Les types utilisateurs : exercices résolus
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

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

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

Solution :

```
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.

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

On peut alors en déduire l'algorithme suivant :

```
R1 : Raffinage De « Compter le nombre d'occurrences des chiffres d'un nombre »
  | Initialiser les compteurs à 0
  | Répéter
  |   | Comptabiliser l'unité de nombre
  |   | Supprimer l'unité de nombre
  | JusquÀ nombre = 0


R2 : Raffinage De « Initialiser les compteurs à 0 »
  | Pour i ← 0 JusquÀ i = 9 Faire
  |     nb[i] ← 0
  | FinPour

R2 : Raffinage De « Comptabiliser l'unité de nombre »
  | chiffre = nombre Mod 10
```

```
    | nb[chiffre] ← nb[chiffre] + 1

  R2 : Raffinage De « Supprimer l'unité de nombre »
    | nombre ← nombre Div 10
```

En C.

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Compter le nb d'occurrences des chiffres d'un nombre
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int nombre;         /* entier saisi par l'utilisateur */
    int nb[10]; /* nb[i] : nb d'occurrences du chiffre i dans nombre */
    int i;      /* pour parcourir les chiffres */

    /* saisir le nombre */
    printf("Nombre : ");
    scanf("%d", &nombre);

    /* initialiser les compteurs */
    for (i = 0; i < 10; i++) {
        nb[i] = 0;
    }

    /* comptabiliser chaque chiffre de nombre */
    do {
        int chiffre;    /* chiffre unité de nombre */

        /* comptabiliser l'unité de nombre */
        chiffre = nombre % 10;
        nb[chiffre]++;

        /* supprimer l'unité de nombre */
        nombre = nombre / 10;

    } while (nombre > 0);

    /* afficher les occurrences */
    for (i = 0; i < 10; i++) {
        printf("nb de %d : %d\n", i, nb[i]);
    }

    return EXIT_SUCCESS;
}
```

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 c'est-à-dire son abscisse (x) et son ordonnée (y) et sa direction. Seulement quatre directions sont possibles : haut, bas, droite ou gauche.

Les robots de type 1 ont des possibilités réduites qui se limitent à pivoter de 90^o 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 obstacles.

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 appelées 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 regoupe ces deux informations.

```
Type
    Position =
        Enregistrement
            x: Entier   -- abscisse
            y: Entier   -- ordonnée
        FinEnregistrement

    Direction = (NORD, SUD, EST, OUEST)

    Robot1 =
        Enregistrement
            position: Position
            direction: Direction
        FinEnregistrement
```

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

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_LIGNES` et `NB_COLONNES`).

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 :

```
Constante
```

```
    NB_LIGNES = 20
    NB_COLONNES = 30

Type
    Case = (LIBRE, MUR)
        -- indique si la case est LIBRE et peut donc être occupé par le robot
        -- ou si elle contient un mur

    Environnement = Tableau [1..NB_LIGNES, 1..NB_COLONNES] De Case
```

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

```
R0 : Faire avancer le robot au maximum
                robot: in Robot1, environnement: in Environnement

tests :
    - pas de mur => la limite est atteinte
    - un mur => s'arrête devant le mur

R1 : Raffinage De « Faire avancer le robot au maximum »
    TantQue (limite environnement non atteinte) et (Pas de mur) Faire
        Faire avancer le robot d'une case (vers l'est)
    FinTQ

R2 : Raffinage De « limite environnement non atteinte »
    Résultat ← robot.position.x < NB_LIGNES

R2 : Raffinage De « Pas de mur »
    Résultat ← environnement[robot.position.x + 1, robot.position.y] <> MUR

R2 : Raffinage De « Faire avancer le robot d'une case »
    robot.position.x ← robot.position.x + 1
```

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

```
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Faire avancer un robot vers l'est au maximum
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define NB_LIGNES 20
#define NB_COLONNES 30
```

```c
struct Position {
    int x;      /* abscisse */
    int y;      /* ordonnée */
};

typedef struct Position Position;

enum Direction { NORD, SUD, EST, OUEST };

typedef enum Direction Direction;

struct Robot1 {
    Position position;
    Direction direction;
};

typedef struct Robot1 Robot1;

enum Case { LIBRE, MUR };

typedef enum Case Case;

typedef Case Environnement[NB_LIGNES][NB_COLONNES];

int main()
{
    Robot1 robot;
    Environnement environnement;

    /* Initialiser l'environnement */
    printf("Initialiser l'environnement...\n");
    {
        int l, c; /* parcourir les lignes et colonnes de l'environnement */
        for (l = 0; l < NB_LIGNES; l++) {
            for (c = 0; c < NB_COLONNES; c++) {
                environnement[l][c] = LIBRE;
            }
        }
        /* Mettre un mur */
        environnement[10][0] = MUR;
    }

    /* Initialiser le robot */
    printf("Initialiser le robot...\n");
    robot.direction = EST;
    robot.position.x = 0;
    robot.position.y = 0;

    /* Faire avancer le robot */
    while (robot.position.x < NB_LIGNES - 1     /* limite non atteinte */
```

```
            && environnement[robot.position.x+1][robot.position.y] != MUR)
                                                    /* pas de mur */
        {
            /* faire avancer le robot d'une case (vers l'est) */
            robot.position.x++;

            /* Afficher la position du robot */
            printf("Le robot est en (%d, %d)\n", robot.position.x, robot.position.y);
        }

        return EXIT_SUCCESS;
    }
```

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 :

En C.

```c
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif :
 *      Saisir un tableau en s'arrêtant sur une valeur négative.
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define CAPACITE        10      /* capacité du tableau à lire */

int main()
{
    int tab[CAPACITE];  /* la tableau à trier */
    int taille;         /* la taille effective du tableau à trier */
    int valeur;         /* valeur lue au clavier */

    /* Saisir le tableau */
    printf("Entrez une valeur négative pour arrêter.\n");
    taille = 0;
    scanf("%d", &valeur);
    while (valeur >= 0  && taille < CAPACITE) {
        /* ranger la valeur */
        tab[taille] = valeur;
        taille++;

        /* Saisir une nouvelle valeur */
        scanf("%d", &valeur);
    }
    if (valeur >= 0) {
        printf("Capactité du tableau atteinte.  Arrêt de la saisie\n");
    }


    /* Afficher le tableau */
    printf("[");
    if (taille > 0) {   /*{ le tableau n'est pas vide }*/
        int i;  /* pour parcourir les indices du tableau */

        /* afficher le premier élément */
        printf("%d", tab[0]);

        /* afficher les autres éléments */
```

```
            for (i=1; i < taille; i++) {
                printf("; %d", tab[i]);
            }
        }
        printf("]");

        return EXIT_SUCCESS;
    }
```

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 $A[1] \leq A[2] \leq \cdots \leq A[N]$.

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^{e} élément du vecteur à sa place dans le sous-vecteur $A(1..i)$ sachant que $A(1..i-1)$ 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 $(i-1)$ 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 liste 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 liste ordonnée.

Solution :

```
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 précisemment, le sous-tableau compris entre les indices 1 et i+1 est trié. Ainsi, au bout de la (N-1)^{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.

```
R1 : Raffinage De << Trier un tableau A de taille effective N >>
  | Pour indice := 2 JusquÀ indice = N Faire
  |    Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est trié
  | FinPour

R2 : Raffinage De << Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est
  | Déterminer la position théorique de A[indice] dans A(1..indice)
  | Décaler les éléments compris entre la position théorique et indice-1
  | Ranger l'élément A[indice]
```

Si on essaie d'exécuter mentalement ce raffinage[1], on constate que l'on a perdu la valeur A[indice] lorsque l'on veur la ranger. Elle a été écrasée lors du raffinage. Il est donc nécessaire de la conserver avant et donc de revoir notre raffinage.

```
R2 : Raffinage De << Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est
  | Conserver la valeur de A[indice]                    memoire: out Entier
```

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

```
  | Déterminer la position pos de A[indice] dans A(1..indice)
  | Décaler les éléments compris entre pos et indice-1
  | Ranger l'élément mémorisé

R3 : Raffinage De << Déterminer la position théorique de A[indice] dans A(1..indice) >>
  | pos ⟵ 1
  | TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
  |     pos ⟵ pos + 1
  | FinTQ

R3 : Raffinage De << Décaler les éléments compris entre la position théorique et indice
  | Pour i ⟵ indice - 1 Décrémenter JusquÀ i = pos Faire
  |   A[i+1] ⟵ A[i]
  | FinPour
```

On peut en déduire l'algorithme.

```
Constante
    CAPACITÉ = 100
Type
    Vecteur = Tableau [1..CAPACITÉ] De Entier

Variable
    A: Vecteur  -- le tableau à trier
    N: Entier   -- le nombre d'éléments de A
    indice: Entier      -- l'indice de l'élément à insérer
    mémoire: Entier     -- mémoriser A[indice]
    pos: Entier -- position de l'élément mémoire dans A(1..indice)
    i: Integer; -- parcourir les éléments du tableau

Début
    Pour indice := 2 JusquÀ indice = N Faire
        -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est trié
        --   Conserver la valeur de A[indice]
        mémoire ⟵ A[indice]

        --   Déterminer la position pos de A[indice] dans A(1..indice)
        pos ⟵ 1
        TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
            pos ⟵ pos + 1
        FinTQ

        --   Décaler les éléments compris entre pos  et indice-1
        Pour i ⟵ indice - 1 Décrémenter JusquÀ i = pos Faire
            A[i+1] ⟵ A[i]
        FinPour

        --   Ranger l'élément mémorisé
        A[pos] ⟵ mémoire
    FinPour
Fin -- trier
```

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

```
Variable
    A: Vecteur  -- le tableau à trier
    N: Entier   -- le nombre d'éléments de A
    indice: Entier      -- l'indice de l'élément à insérer
    mémoire: Entier     -- mémoriser A[indice]
    pos: Entier -- pour chercher la position d'insertion dans A(1..indice)

Début
    Pour indice := 2 JusquÀ indice = N Faire
        -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice) est trié
        --   Conserver la valeur de A[indice]
        mémoire ← A[indice]

        --   Faire de la place pour A[indice] dans A(1..indice)
        pos ← indice
        TantQue (pos > 1 ) Et (memoire < A[pos-1]) Faire
            A[pos] ← A[pos-1]
            pos ← pos - 1
        FinTQ

        --   Ranger l'élément mémorisé
        A[pos] ← mémoire
    FinPour
Fin -- trier
```

4.2 En conservant le principe du tri par insertion, 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..indice-1] est trié, on peut utiliser une recherche par dichotomie pour rechercher la position théorique de l'élément dans le tableau.

Le programme en C est le suivant :

```
/*****************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.1
 *  Objectif : Trier un tableau par insertion séquentielle
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define MAX     20      /* Capacité maximale du tableau */

int main()
{
    int tab[MAX];       /* la tableau à trier */
```

```
int nb;             /* la taille effective du tableau à trier */
int valeur;         /* valeur lue au clavier */
int indice;         /* tab[0..indice] est trié */
int position;       /* position d'insertion de tab[indice] */
int i;              /* variable de boucle */
int memoire;        /* conserver la valeur de tab[indice]   */

/* Saisir le tableau */
printf("Entrez une valeur négative pour arrêter.\n");
nb = 0;
scanf("%d", &valeur);
while (valeur >= 0  && nb < MAX) {
    /* ranger la valeur */
    tab[nb] = valeur;
    nb++;

    /* Saisir une nouvelle valeur */
    scanf("%d", &valeur);
}
if (valeur >= 0) {
    printf("Capactité du tableau atteinte.  Arrêt de la saisie\n");
}

/* Trier le tableau */
for (indice = 1; indice < nb; indice++) {
    /* Insérer tab[indice] dans tab(0..indice-1) */

    /* Conserver la valeur de tab[indice] */
    memoire = tab[indice];

    /* déterminer la position théorique de tab[indice] */
    position = 0;
    while (position < indice && memoire >= tab[position]) {
        position++;
    }

    /* décaler les éléments compris entre position et indice-1 */
    for (i = indice-1; i >= position; i--) {
        tab[i+1] = tab[i];
    }

    /* ranger l'élément */
    tab[position] = memoire;
}

/* Afficher le tableau */
printf("[");
if (nb > 0) {       /*{ le tableau n'est pas vide }*/
    int i;  /* pour parcourir les indices du tableau */

    /* afficher le premier élément */
```

```
            printf("%i", tab[0]);

            /* afficher les autres éléments */
            for (i=1; i < nb; i++) {
                printf("; %i", tab[i]);
            }
        }
        printf("]");

        return EXIT_SUCCESS;
    }
```