING WS-I FROM 1 BY 1
               UNTIL WS-I > 10
               COMPUTE WS-RESULT = WS-NOMBRE * WS-I
               DISPLAY WS-NOMBRE ' x ' WS-I ' = ' WS-RESULT
           END-PERFORM.
```

■ Résultat attendu

La table de multiplication de 7 s'affiche de 7x1=7 à 7x10=70.

■ Conseils

- PERFORM 5 TIMES répète un bloc exactement 5 fois.
- Un paragraphe COBOL est identifié par un nom suivi d'un point.

Exercice 6 — Tables (Tableaux) avec OCCURS

Niveau : ■ Débutant

■ Objectif

Déclarer et manipuler des tableaux en COBOL avec la clause OCCURS.

■ Contexte

Les tableaux en COBOL sont déclarés avec OCCURS. L'accès se fait par un indice numérique (commence à 1).

■ Instructions

1. Déclarer un tableau WS-NOTES de 10 éléments PIC 9(3) avec OCCURS 10 TIMES.
2. Initialiser les 10 notes manuellement avec des MOVE.
3. Parcourir le tableau pour calculer la somme totale.
4. Calculer et afficher la moyenne, le max et le min.

■ Exemple / Structure de départ

```
01 WS-TABLEAU.
   05 WS-NOTES  PIC 9(3) OCCURS 10 TIMES.
01 WS-SOMME    PIC 9(6) VALUE 0.
01 WS-I        PIC 9(2).

    PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 10
        ADD WS-NOTES(WS-I) TO WS-SOMME
    END-PERFORM.
```

■ Résultat attendu

Affichage de la moyenne, du max et du min des 10 notes.

■ Conseils

- WS-NOTES(1) = premier élément. Les indices commencent à 1, pas à 0 !
- INITIALIZE WS-TABLEAU remet tous les éléments à leurs valeurs par défaut.

PARTIE 3 — Fichiers Séquentiels

Exercice 7 — Lecture d'un fichier séquentiel

Niveau : ■ Intermédiaire

■ Objectif

Lire un fichier séquentiel enregistrement par enregistrement et afficher son contenu.

■ Contexte

La gestion de fichiers repose sur FILE-CONTROL (association logique/physique), FILE SECTION (description enregistrement) et la PROCEDURE DIVISION (open/read/close).

■ Instructions

1. Créer un fichier FIC-EMPLOYES.dat : matricule(6) + nom(20) + prenom(15) + salaire(8) = 49 cars/ligne.
2. Dans FILE-CONTROL, utiliser ORGANIZATION IS SEQUENTIAL.
3. Décrire l'enregistrement dans la FILE SECTION.
4. Lire dans une boucle jusqu'à AT END. Compter et afficher chaque enregistrement + total.

■ Exemple / Structure de départ

```
        FILE-CONTROL.
            SELECT FIC-EMPLOYES ASSIGN TO 'FIC-EMPLOYES.dat'
                ORGANIZATION IS SEQUENTIAL.
        FD  FIC-EMPLOYES LABEL RECORDS ARE STANDARD.
        01  ENR-EMPLOYES.
            05 EMP-MATRICULE  PIC X(6).
            05 EMP-NOM        PIC X(20).
            05 EMP-SALAIRE    PIC 9(8).

            OPEN INPUT FIC-EMPLOYES.
            READ FIC-EMPLOYES AT END SET FIC-TERMINE TO TRUE END-READ.
            PERFORM UNTIL FIC-TERMINE
                DISPLAY EMP-MATRICULE ' ' EMP-NOM
                READ FIC-EMPLOYES AT END SET FIC-TERMINE TO TRUE END-READ
            END-PERFORM.
            CLOSE FIC-EMPLOYES.
```

■ Résultat attendu

Chaque employé s'affiche, suivi du nombre total d'enregistrements.

■ Conseils

- Toujours effectuer une première lecture (priming read) avant la boucle.
- OPEN INPUT pour lecture, OPEN OUTPUT pour écriture, OPEN I-O pour les deux.
- Le 88-level est élégant pour le booléen de fin de fichier.

Exercice 8 — Écriture et transformation de fichier

Niveau : ■ Intermédiaire

■ Objectif

Lire un fichier d'entrée, transformer les données et écrire le résultat dans un fichier de sortie.

■ Instructions

1. Reprendre FIC-EMPLOYES comme fichier d'entrée.
2. Créer FIC-RAPPORT avec : numéro de ligne, nom complet (prénom + nom), salaire formaté.
3. Augmenter le salaire de 5% pour les employés gagnant moins de 2000.
4. Écrire une ligne titre en tête et une ligne total en fin de fichier.
5. Compter et afficher le nombre d'employés augmentés.

■ Exemple / Structure de départ

```
01  ENR-RAPPORT.
    05 RPT-NUMERO    PIC ZZ9.
    05 FILLER        PIC X(2) VALUE SPACES.
    05 RPT-NOM-COMP  PIC X(36).
    05 RPT-SALAIRE   PIC ZZZ,ZZ9.

    STRING EMP-PRENOM DELIMITED SPACE
           ' '        DELIMITED SIZE
           EMP-NOM    DELIMITED SPACE
           INTO RPT-NOM-COMP.

    IF EMP-SALAIRE < 2000
        COMPUTE EMP-SALAIRE = EMP-SALAIRE * 1.05
        ADD 1 TO WS-NB-AUGMENTES
    END-IF.
```

■ Résultat attendu

Un fichier rapport formaté avec titre, données et total.

■ Conseils

- STRING concatène plusieurs champs. UNSTRING fait l'inverse.
- PIC ZZZ,ZZ9 supprime les zéros de tête et ajoute le séparateur milliers.
- FILLER est un champ anonyme pour le remplissage.

Exercice 9 — Tri et traitement de rupture

Niveau : ■ Intermédiaire

■ Objectif

Utiliser le verbe SORT pour trier un fichier et implémenter un traitement de rupture avec sous-totaux.

■ Contexte

Le traitement de rupture détecte les changements de valeur d'une clé pour calculer des sous-totaux par groupe.

■ Instructions

1. Créer FIC-VENTES : code_region(2) + code_vendeur(4) + montant_vente(9).
2. Trier par code_region puis code_vendeur avec SORT.
3. Parcourir le fichier trié et calculer un sous-total par vendeur et par région.
4. Afficher un rapport avec sous-totaux et total général.

■ Exemple / Structure de départ

```
SD  WRK-VENTES.
01  WRK-ENR.
    05 WRK-REGION   PIC X(2).
    05 WRK-VENDEUR  PIC X(4).
    05 WRK-MONTANT  PIC 9(9).

    SORT WRK-VENTES
        ASCENDING KEY WRK-REGION WRK-VENDEUR
        USING  FIC-VENTES
        GIVING FIC-VENTES-TRIE.

    IF WRK-REGION NOT = WS-REGION-PREC
        PERFORM SOUS-TOTAL-REGION
    END-IF.
```

■ Résultat attendu

Rapport structuré avec sous-totaux par vendeur, par région et total général.

■ Conseils

- Initialisez les clés précédentes avec les valeurs du premier enregistrement avant la boucle.
- L'ordre de traitement des ruptures va du plus fin (vendeur) au plus large (région).

PARTIE 4 — COBOL et SQL (ocesql + PostgreSQL)

Exercice 10 — SELECT simple en COBOL/SQL

Niveau : ■ Avancé

■■ *Cet exercice nécessite le Docker mis à jour avec ocesql + PostgreSQL. Voir le Dockerfile fourni dans le repo.*

■ Objectif

Intégrer des requêtes SQL dans un programme COBOL pour lire des données depuis une table PostgreSQL via ocesql.

■ Contexte

ocesql est un précompilateur qui transforme les blocs EXEC SQL ... END-EXEC en appels de fonctions C, puis compile le tout avec GNU COBOL. La syntaxe est très proche du DB2 mainframe.

■ Instructions

1. Vérifier que ocesql est disponible : ocesql --version dans le Docker.
2. Créer la table EMPLOYES dans PostgreSQL (voir script SQL fourni).
3. Déclarer les variables hôtes et le bloc INCLUDE SQLCA dans WORKING-STORAGE.
4. Écrire un SELECT INTO pour récupérer un employé par matricule.
5. Tester SQLCODE : 0=trouvé, 100=non trouvé, négatif=erreur. Afficher le résultat adapté.

■ Exemple / Structure de départ

```
       WORKING-STORAGE SECTION.
       EXEC SQL INCLUDE SQLCA     END-EXEC.

       01 WS-MATRICULE  PIC X(6).
       01 WS-NOM        PIC X(20).
       01 WS-SALAIRE    PIC S9(8)V99.

           EXEC SQL
               SELECT NOM, SALAIRE
               INTO :WS-NOM, :WS-SALAIRE
               FROM EMPLOYES
               WHERE MATRICULE = :WS-MATRICULE
           END-EXEC.

           EVALUATE SQLCODE
               WHEN 0   DISPLAY 'Trouvé : ' WS-NOM
               WHEN 100 DISPLAY 'Employé non trouvé'
               WHEN OTHER DISPLAY 'Erreur SQL : ' SQLCODE
           END-EVALUATE.
```

■ Résultat attendu

Le programme affiche les données ou un message d'erreur approprié.

■ Conseils

- Compiler avec ocesql : ocesql mon.pco mon.cob && cobc -x mon.cob -locesql
- SELECT INTO ne fonctionne que si la requête retourne exactement 1 ligne.
- Toujours tester SQLCODE après chaque ordre SQL.

Exercice 11 — Curseur et traitement de masse

Niveau : ■ Avancé

■■ *Nécessite le Docker mis à jour avec ocesql + PostgreSQL.*

■ Objectif

Utiliser un curseur pour traiter plusieurs lignes d'une table en COBOL.

■ Contexte

Quand une requête retourne plusieurs lignes, il faut un curseur. Étapes : DECLARE → OPEN → FETCH en boucle → CLOSE.

■ Instructions

1. Déclarer un curseur C-EMPLOYES sélectionnant tous les employés triés par salaire DESC.
2. Ouvrir le curseur, lire les enregistrements un par un avec FETCH.
3. Boucler jusqu'à SQLCODE = 100 (fin de données).
4. Pour chaque employé : calculer un bonus de 10% et écrire une ligne dans un fichier de sortie.
5. Afficher le nombre total de lignes traitées et la masse salariale totale.

■ Exemple / Structure de départ

```
EXEC SQL
    DECLARE C-EMPLOYES CURSOR FOR
    SELECT MATRICULE, NOM, SALAIRE
    FROM EMPLOYES ORDER BY SALAIRE DESC
END-EXEC.

EXEC SQL OPEN C-EMPLOYES END-EXEC.
PERFORM UNTIL SQLCODE = 100
    EXEC SQL
        FETCH C-EMPLOYES
        INTO :WS-MAT, :WS-NOM, :WS-SAL
    END-EXEC
    IF SQLCODE = 0
        PERFORM TRAITEMENT-EMPLOYE
    END-IF
END-PERFORM.
EXEC SQL CLOSE C-EMPLOYES END-EXEC.
```

■ Résultat attendu

Fichier de sortie avec tous les employés, leurs bonus et les totaux.

■ Conseils

- FOR UPDATE OF colonne dans DECLARE CURSOR permet de modifier les lignes fetchées.
- WHERE CURRENT OF curseur dans un UPDATE modifie la ligne courante.
- Toujours fermer le curseur même en cas d'erreur (paragraphe de fin commun).

Exercice 12 — UPDATE, INSERT, DELETE et transactions

Niveau : ■ Avancé

■■ *Nécessite le Docker mis à jour avec ocesql + PostgreSQL.*

■ Objectif

Maîtriser les modifications de données avec une gestion robuste des erreurs et des transactions.

■ Instructions

1. Lire un fichier de mouvements : code_action (I/U/D) + données employé.
2. Selon le code : I=INSERT, U=UPDATE salaire, D=DELETE.
3. Après chaque groupe de 100 opérations, effectuer un COMMIT.
4. En cas d'erreur SQL (SQLCODE < 0) : ROLLBACK + écriture dans un fichier de rejet.
5. Afficher en fin : nombre d'insérés, mis à jour, supprimés, rejetés.

■ Exemple / Structure de départ

```
            EVALUATE WS-CODE-ACTION
                WHEN 'I'
                    EXEC SQL INSERT INTO EMPLOYES
                        VALUES (:WS-MAT, :WS-NOM, :WS-SAL)
                    END-EXEC
                WHEN 'U'
                    EXEC SQL UPDATE EMPLOYES
                        SET SALAIRE = :WS-SAL
                        WHERE MATRICULE = :WS-MAT
                    END-EXEC
                WHEN 'D'
                    EXEC SQL DELETE FROM EMPLOYES
                        WHERE MATRICULE = :WS-MAT
                    END-EXEC
            END-EVALUATE.
            IF SQLCODE < 0
                EXEC SQL ROLLBACK END-EXEC
                PERFORM ECRITURE-REJET
            END-IF.
```

■ Résultat attendu

Programme traitant tous les mouvements avec commit par lot et fichier de rejet.

■ Conseils

- COMMIT valide. ROLLBACK annule jusqu'au dernier COMMIT.
- Un UPDATE sans WHERE met à jour TOUTES les lignes — toujours vérifier !

PARTIE 5 — Niveau Expert

Exercice 13 — Programme ETL complet : fichier vers base de données

Niveau : ■ Expert

■■ *Les parties SQL nécessitent le Docker mis à jour avec ocesql + PostgreSQL.*

■ Objectif

Concevoir un programme ETL (Extract, Transform, Load) qui lit des fichiers sources, applique des règles métier et charge les données en base avec reporting.

■ Contexte

Un ETL industriel gère : validation des données, rejets avec codes erreur, jointures entre fichiers, agrégations et reporting complet.

■ Instructions

1. Lire un fichier EMPLOYES et un fichier SERVICES (code + libellé + budget).
2. Charger SERVICES en mémoire dans un tableau OCCURS 50 TIMES.
3. Pour chaque employé : valider le matricule (INSPECT), vérifier que le service existe (SEARCH).

4. Calculer le salaire chargé (salaire * 1.42 pour charges patronales).
5. Insérer les valides en base, décrémenter le budget du service correspondant.
6. Écrire un fichier de rejet avec code erreur pour les invalides.
7. En fin : COMMIT + rapport COUNT(*) et SUM(salaire_charge) par service.

■ Exemple / Structure de départ

```
       01 WS-TABLE-SERVICES.
          05 WS-SERVICE OCCURS 50 TIMES INDEXED BY WS-IDX.
             10 WS-SVC-CODE   PIC X(4).
             10 WS-SVC-LIB    PIC X(30).
             10 WS-SVC-BUDGET PIC S9(12)V99.

           SET WS-IDX TO 1.
           SEARCH WS-SERVICE
               AT END MOVE 'E02' TO WS-CODE-REJET
               WHEN WS-SVC-CODE(WS-IDX) = EMP-CODE-SVC
                   MOVE 'OK' TO WS-STATUT
           END-SEARCH.
```

■ Résultat attendu

Programme industriel avec ETL, validation, rejets, transactions et reporting.

■ Conseils

- SEARCH ALL (recherche binaire) est plus performant mais nécessite OCCURS ... ASCENDING KEY.
- Utilisez des 88-levels pour les codes erreur : 88 ERR-MAT-INVALIDE VALUE 'E01'.
- Un COMMIT toutes les N lignes évite les verrous trop longs en production.
- Chaque règle métier doit être documentée avec des commentaires (*) dans le code.

Exercice 14 — Débogage d'un programme existant

Niveau : ■ Expert

■ Objectif

Analyser, corriger et optimiser un programme COBOL présentant des bugs et des problèmes de performance.

■ Contexte

Exercice de code review : le programme ci-dessous compile mais produit des résultats incorrects. Il contient 6 bugs et 3 problèmes de performance à identifier et corriger.

■ Instructions

1. Analyser le code suivant et identifier tous les problèmes.
2. Pour chaque problème : noter la ligne approximative, décrire le bug, proposer la correction.
3. Corriger le programme et valider que les résultats sont corrects.
4. Estimer l'impact des optimisations de performance.

■ Exemple / Structure de départ

```
       * BUG 1 — Boucle infinie potentielle
            PERFORM UNTIL WS-COMPTEUR = 1000
                READ FIC-INPUT AT END MOVE 'O' TO WS-FIN
                ADD ENR-MONTANT TO WS-TOTAL
            END-PERFORM.

       * BUG 2 — Division sans protection contre le zéro
            DIVIDE WS-TOTAL BY WS-COMPTEUR GIVING WS-MOYENNE.

       * BUG 3 — Curseur non fermé en cas d'erreur SQL
            EXEC SQL OPEN C-DONNEES END-EXEC.
            PERFORM FETCH-LOOP.
            EXEC SQL CLOSE C-DONNEES END-EXEC.

       * BUG 4 — UPDATE sans WHERE (toutes les lignes !)
            EXEC SQL
                UPDATE MA_TABLE SET STATUT = 'T'
            END-EXEC.

       * PERF — SELECT en boucle 10 000 fois
            PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 10000
                EXEC SQL SELECT VAL INTO :WS-VAL
                FROM CONFIG WHERE CODE = :WS-CODE END-EXEC
            END-PERFORM.
```

■ Résultat attendu

Liste des 6 bugs et 3 problèmes de performance identifiés et corrigés.

■ Conseils

- Bug 1 : la condition d'arrêt doit tester WS-FIN, pas un compteur fixe.
- Bug 2 : tester WS-COMPTEUR > 0 avant de diviser.
- Bug 3 : utiliser un paragraphe FIN-PGM commun pour le CLOSE.
- Bug 4 : toujours une clause WHERE sur UPDATE/DELETE.
- Perf : charger CONFIG en mémoire une seule fois avant la boucle.
- Cherchez aussi : dépassement de PIC, SQLCODE non testé, variable non initialisée.

Exercice 15 — Architecture modulaire : CALL et sous-programmes

Niveau : ■ Expert

■ Objectif

Concevoir une architecture COBOL modulaire avec programme principal et sous-programmes indépendants via CALL.

■ Contexte

Les grandes applications mainframe sont découpées en modules réutilisables. CALL 'NOM' USING passe des paramètres par référence (BY REFERENCE) ou par valeur (BY CONTENT).

■ Instructions

1. Créer PGMPRINC (programme principal) qui orchestre le traitement.
2. Créer VALIDEMP : valide un enregistrement employé, retourne un code retour (0=OK, 8=erreur).
3. Créer CALCSAL : calcule les éléments de paie (brut, charges, net).
4. Créer LOGERROR : écrit dans un fichier de log centralisé.
5. Le programme principal appelle ces modules via CALL. Définir une LINKAGE SECTION dans chaque sous-programme.
6. Tester la chaîne complète et vérifier la bonne transmission des paramètres.

■ Exemple / Structure de départ

```
      * === PROGRAMME PRINCIPAL ===
       01 WS-RETOUR  PIC S9(4) COMP.

           CALL 'VALIDEMP' USING BY REFERENCE WS-EMPLOYE
                                 BY REFERENCE WS-RETOUR.
           IF WS-RETOUR = 0
               CALL 'CALCSAL' USING BY REFERENCE WS-EMPLOYE
                                    BY REFERENCE WS-PAIE
           ELSE
               CALL 'LOGERROR' USING BY CONTENT WS-RETOUR
           END-IF.

      * === SOUS-PROGRAMME VALIDEMP ===
       LINKAGE SECTION.
       01 LS-EMPLOYE.
          05 LS-MATRICULE  PIC X(6).
       01 LS-RETOUR  PIC S9(4) COMP.

       PROCEDURE DIVISION USING LS-EMPLOYE LS-RETOUR.
           MOVE 0 TO LS-RETOUR.
           IF LS-MATRICULE = SPACES MOVE 8 TO LS-RETOUR END-IF.
           GOBACK.
```

■ Résultat attendu

Architecture modulaire fonctionnelle avec 4 modules indépendants et testables.

■ Conseils

- BY REFERENCE : l'appelant voit les modifications. BY CONTENT : copie protégée.
- GOBACK est préféré à STOP RUN dans un sous-programme.
- Convention standard : 0=OK, 4=warning, 8=erreur, 12=fatal.
- La LINKAGE SECTION ne réserve pas de mémoire, elle pointe vers la zone de l'appelant.