

Les types utilisateurs : exercices résolus en C

Corrigé

Objectifs

- Connaître les types utilisateurs : tableaux, enregistrements et type énumérés ;
- Connaître les algorithmes fondamentaux sur les tableaux ;
- Continuer à appliquer les principes de la semaine 1.

Exercice 1 : Occurrences des chiffres d'un entier	1
Exercice 2 : Modéliser un robot de type 1	4
Exercice 3 : Lecture d'un tableau	8
Exercice 4 : Tri par insertion séquentielle	10

Exercice 1 : Occurrences des chiffres d'un entier

Écrire un programme qui compte le nombre d'occurrences des 10 chiffres dans un entier naturel donné.

Par exemple, l'entier 4214 a une occurrence du chiffre 1, une de 2 et deux de 4.

Solution :

1 **R0** : Compter le nombre d'occurrences des chiffres d'un nombre

Puisque l'objectif est de compter le nombre d'occurrences de chacun des chiffres, l'idée est donc de prendre autant de compteur que de chiffre. Dans ce cas, on prend un tableau de compteur indicé par les chiffres.

```
1 nb: Tableau [0..9] De Entier
2   -- nb[i] = nb d'occurrences de i dans le nombre.
```

On peut alors en déduire l'algorithme suivant :

```
1 R1 : Raffinage De « Compter le nombre d'occurrences des chiffres d'un nombre »
2   | Initialiser les compteurs à 0
3   | Répéter
4   |   | Comptabiliser l'unité de nombre
5   |   | Supprimer l'unité de nombre
6   | JusquÀ nombre = 0
7
8
9 R2 : Raffinage De « Initialiser les compteurs à 0 »
10  | Pour i <- 0 JusquÀ i = 9 Faire
11  |   nb[i] <- 0
12  | FinPour
13
14 R2 : Raffinage De « Comptabiliser l'unité de nombre »
```

```

15 | chiffre = nombre Mod 10
16 | nb[chiffre] <- nb[chiffre] + 1
17
18 R2 : Raffinage De « Supprimer l'unité de nombre »
19 | nombre <- nombre Div 10

1 /*****
2 * Auteur : Xavier Crégut <cregut@enseeiht.fr>
3 * Version : 1.3
4 * Objectif : Compter le nb d'occurrences des chiffres d'un nombre
5 *****/
6
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <assert.h>
10
11 int main()
12 {
13     int nombre;          /* entier saisi par l'utilisateur */
14     int nb[10]; /* nb[i] : nb d'occurrences du chiffre i dans nombre */
15     int i;              /* pour parcourir les chiffres */
16
17     /* saisir le nombre */
18     printf("Nombre_(>_0)_:_");
19     scanf("%d", &nombre);
20
21     assert(nombre >= 0);
22     /* Arrête le programme si !(nombre > 0). Ceci est acceptable car il
23      * s'agit d'un programme de test qui n'a donc pas à être robuste.
24      */
25
26     /* initialiser les compteurs */
27     for (i = 0; i < 10; i++) {
28         nb[i] = 0;
29     }
30
31     /* comptabiliser chaque chiffre de nombre */
32     do {
33         int chiffre; /* chiffre unité de nombre */
34
35         /* comptabiliser l'unité de nombre */
36         chiffre = nombre % 10;
37         nb[chiffre]++;
38
39         /* supprimer l'unité de nombre */
40         nombre = nombre / 10;
41
42     } while (nombre > 0);
43
44     /* afficher le nombre d'occurrences des chiffres */
45     for (i = 0; i < 10; i++) {
46         printf("nb_de_%d:_%d\n", i, nb[i]);

```

```
47     }  
48  
49     return EXIT_SUCCESS;  
50 }
```

Exercice 2 : Modéliser un robot de type 1

L'objectif de cet exercice est de modéliser un robot de type 1. Un tel robot se déplace dans un environnement qui peut être modélisé par un quadrillage dont chaque case correspond à une position possible du robot.

Un robot est donc caractérisé par sa position (son abscisse, x , et son ordonnée, y) et sa direction (nord, sud, est ou ouest). La figure 1 décrit un robot à la position $x = 4$ et $y = 2$, sa direction est « ouest ».

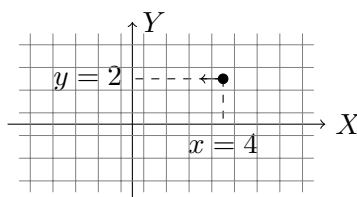


FIGURE 1 – Le robot dans un environnement illimité

Les robots de type 1 ont des possibilités réduites qui se limitent à pivoter de 90° vers la droite et à avancer d'une case suivant sa direction courante.

Pour simplifier, on considère que le robot évolue dans un environnement infini et sans obstacle.

2.1 Définir les types nécessaires pour modéliser un robot de type 1.

Solution : Un robot de type 1 est caractérisé par :

- une position, c'est-à-dire une abscisse (x) et une ordonnée. On peut donc définir un type `Position` comme un enregistrement.
- une direction. La direction peut prendre 4 valeurs que l'on peut par exemple appeler Nord, Sud, Est et Ouest. On aurait également pu les appeler Gauche, Droite, Haut et Bas. En conséquence, il s'agit d'un type énuméré.

Le type `Robot1` est donc un enregistrement qui regroupe ces deux informations.

```

1  Type
2      Position =
3          Enregistrement
4              x: Entier    -- abscisse
5              y: Entier    -- ordonnée
6          FinEnregistrement
7
8      Direction = (NORD, SUD, EST, OUEST)
9
10     Robot1 =
11         Enregistrement
12             position: Position
13             direction: Direction
14         FinEnregistrement

```

2.2 L'environnement dans lequel évolue le robot est en fait fini et comporte des obstacles. On suppose qu'un obstacle occupe une case du quadrillage. La question est donc de savoir si la case est libre ou contient un obstacle.

Définir un type pour modéliser l'environnement.

Solution : L'environnement dans lequel évolue le robot est un quadrillage, donc il peut être modélisé par un tableau à deux dimensions. Nous utilisons deux constantes pour en préciser les dimensions (NB_COLONNES et NB_LIGNES).

Chaque case du tableau est soit une position qui peut être occupé par le robot, soit un mur, donc deux valeurs possibles. On peut donc représenter cette information par un type booléen (il faut choisir ce que signifie **VRAI** et **FAUX**) ou définir un type énuméré.

Nous choisissons ici cette deuxième solution :

```

1  Constante
2      NB_COLONNES = 20
3      NB_LIGNES = 30
4
5  Type
6      Case = (LIBRE, MUR)
7          -- indique si la case est LIBRE et peut donc être occupé par le robot
8          -- ou si elle contient un mur
9
10     Environnement = Tableau [1..NB_COLONNES, 1..NB_LIGNES] De Case

```

2.3 Écrire un programme qui fait avancer un robot toujours tout droit jusqu'à ce qu'il rencontre un obstacle ou qu'il arrive à la limite de son environnement. On supposera que le robot est initialement sur une position valide avec pour direction « est ».

Solution : On considère un robot particulier dans un environnement donné. Dans les programmes de test, il s'agira donc d'initialiser ces deux informations de manière à ce que l'on puisse tester notre algorithme dans des cas significatifs. Notons que la direction est fixé à l'EST.

```

1  R0 : Faire avancer le robot au maximum
2      robot: in Robot1, environnement: in Environnement
3
4  tests :
5      - pas de mur => la limite est atteinte
6      - un mur => s'arrête devant le mur
7
8  R1 : Raffinage De « Faire avancer le robot au maximum »
9      TantQue (limite environnement non atteinte) et (Pas de mur) Faire
10     Faire avancer le robot d'une case (vers l'est)
11     FinTQ
12
13 R2 : Raffinage De « limite environnement non atteinte »
14     Résultat <- robot.position.x < NB_COLONNES
15
16 R2 : Raffinage De « Pas de mur »
17     Résultat <- environnement[robot.position.x + 1, robot.position.y] <> MUR
18
19 R2 : Raffinage De « Faire avancer le robot d'une case »
20     robot.position.x <- robot.position.x + 1

```

Attention, en passant en C les tableaux ne commencent plus à 1 mais à 0. Il faut donc faire une translation des indices !

```
1  /*****
2  *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3  *  Version  : 1.2
4  *  Objectif : Faire avancer un robot vers l'est au maximum
5  *****/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define NB_COLONNES 20
11 #define NB_LIGNES 30
12
13 struct Position {
14     int x;      /* abscisse */
15     int y;      /* ordonnée */
16 };
17
18 typedef struct Position Position;
19
20 enum Direction { NORD, SUD, EST, OUEST };
21
22 typedef enum Direction Direction;
23
24 struct Robot1 {
25     Position position;
26     Direction direction;
27 };
28
29 typedef struct Robot1 Robot1;
30
31 enum Case { LIBRE, MUR };
32
33 typedef enum Case Case;
34
35 typedef Case Environnement[NB_COLONNES][NB_LIGNES];
36
37 int main()
38 {
39     Robot1 robot;
40     Environnement environnement;
41
42     /* Initialiser l'environnement */
43     printf("Initialiser_l'environnement...\n");
44     {
45         int l, c; /* parcourir les lignes et colonnes de l'environnement */
46         for (l = 0; l < NB_COLONNES; l++) {
47             for (c = 0; c < NB_LIGNES; c++) {
48                 environnement[l][c] = LIBRE;
49             }
50         }
51         /* Mettre un mur */
52         environnement[10][0] = MUR;
```

```
53     }
54
55     /* Initialiser le robot */
56     printf("Initialiser_le_robot...\n");
57     robot.direction = EST;
58     robot.position.x = 0;
59     robot.position.y = 0;
60
61     /* Faire avancer le robot */
62     while (robot.position.x < NB_COLONNES - 1 /* limite non atteinte */
63           && environnement[robot.position.x+1][robot.position.y] != MUR)
64           /* pas de mur */
65     {
66         /* faire avancer le robot d'une case (vers l'est) */
67         robot.position.x++;
68
69         /* Afficher la position du robot */
70         printf("Le_robot_est_en_(%d,_%d)\n", robot.position.x, robot.position.y);
71     }
72
73     return EXIT_SUCCESS;
74 }
```

Exercice 3 : Lecture d'un tableau

Écrire un programme qui initialise un tableau d'entiers en lisant des valeurs au clavier. On considèrera que les valeurs sont saisies les unes après les autres et que la saisie s'arrête sur une valeur négative. La valeur négative ne doit pas être conservée dans le tableau.

Solution :

```

1  /*****
2  *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3  *  Version : 1.2
4  *  Objectif :
5  *  Saisir un tableau en s'arrêtant sur une valeur négative.
6  *****/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 #define CAPACITE      10      /* capacité du tableau à lire */
12
13 int main()
14 {
15     int tab[CAPACITE]; /* la tableau à trier */
16     int taille;        /* la taille effective du tableau à trier */
17     int valeur;        /* valeur lue au clavier */
18
19     /* Saisir le tableau */
20     printf("Entrez une valeur négative pour arrêter.\n");
21     taille = 0;
22     scanf("%d", &valeur);
23     while (valeur >= 0 && taille < CAPACITE) {
24         /* ranger la valeur */
25         tab[taille] = valeur;
26         taille++;
27
28         /* Saisir une nouvelle valeur */
29         scanf("%d", &valeur);
30     }
31     if (valeur >= 0) {
32         printf("Capacité du tableau atteinte. Arrêt de la saisie\n");
33     }
34
35
36     /* Afficher le tableau */
37     printf("[");
38     if (taille > 0) { /*{ le tableau n'est pas vide }*/
39         int i; /* pour parcourir les indices du tableau */
40
41         /* afficher le premier élément */
42         printf("%d", tab[0]);
43
44         /* afficher les autres éléments */
45         for (i=1; i < taille; i++) {
46             printf(";%d", tab[i]);

```



```
47     }
48     }
49     printf("]");
50
51     return EXIT_SUCCESS;
52 }
```

Exercice 4 : Tri par insertion séquentielle

Soit $A[1..N]$ un vecteur de N entiers relatifs quelconques, l'objectif est de trier le vecteur A . Le vecteur A est trié si $A[1] \leq A[2] \leq \dots \leq A[N]$.

Le tri utilisé est le tri par insertion séquentielle. C'est un tri en $(N - 1)$ étapes. L'étape i consiste à placer le $(i + 1)^{\text{e}}$ élément du vecteur à sa place dans le sous-vecteur $A(1..i + 1)$ sachant que $A(1..i)$ a été trié par les étapes précédentes. Dans le cas du tri par insertion séquentielle, la recherche de la position d'insertion, se fait séquentiellement en comparant successivement l'élément à insérer aux i premiers éléments du vecteur.

Exemple : Voici les différentes valeurs du vecteur 8 2 9 5 1 7 après chaque étape (la partie encadrée correspond à la partie du vecteur déjà traitée et donc triée) :

vecteur initial	:	8 2 9 5 1 7
après l'étape 1	:	2 8 9 5 1 7
après l'étape 2	:	2 8 9 5 1 7
après l'étape 3	:	2 5 8 9 1 7
après l'étape 4	:	1 2 5 8 9 7
après l'étape 5	:	1 2 5 7 8 9

4.1 Écrire un programme qui lit une série de N ($N > 0$, lu au clavier) nombres relatifs, les range dans un vecteur A , les classe par ordre croissant en utilisant le tri par insertion séquentielle et, enfin, affiche la série ordonnée.

Solution :

1 **R0** : Trier un tableau A de taille effective N (les indices sont $1..N$).

On remarque qu'après chaque étape i du tri, les $(i+1)$ premiers éléments sont à leur place. Plus précisément, le sous-tableau compris entre les indices 1 et $i+1$ est trié. Ainsi, au bout de la $(N-1)^{\text{e}}$ étape l'ensemble du tableau est trié.

L'étape i consiste à trier le tableau $1..i+1$ en sachant que le tableau $1..i$ est déjà trié.

1 **R1** : **Raffinage De** << Trier un tableau A de taille effective N >>

2 | **Pour** indice := 2 **JusquÀ** indice = N **Faire**

3 | Insérer $A[\text{indice}]$ dans $A(1..\text{indice})$ sachant que $A(1..\text{indice})$ est trié

4 | **FinPour**

5

6 **R2** : **Raffinage De** << Insérer $A[\text{indice}]$ dans $A(1..\text{indice})$ sachant que $A(1..\text{indice})$ est trié >>

7 | Déterminer la position théorique de $A[\text{indice}]$ dans $A(1..\text{indice})$

8 | Décaler les éléments compris entre la position théorique et $\text{indice}-1$

9 | Ranger l'élément $A[\text{indice}]$

Si on essaie d'exécuter mentalement ce raffinement¹, on constate que l'on a perdu la valeur $A[\text{indice}]$ lorsque l'on veut la ranger. Elle a été écrasée lors du raffinement. Il est donc nécessaire de la conserver avant et donc de revoir notre raffinement.

1 **R2** : **Raffinage De** << Insérer $A[\text{indice}]$ dans $A(1..\text{indice})$ sachant que $A(1..\text{indice})$ est trié >>

2 | Conserver la valeur de $A[\text{indice}]$ memoire: **out Entier**

1. Ce qui doit toujours être fait, puisque ceci fait partie des choses à faire pour vérifier qu'un raffinement est correct.

```

3   | Déterminer la position pos de A[indice] dans A(1..indice)
4   | Décaler les éléments compris entre pos et indice-1
5   | Ranger l'élément mémorisé
6
7   R3 : Raffinage De << Déterminer la position théorique de A[indice] dans A(1..indice) >
8   | pos <- 1
9   | TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
10  |     pos <- pos + 1
11  | FinTQ
12
13  R3 : Raffinage De << Décaler les éléments compris entre la position théorique et indice
14  | Pour i <- indice - 1 Décrémenter Jusqu'À i = pos Faire
15  |     A[i+1] <- A[i]
16  | FinPour

```

On peut en déduire l'algorithme.

```

1   Constante
2     CAPACITÉ = 100
3   Type
4     Vecteur = Tableau [1..CAPACITÉ] De Entier
5
6   Variable
7     A: Vecteur -- le tableau à trier
8     N: Entier -- le nombre d'éléments de A
9     indice: Entier -- l'indice de l'élément à insérer
10    memoire: Entier -- mémoriser A[indice]
11    pos: Entier -- position de l'élément mémoire dans A(1..indice)
12    i: Integer; -- parcourir les éléments du tableau
13
14  Début
15    Pour indice := 2 Jusqu'À indice = N Faire
16      -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est trié
17      -- Conserver la valeur de A[indice]
18      memoire <- A[indice]
19
20      -- Déterminer la position pos de A[indice] dans A(1..indice)
21      pos <- 1
22      TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
23        pos <- pos + 1
24      FinTQ
25
26      -- Décaler les éléments compris entre pos et indice-1
27      Pour i <- indice - 1 Décrémenter Jusqu'À i = pos Faire
28        A[i+1] <- A[i]
29      FinPour
30
31      -- Ranger l'élément mémorisé
32      A[pos] <- memoire
33    FinPour
34  Fin

```

On peut en écrire une version légèrement optimisée en réalisant les décalages en même temps que l'on cherche la position d'insertion.

```

1  Variable
2      A: Vecteur -- le tableau à trier
3      N: Entier  -- le nombre d'éléments de A
4      indice: Entier -- l'indice de l'élément à insérer
5      mémoire: Entier -- mémoriser A[indice]
6      pos: Entier -- pour chercher la position d'insertion dans A(1..indice)
7
8  Début
9      Pour indice := 2 JusquÀ indice = N Faire
10         -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est trié
11         -- Conserver la valeur de A[indice]
12         mémoire <- A[indice]
13
14         -- Faire de la place pour A[indice] dans A(1..indice)
15         pos <- indice
16         TantQue (pos > 1 ) Et (memoire < A[pos-1]) Faire
17             A[pos] <- A[pos-1]
18             pos <- pos - 1
19         FinTQ
20
21         -- Ranger l'élément mémorisé
22         A[pos] <- mémoire
23     FinPour
24 Fin

1  /*****
2  * Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3  * Version  : 1.2
4  * Objectif : Trier un tableau par insertion séquentielle
5  *****/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define MAX    20    /* Capacité maximale du tableau */
11
12 int main()
13 {
14     int tab[MAX];    /* la tableau à trier */
15     int nb;          /* la taille effective du tableau à trier */
16     int valeur;      /* valeur lue au clavier */
17     int indice;      /* tab[0..indice] est trié */
18     int position;    /* position d'insertion de tab[indice] */
19     int i;           /* variable de boucle */
20     int memoire;     /* conserver la valeur de tab[indice] */
21
22     /* Saisir le tableau */
23     printf("Entrez_une_valeur_négative_pour_arrêter.\n");
24     nb = 0;
25     scanf("%d", &valeur);

```

```
26     while (valeur >= 0 && nb < MAX) {
27         /* ranger la valeur */
28         tab[nb] = valeur;
29         nb++;
30
31         /* Saisir une nouvelle valeur */
32         scanf("%d", &valeur);
33     }
34     if (valeur >= 0) {
35         printf("Capacité du tableau atteinte. Arrêt de la saisie\n");
36     }
37
38     /* Trier le tableau */
39     for (indice = 1; indice < nb; indice++) {
40         /* Insérer tab[indice] dans tab(0..indice-1) */
41
42         /* Conserver la valeur de tab[indice] */
43         memoire = tab[indice];
44
45         /* déterminer la position théorique de tab[indice] */
46         position = 0;
47         while (position < indice && memoire >= tab[position]) {
48             position++;
49         }
50
51         /* décaler les éléments compris entre position et indice-1 */
52         for (i = indice-1; i >= position; i--) {
53             tab[i+1] = tab[i];
54         }
55
56         /* ranger l'élément */
57         tab[position] = memoire;
58     }
59
60     /* Afficher le tableau */
61     printf("[");
62     if (nb > 0) { /*{ le tableau n'est pas vide }*/
63         int i; /* pour parcourir les indices du tableau */
64
65         /* afficher le premier élément */
66         printf("%i", tab[0]);
67
68         /* afficher les autres éléments */
69         for (i=1; i < nb; i++) {
70             printf(";%i", tab[i]);
71         }
72     }
73     printf("]");
74
75     return EXIT_SUCCESS;
76 }
```

4.2 En conservant le principe du tri par insertion, expliquer comment améliorer l'efficacité de cet algorithme.

Solution : Si on respecte le principe du tri par insertion, il faudra toujours réaliser les décalages. On ne peut donc gagner en performance qu'en optimisant la recherche (l'optimisation présentée précédemment n'est pas une optimisation réellement intéressante !). Comme on sait que le tableau $A[1..\text{indice}-1]$ est trié, on peut utiliser une recherche par dichotomie pour rechercher la position théorique de l'élément dans le tableau.