Chapitre 12

Révisions : Les huitres

Exercice 1 : Une structure pour les huitres.

On souhaite développer un logiciel de simulation de culture ostréicole (culture des huitres). Pour cela on a besoin d'un type structuré `huitre` comprenant les champs :

- `contaminee` qui décrit si l'huitre est malade ou non,
- `perles` qui décrit le nombre de perles contenues dans l'huitre,
- `poids` décrivant le poids en grammes de l'huitre (avec une précision au dixième de gramme).

Définir ce type en choisissant pour chacun des champs le type le plus adapté.
Correction :

```
typedef struct {
        bool contaminee ;
        int perles ;
        double poids ;
} huitre ;
```

Exercice 2 : Saisie controlée des huitres.

On souhaite développer une fonction de saisie contrôlée `saisie_ent_pos_st()` répondant aux spécifications suivantes :

(i) La fonction affiche d'abord un message passé en paramètre.

(ii) La fonction renvoie un entier saisi par un utilisateur et dont la saisie est contrôlées de sorte que l'entier soit strictement positif et strictement inférieur à une borne maximale passée en paramètre.

(iii) La fonction affiche le message d'invite suivant avant chaque saisie :
`Saisir un entier strictement positif inférieur à XXX=`
où `XXX` est remplacé par la valeur de la borne maximale passée en paramètre.

Pour développer cette fonction on procédera en 2 temps :

Question 2.1 : La fonction booléenne `est_strict_pos_max()` permet de tester la condition (ii). Elle renvoie `true` si l'entier qui lui est passé en paramètre est strictement supérieur à 0 et strictement inférieur à la borne qui lui est également passée en paramètre et `false` sinon.

Proposer deux versions de cette fonction. Dans la première version, la contrainte est de ne pas utiliser l'opérateur logique `&&`, et dans la seconde version la contrainte est de ne pas utiliser l'opérateur logique `||`.
Correction :

```
bool est_strict_pos_max( int val, int borne_sup)
{
        return !( val < 1 || val >= borne_sup );
}

bool est_strict_pos_max(int val, int borne_sup)
{
        return ( ( val > 0 && val <= borne_sup ) );
}
```

et toutes les autres combinaisons correctes... ◇

Question 2.2 : Proposer une définition de la fonction `saisie_ent_pos_st()` répondant aux spécifications (i), (ii) et (iii) qui utilise une boucle `do while` et la fonction `est_strict_pos_max()` définie à la question précédente.

Correction :

```
int saisie_ent_pos_st( int borne_sup, string mes)
{
    int res ;
    affichage ( mes, "\n") ;
    do {
        affichage ( "Saisir un entier strictement positif", borne_sup, "\n") ;
        saisie (res) ;
    } while( ! est_strict_pos_max(res, borne_sup) );
    return res ;
}
```

◇

Exercice 3 : Affichage d'une huitre.

Proposer une fonction `affiche_huitre()` qui affiche de façon formatée une huitre passée en paramètre. L'affichage prendra la forme suivante :

- `H[Saine, 10.3, 0]`, pour une huitre non contaminée d'un poids de 10.3 grammes et contenant aucune perle ;
- `H[Infec, 70.9, 3]`, pour une huitre contaminée d'un poids de 70.9 grammes et contenant 3 perles ;

Correction :

```
void affiche_huitre(huitre h)
{
    string contagion;
    if ( h.contaminee )
        contagion="Infec";
    else
        contagion="Saine";
    affichage ( "H[", contagion, ", ", h.poids, ", ", h.perles, "]") ;
}
```

◇

Exercice 4 : Croissance d'une huitre.

Proposer une fonction `croissance()` qui permet de simuler la croissance d'une huitre passée en paramètre. Le taux de croissance est également passé en paramètre, sous forme d'un pourcentage. Par exemple avec un taux de croissance de 10% le poids de l'huitre croit de 10%.

Correction :

```
void croissance( huitre & h, double taux )
{
    h.poids *= 1. + taux / 100 ;
}
```

Exercice 5 : Un programme principal.

Question 5.1 : Proposer un programme principal permettant de *déclarer* une variable `casier` correspondant à un tableau d'`huitre`. La capacité maximale du casier sera fixée par une *constante nommée*.

Correction :

```
#define CAPACITE_CASIER 50
int main()
{
    huitre  casier  [CAPACITE_CASIER] ;
}
```

Question 5.2 : Ajouter au programme principal une opération permettant d'initialiser le contenu du `casier` avec un nombre d'huitres demandé à l'utilisateur. Pour cela on utilisera la fonction `saisie_ent_pos_st()` définie dans l'Exercice 2. Le casier est initialisé avec des huitres non-contaminées, ne contenant pas de perle et dont le poids est aléatoire mais compris entre 3 et 5 grammes.

On rappelle que la fonction `rand()` renvoie un entier compris entre 0 et `RAND_MAX`.

Correction :

```
int h;
int nb_huitres = saisie_ent_pos_st( CAPACITE_CASIER, "Saisie du nombre d'huitre: ") ;
srand(time(NULL));
for (h = 0 ; h < nb_huitres ; h++ )
{
        casier [h].contaminee = false;
        casier [h].perles = 0 ;
        casier [h].poids = 3.0 + 2.0 * rand() / RAND_MAX ;
}
```

Question 5.3 : Ajouter au programme principal une opération permettant d'initialiser un tableau `tab_poids` contenant les poids des huitres d'un casier. La capacité maximale de ce tableau est la même que celle d'un casier. **Par la suite, on supposera que l'on dispose de ce tableau de poids.**

Correction :

```
double tab_poids [CAPACITE_CASIER] ;
int h;
for ( h = 0 ; h < nb_huitres ; h++ )
{
       tab_poids[h] = casier[h].poids ;
}
```

Exercice 6 : Simulation d'un parc à huitres.

L'ostréiculteur souhaite déterminer le nombre de semaines d'attente nécessaires avant de sortir

les huitres de leur milieu de culture pour les mettre en bourriches. Pour cela il va simuler de façon simplifiée la croissance des huitres en ne considérant dans un premier temps que le facteur de poids.

À partir d'ici, la notion de casier est abandonnée. Chaque huitre est représentée par un poids mémorisé dans un tableau (par exemple celui défini à la question précédente). *À partir d'ici, on ne manipule donc plus qu'un tableau de poids* `tab_poids` *et on connait le nombre* `n_eff` *de cases de ce tableau qui sont effectivement utilisées !.*

Les huitres croissent en une semaine selon les règles suivantes :

- Le poids d'une huitre h_i à une semaine `s` (noté $poids(h_i, s)$) dépend de son propre poids à la semaine `s-1` et du poids des huitres adjacentes (dans le tableau) à la semaine `s-1`
  - Si le poids de l'huitre h_i est égale à la moyenne des poids des huitres adjacentes à plus ou moins 10% alors le poids de l'huitre h_i augmente de 15%.
  - Si le poids de l'huitre h_i est supérieur à cette fourchette alors le poids de l'huitre h_i augmente 20 grammes,
  - Si le poids de l'huitre h_i est inférieur à cette fourchette alors le poids de l'huitre ne change pas.
  - Dans tous les cas une huitre ne peut dépasser le poids de 200 grammes.
- Pour les huitres des extrémités (celles du début et de la fin du tableau) la formule ne peut être la même car elles n'ont qu'un seul voisin. Dans ce cas, dans les règles précédentes, la moyenne des poids des 2 voisins est remplacée par le poids de l'unique voisin.

Question 6.1 : Définir une fonction `poids_suivant()` qui prend 2 paramètres : le poids d'une huitre et le poids moyen de ses deux voisines `p_vois` (ou le poids de son unique voisin) et renvoie le poids de l'huitre après une semaine de croissance.
Correction :

```
1  double poids_suivant( double p_h, double p_vois)
2  {
3          double res ;
4          if ( p_h <= 1.1*p_vois && p_h >= 0.9*p_vois )
5                  res = 1.15*p_h ;
6          else if (p_h > 1.1 * p_vois )
7                  res = 20.0 + p_h ;
8          else if (p_h < 0.9 * p_vois)
9                  res = p_h ;
10         if ( res > 200. )
11                 return 200. ;
12         else
13                 return res ;
14 }
```

Question 6.2 : Définir une fonction `cycle()` permettant d'appliquer un cycle de croissance d'une semaine à l'ensemble d'un tableau de poids passé en paramètre. Les poids du tableau sont modifiés par la fonction et les modification répercutées dans le programme appelant. On utilisera la fonction `poids_suivant()` et on fera attention aux cas des cases de début et de fin du tableau.

Attention : Lors de cette procédure de mise à jour des poids, on veillera à ne pas écraser le contenu d'une case du tableau tant qu'on en aura besoin. Par exemple, on ne peut pas modifier le poids de l'huitre h_i avant d'avoir calculé le nouveau poids de l'huitre h_{i+1}.
Correction :

```
1  void cycle( double tab_poids [], int n_eff )
2  {
```

```
        double buf [n_eff];
        int h ;
        double p_h ;
        double p_vois ;
        double prec ;
        for ( h = 0 ; h < n_eff ; h++ )
        {
                if ( h == 0 )
                        p_vois = tab_poids[h+1] ;
                else if (h == neff - 1)
                        p_vois = tab_poids[h-1] ;
                else
                        p_vois = ( tab_poids[h-1] + tab_poids[h+1] ) / 2;
                buf[h] = poids_suivant( p_h, p_vois) ;
        }
        for ( h = 0 ; h < n_eff ; h++ )
                tab_poids[h] = buf[h] ;
}
```

Alternative sans tableau tampon :

```
void cycle( double tab_poids [], int n_eff )
{
        double next, but;
        int h;
        for(h = 0; h < n_eff; h++ )
        {
                if ( h == 0 )
                        buf = poids_suivant(tab_poids[h], tab_poids[h+1]) ;
                else if ( h == n_eff - 1 )
                        buf = poids_suivant(tab_poids[h], tab_poids[h-1] ;
                else
                        buf = poids_suivant(tab_poids[h], (tab_poids[h-1] + tab_poids[h+1] ) /2 ;
                if ( h != 0)
                        poids[h-1] = next ;
                next = but ;
        }
}
```

Question 6.3 : Proposer un programme principal qui permet de calculer et d'afficher le nombre de semaines nécessaires pour que la somme des poids du huitres atteigne un poids total de 5 kilogrammes (ie le nombre de `cycle` à appliquer au tableau pour que la somme de ses éléments atteigne un total de 5 kilogrammes).

Correction :

```
int nb_cycles = 0 ;
int somme = 0;
int h;
for (h = 0 ; h < n_eff ; h++ )
                somme = tab_poids[h];
 while (somme < 5000.)
{
        somme=0;
        cycle(tab_poids,n_eff);
        nb_cycles ++ ;
        for ( h = 0 ; h < n_eff ; h++ )
                somme = tab_poids[h];
}
```

```
14 affichage ( "Nombre de cycles = ", somme) ;
```

Question 6.4 : Sur le marché les huitres se vendent en petites bourriches qui regroupent 20 huitres de même calibre. On considère qu'il existe 2 calibres :

Calibre 1 les huitres dont le poids est inférieur à 150 grammes.

Calibre 2 les huitres dont le poids est supérieur ou égal à 150 grammes.

Proposer une fonction `nombre_bourriches()` qui prend en paramètre un tableau de poids et son nombre de cases et qui renvoie au programme appelant le nombre de bourriches complètes de chaque calibre qu'il est possible de faire.

Correction :

```
1  void nombre_bourriches( double tab_poids [], int n_eff, int & cal_1, int & cal_2 )
2  {
3          cal_1 = 0;
4          cal_2 = 0;
5          int h;
6          for (h = 0; h < n_eff ; h++ )
7          {
8                  if ( tab_poids < 150. )
9                          cal_1 ++ ;
10                 else
11                         cal_2 ++ ;
12         }
13         cal_1 /= 20;
14         cal_2 /= 20;
15 }
```

Question 6.5 : Quelles sont les modifications à apporter à la fonction `nombre_bourriches()` pour qu'elle puisse selon les cas (selon le choix de celui qui l'utilise), alternativement renvoyer le nombre de bourriches complètes ou le nombre de bourriches total (complètes + commencées).

Correction :

Il faut prévoir un paramètre booléen permettant à l'utilisateur de décider ce qu'il veut : nombre de bourriches complètes ou nombre de total de bourriches, et ajouter une alternative permettant d'ajouter 1 au nombre de bourriches si ce restede la division n'est pas nul. ◇

Question 6.6 : Proposer un programme principal permettant d'afficher le nombre de bourriches complètes de chaque calibre en utiisant la fonction `nombre_bourriches()`.

Correction :

```
1  int calibre_1;
2  int calibre_2;
3
4  nombre_bourriches( tab_poids, n_eff, calibre_1, calibre_2);
5
6  affichage ( "Nombre de bourriches :\n");
7  affichage ( "Calibre 1 : ", calibre_1, "\n");
8  affichage ( "Calibre 2 : ", calibre_2, "\n");
```