Les sous-programmes : exercices résolus
Corrigé

Objectifs

- Savoir écrire des sous-programmes ;
- Comprendre les modes de passage de paramètres ;
- Faire la différence entre fonction et procédure ;
- Continuer à appliquer les principes des semaines 1 & 2.

Exercice 1 : Notion de produit *1*
Exercice 2 : Notion de ligne de facture *2*
Exercice 3 : Notion de facture *4*
Exercice 4 : Programme de test *7*

L'objectif de ces exercices est de réaliser l'édition d'une facture dans une version relativement simplifiée puisqu'il s'agit de pouvoir ajouter et/ou supprimer des lignes sur une facture et de l'éditer à l'écran.

Nous ne traiterons pas l'interface homme/machine (IHM) et nous nous contenterons donc de développer les types et sous-programmes qui constituent ce modèle.

Exercice 1 : Notion de produit

Commençons par définir le type Produit et les opérations associées.

1.1 Écrire le type Produit sachant qu'un produit est caractérisé par un code (sur 3 caractères), un libellé (9 caractères) et un prix hors taxe.

Solution : Le type Produit est un enregistrement composé de trois champs un code de type chaîne de 3 caractères, un libellé (chaîne de 9 caractères) et un prix hors taxe (**Réel**).

C

```
#define LG_CODE    3    /* longueur maximale d'un code */
#define LG_LIBELLE 9    /* longueur maximale d'un libellé */

typedef char Code[LG_CODE+1];
typedef char Libelle[LG_LIBELLE+1];

struct Produit {
    Code code;
    Libelle libelle;
    double prix_ht;
};

typedef struct Produit Produit;
```

1.2 Écrire un sous-programme pour initialiser un produit à partir de son code, son libellé et son prix hors taxe.

Solution : Le but du sous-programme est « initialiser un produit à partir de son code, son libellé, et son prix hors taxe ».

Les paramètres sont donc :

- le produit à initialiser : **out** ;
- le code du produit : **in** ;
- le libellé du produit : **in** ;
- le prix du produit : **in**.

On constate qu'il n'y a qu'un seul paramètre en sortie et tous les autres sont en entrée. Nous en faisons quand même une procédure car il s'agit d'un sous-programme d'initialisation. Il est préférable dans ce cas que la mémoire soit fournie par le programme appelant (ceci évite par exemple de faire des copies inutiles lors du retour de la fonction).

On peut donc en déduire la spécification.

```
1 Procédure initialiser_produit(p: out Produiti ;
2             c: in Code ; l: in Libellé ; prix_ht: in Réel) Est
3     -- Initialiser le produit p à partir de son code c, de son libellé l
4     -- et de son prix hors taxe.
```

On peut maintenant en déduire le code.

```
1     Début
2         produit.code ← c
3         produit.libellé ← l
4         produit.prix_ht ← prix_ht
5     Fin
```

C

```
1 void initialiser_produit(Produit *p, Code c, Libelle l, double prix_ht)
2 {
3     strncpy(p->code, c, LG_CODE);
4     p->code[LG_CODE] = '\0';    /* être sûr d'avoir le zéro terminal */
5     strncpy(p->libelle, l, LG_LIBELLE);
6     p->libelle[LG_LIBELLE] = '\0'; /* être sûr d'avoir le zéro terminal */
7     p->prix_ht = prix_ht;
8 }
```

En C, on ne peut pas faire directement l'affectation des chaînes de caractères. Il faut en fait copier les caractères de l'une dans l'autre. Nous utilisons la fonction `strncat` de `string.h` qui permet de tenir compte de la capacité de la chaîne destination. On force ensuite un caractère nul pour être sûr que la chaîne, même dans le cas où la copie a été tronquée, a le zéro final.

Exercice 2 : Notion de ligne de facture

Le type LigneFacture permet de représenter une ligne d'une facture. Il est caractérisé par le produit concerné, la quantité commandée et le prix total hors taxe de la ligne.

2.1 Définir le type LigneFacture.

Solution : Le type LigneFacture est également un enregistrement. Notons la redondance entre le prix hors taxe de la ligne et des informations prix hors taxe d'un produit et quantité commandée.

C

```
1 struct LigneFacture {
2     Produit produit;
3     int quantite;        /* > 0 */
4     double prix_ht;      /* prix_ht = quantite * produit.prix_ht */
5 };
6
7 typedef struct LigneFacture LigneFacture;
```

2.2 Définir une opération qui permet d'initialiser une ligne de facture à partir du produit concerné et de la quantité commandée. Le prix total hors taxe de la ligne peut être déduit de ces deux informations puisqu'il correspond au prix hors taxe du produit multiplié par la quantité commandée.

Solution : L'initialisation d'une ligne de facture est réalisée sur le même principe que le produit.

C

```
1 /* Initialiser une ligne de facture à partir du produit concerné et de
2  * la quantité commandée.
3  */
4 void initialiser_ligne(LigneFacture *ligne, Produit p, int quantite)
5 {
6     ligne->produit = p;
7     ligne->quantite = quantite;
8     ligne->prix_ht = p.prix_ht * quantite;
9 }
```

2.3 Écrire une opération qui affiche une ligne de facture. Les informations sont affichées de manière tabulée avec dans l'ordre le code du produit, son libellé, son prix hors taxe, la quantité commandé, le prix total hors taxe et le prix total TTC. On supposera le taux de TVA constant et indépendant des produits.

Voici un exemple de ligne affichée.

```
1 | 001 |       Truc |   99.00 | 10 |  990.00 | 1184.04 |
```

Solution : Ce sous-programme a un seul paramètre en entrée, la ligne de facture. Il s'agit donc d'une procédure.

C

```
1 /* Afficher une ligne de facture.
2  */
3 void afficher_ligne(LigneFacture ligne)
4 {
5     double prix_ttc;
6
7     /* calculer le prix ttc */
8     prix_ttc = ligne.prix_ht * (1 + TVA);
9
10    printf("| %3s | %10s | %7.2f | %2d | %7.2f | %7.2f |\n",
11           ligne.produit.code, ligne.produit.libelle, ligne.produit.prix_ht,
12           ligne.quantite, ligne.prix_ht, prix_ttc);
13 }
```

Exercice 3 : Notion de facture

Définissons enfin le type Facture. Une facture contient plusieurs lignes (20 au maximum). Elle est caractérisée par un numéro de facture.

3.1 Définir le type Facture.

Solution : Le type Facture est un tableau de lignes de facture. On ne sait pas à priori combien il y aura de lignes mais on sait qu'il y en a moins de 20. On peut donc prendre un tableau de capacité 20 et il faudra gérer la taille effective. Ces deux informations sont donc regroupées dans un enregistrement

C

```c
#define NB_LIGNES  20   /* nombre maximal de lignes sur une facture */

struct Facture {
    int numero;
    LigneFacture lignes[NB_LIGNES];
    int nb_lignes;      /* nombre effectif de lignes sur la facture */
};

typedef struct Facture Facture;
```

3.2 Écrire une opération qui permet d'initialiser une facture à partir de son numéro. Une telle facture est bien entendu vierge et ne contient donc aucune ligne.

Solution : Ce sous-programme a un seul paramètre, la facture à initialiser. C'est un paramètre en sortie. On fait donc une procédure.

C

```c
/* Initialiser une facture.
 *
 * Assure
 *      une_facture.nb_lignes == 0
 */
void initialiser_facture(Facture *une_facture, int son_numero)
{
    une_facture->numero = son_numero;
    une_facture->nb_lignes = 0;

    assert(une_facture->nb_lignes == 0);
}
```

3.3 Écrire un sous-programme qui permet de calculer le montant total hors taxe de la facture. Il s'agit de faire la somme des totaux hors taxe des lignes de la facture.

Solution : Ce sous-programme prend en paramètre une facture (entrée) et fournit le total hors taxe de la facture (sortie). Il y a donc un seul paramère en sortie, d'autres qui sont tous en entrée. On en fait donc une fonction.

```
Fonction montant_total_ht_facture(une_facture: Facture): Réel Est
    -- Le montant total hors taxe de la facture une_facture.
```

C

```
/* Déterminer le montant total HT d'une facture */
double montant_total_ht_facture(Facture une_facture)
{
    double resultat = 0;
    int l;      /* parcourir les lignes de la facture */
    for (l = 0; l < une_facture.nb_lignes; l++) {
        resultat += une_facture.lignes[l].prix_ht;
    }
    return resultat;
}
```

Notons qu'en C, on a défini une variable locale `resultat` qui sert à accumuler le total des lignes.

3.4 Écrire un sous-programme pour ajouter une ligne à une facture, c'est-à-dire un produit et la quantité commandée.

Solution : Ce sous-programme prend en paramètre la facture à modifier (**in out**), le produit à ajouter (**in**) et la quantité commandée (**in**). On fait donc une procédure.

Notons que pour l'implantation de cette procédure, on peut utiliser l'opération `initialiser_ligne` déjà définie.

C

```
/* Ajouter une ligne à une facture.
 * Nécessite
 *      une_facture.nb_lignes < NB_LIGNES;
 */
void ajouter_ligne(Facture *une_facture, Produit p, int quantite)
{
    assert(une_facture->nb_lignes < NB_LIGNES);
    initialiser_ligne(&une_facture->lignes[une_facture->nb_lignes],
            p, quantite);
    une_facture->nb_lignes++;
}
```

3.5 Écrire un sous-programme qui permet d'éditer une facture. Il s'agit d'afficher un entête portant le numéro de la facture. Ensuite sont affichées toutes les lignes et enfin le total hors taxe et TTC de la facture.

Voici un exemple d'édition d'une facture.

```

Édition de la facture 1
-------------------------------------------------------
| 001 |       Truc |   99.00 | 10 |  990.00 | 1184.04 |
| 020 |      Chose |   15.00 |  2 |   30.00 |   35.88 |
| 163 |     Bidule |  250.00 |  1 |  250.00 |  299.00 |
-------------------------------------------------------
                                  | 1270.00 | 1518.92 |
```

Solution : Ce sous-programme prend un seul paramètre, la facture à éditer (**in**). Nous faisons une procédure.

C

```c
/* Éditer à l'écran la facture.
 */
void editer_facture(Facture une_facture)
{
    int l;      /* parcourir les lignes de la facture */
    double total_ht;    /* total ht de la facture */
    double total_ttc;   /* total ttc de la facture */

    /* Calculer les totaux */
    total_ht = montant_total_ht_facture(une_facture);
    total_ttc = total_ht * (1 + TVA);

    /* Afficher l'entête */
    printf("\n\n");
    printf("Édition de la facture %d\n", une_facture.numero);
    printf("-------------------------------------------------------\n");

    /* Afficher les lignes */
    for (l = 0; l < une_facture.nb_lignes; l++) {
        afficher_ligne(une_facture.lignes[l]);
    }

    /* Afficher le total */
    printf("-------------------------------------------------------\n");
    printf("%33s | %7.2f | %7.2f |\n", "", total_ht, total_ttc);
}
```

3.6 Écrire un sous-programme pour supprimer une ligne d'une facture en fonction de son numéro. Bien sûr, la première ligne porte le numéro 1.

Solution : Ce sous-programme a pour paramètres la facture à modifier (**in out**) et le numéro de la ligne à supprimer (**in**). On fait donc une procédure.

C

```c
/* Supprimer une ligne de la facture.  Le numéro n_ligne correspond à la ligne
 * de la facture qu'il faut supprimer.
 *
 * Nécessite
 *      1 <= n_ligne && n_ligne <= une_facture->nb_lignes;
 */
void supprimer_ligne(Facture *une_facture, int n_ligne)
{
    int l;      /* parcourir les lignes à décaler */

    assert(1 <= n_ligne && n_ligne <= une_facture->nb_lignes);

    /* Décaler les lignes */
    for (l = n_ligne; l < une_facture->nb_lignes; l++) {
        une_facture->lignes[l-1] = une_facture->lignes[l];
    }
    une_facture->nb_lignes--;
}
```

Exercice 4 : Programme de test

Écrire un programme de test pour les types et opérations définis dans les exercices précédents.

Remarque : Même si c'est le dernier exercice, il est souhaitable (et nécessaire) de le faire en même temps que les autres, au fur et à mesure que les sous-programmes sont écrits. En effet, tester au fur et à mesure permet de tester de manière plus exhaustive (donc de détecter plus d'erreurs), de localiser plus vide la cause de l'erreur et donc de la corriger plus facilement.

Solution :

C

```
1  /* Programme principal, en fait programme de test !
2   * Attention : ce programme de test n'est absolument pas complet !
3   */
4  int main()
5  {
6      Facture f1;
7      Produit p1, p2, p3;
8
9      initialiser_produit(&p1, "001", "Truc", 99);
10     initialiser_produit(&p2, "020", "Chose", 15);
11     initialiser_produit(&p3, "163", "Bidule", 250);
12
13     /* Création de la facture en ajoutant les lignes */
14     initialiser_facture(&f1, 1);
15     editer_facture(f1);
16     ajouter_ligne(&f1, p1, 10);
17     editer_facture(f1);
18     ajouter_ligne(&f1, p2, 2);
19     editer_facture(f1);
20     ajouter_ligne(&f1, p3, 1);
21     editer_facture(f1);
22
23     /* Suppression de la deuxième ligne */
24     supprimer_ligne(&f1, 2);
25     editer_facture(f1);
26
27     return EXIT_SUCCESS;
28 }
```