Exercice 1 : Vecteurs

On s'intéresse à la définition en C d'un type, que l'on nommera vecteur, permettant la représentation de suites finies $(u_i)_{i\in[\![0,n-1]\!]}$. Un tel vecteur nécessite le stockage de deux grandeurs : sa *taille* : l'entier n dans la phrase précédente, et son *contenu* : les u_i pour $i \in [\![0, n-1]\!]$. Les u_i pour $i \in [\![0, n-1]\!]$ seront représentés au moyen d'un tableau de flottants dynamiquement alloué de taille n.

Q. 1 Définir une structure `struct vector_s` contenant un champs entier `taille` et un champs pointeur vers un flottant `contenu`, ce deuxième champs est en fait un pointeur vers la première case d'un tableau. On rappelle qu'un tableau n'est rien d'autre qu'un pointeur vers son premier élément.

Q. 2 Définir un type `vector` comme étant un pointeur vers une structure `struct vector_s`.

Q. 3 Représenter la mémoire obtenue après exécution des instructions suivantes :

```
vector v = (vector) malloc(sizeof(struct vector_s));
float* c = (float*) malloc(sizeof(float) * 5);
v->taille=5;
v->contenu=c;
v->contenu[0] = 0.;
v->contenu[1] = 2.;
v->contenu[2] = 4.;
```

Q. 4 Définir une fonction `vector new_vector(int taille);` prenant en argument un entier `taille` et retournant un vecteur de longueur `taille` nouvellement alloué. Les cases du tableau `contenu` nouvellement alloué devront être initialisées à 0.

Q. 5 Définir une fonction `void vector_print(vector v)` prenant en argument un vecteur et l'affichant. À titre d'exemple l'affichage du vecteur $(0, 1, 2, 3, 4)$ devra donner :

```
(0.00, 1.00, 2.00, 3.00, 4.00)
```

Q. 6 Définir une fonction `void vector_set(vector v, int i, float x)` permettant la modification de la `i`-ème case du vecteur `v` en y écrivant le nombre flottant `x`. Lorsque la fonction est appelée avec un entier `i` ne représentant pas une case valide du vecteur, on affichera un message d'erreur puis on arrêtera l'exécution du programme au moyen de `exit(1);`.

Q. 7 Définir une fonction `float vector_get(vector v, int i)` permettant la lecture de la `i`-ème case du vecteur `v`. Lorsque la fonction est appelée avec un entier `i` ne représentant pas une case valide du vecteur, on affichera un message d'erreur puis on arrêtera l'exécution du programme au moyen de `exit(1);`.

Q. 8 Représenter la mémoire aux points de programme ① et ② lors de l'exécution du programme suivant :

```
vector v1 = new_vector(3);
vector v2 = new_vector(2);
vector_set(v1, 0, 1.);
vector_set(v1, 1, 2.);
vector_set(v1, 2, 3.);
vector_set(v2, 0, 4.);
```

```
vector_set(v2, 1, 5.);
v2->contenu = v1->contenu; @\circled{1}@
vector_set(v2, 0, 6.);
float x = vector_get(v1, 0); @\circled{2}@
```

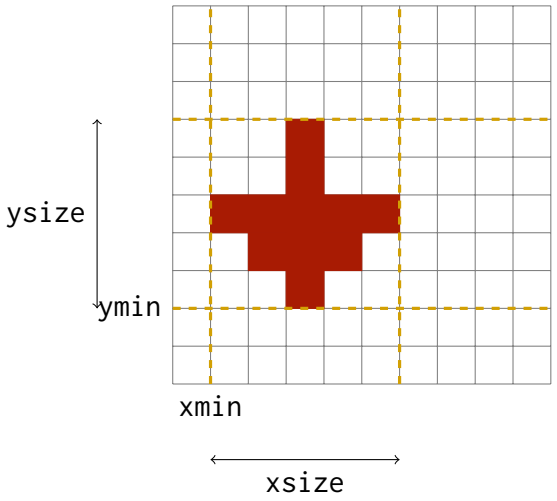
Q. 9 Définir une fonction `vector vector_add(vector v1, vector v2)` retournant la somme des vecteurs `v1` et `v2`. On appelle somme de deux vecteurs $(u_n)_{n\in[\![0;p-1]\!]}$ et $(v_n)_{n\in[\![0;p-1]\!]}$ de même longueur p le vecteur $(w_n)_{n\in[\![0;p-1]\!]}$ tel que $w_i = u_i + v_i$ pour $i \in [\![0; p-1]\!]$.

Q. 10 Définir une fonction `vector vector_concatene(vector v1, vector v2)` retournant la concaténation des vecteurs `v1` et `v2`. On appelle concaténation de deux vecteurs $(u_n)_{n\in[\![0;p_1-1]\!]}$ et $(v_n)_{n\in[\![0;p_2-1]\!]}$ le vecteur $(w_n)_{n\in[\![0;p_1+p_2-1]\!]}$ tel que $w_i = u_i$ pour $i \in [\![0; p_1-1]\!]$ et $w_{p_1+i} = v_i$ pour $i \in [\![0; p_2-1]\!]$.

Q. 11 Définir une fonction `float vector_pdt_scalaire(vector v1, vector v2)` retournant le produit scalaire des vecteurs `v1` et `v2`. On appelle produit scalaire de deux vecteurs $(u_n)_{n\in[\![0;p-1]\!]}$ et $(v_n)_{n\in[\![0;p-1]\!]}$ de même longueur p le réel $\sum_{i=0}^{p-1} u_i v_i$.

Exercice 2 : surfaces

Dans cet exercice on s'intéresse à la représentation de surfaces finies en 2D. Une surface finie en 2D est un sous ensemble de $\mathbb{Z}^2$. Une telle surface étant finie il est possible de définir les coordonnées de son point le plus en bas à gauche mais aussi de définir sa longueur suivant x et sa longueur suivant y. Le schéma ci-dessous illustre ces grandeurs sur l'exemple de la surface rouge. Tout point ne se trouvant pas dans le rectangle défini par les grandeurs `xmin`, `ymin`, `xsize`, `ysize` n'appartient nécessairement pas à la surface, pour les points se trouvant à l'intérieur nous utiliserons un tableau de tableaux de booléens (T dans l'exemple ci-dessous) de dimensions `xsize` $\times$ `ysize` indiquant dans la case $T[i][j]$ si le point de coordonnées $(j + \texttt{xmin}, i + \texttt{ymin})$ se trouve ou non dans la surface.



Q. 1 Donner la représentation de la surface exemple ci-dessus en machine.

Q. 2 Définir un type structuré `surface` permettant la représentation d'une surface.

Q. 3 Définir une fonction `void print_surface(surface s)` permettant l'affichage graphique d'une surface. À titre d'exemple pour la figure ci-dessus l'affichage attendu est :

```
coin bas gauche : 1, 2
  *
  *
*****
 ***
  *
```

Q. 4 Définir une fonction `surface rectangle(int xmin, int ymin, int sx, int sy)` retournant une surface représentant le rectangle de coin bas-gauche $(\texttt{xmin}, \texttt{ymin})$ de longueur suivant x `sx` et de longueur suivant y `sy`.

Q. 5 Donner une représentation de la mémoire après exécution de l'instruction `surface s1 = rectangle(2, 3, 2, 3);`

Q. 6 Définir une fonction `bool mem(int x, int y, surface s)` permettant de tester si un point de $\mathbb{Z}^2$ de coordonnées $(\texttt{x}, \texttt{y})$ est, ou non, dans la surface.

Q. 7 Définir une fonction `union_s` permettant de calculer la surface union de deux surfaces.

Q. 8 Définir une fonction `intersection_s` permettant de calculer la surface union de deux surfaces.

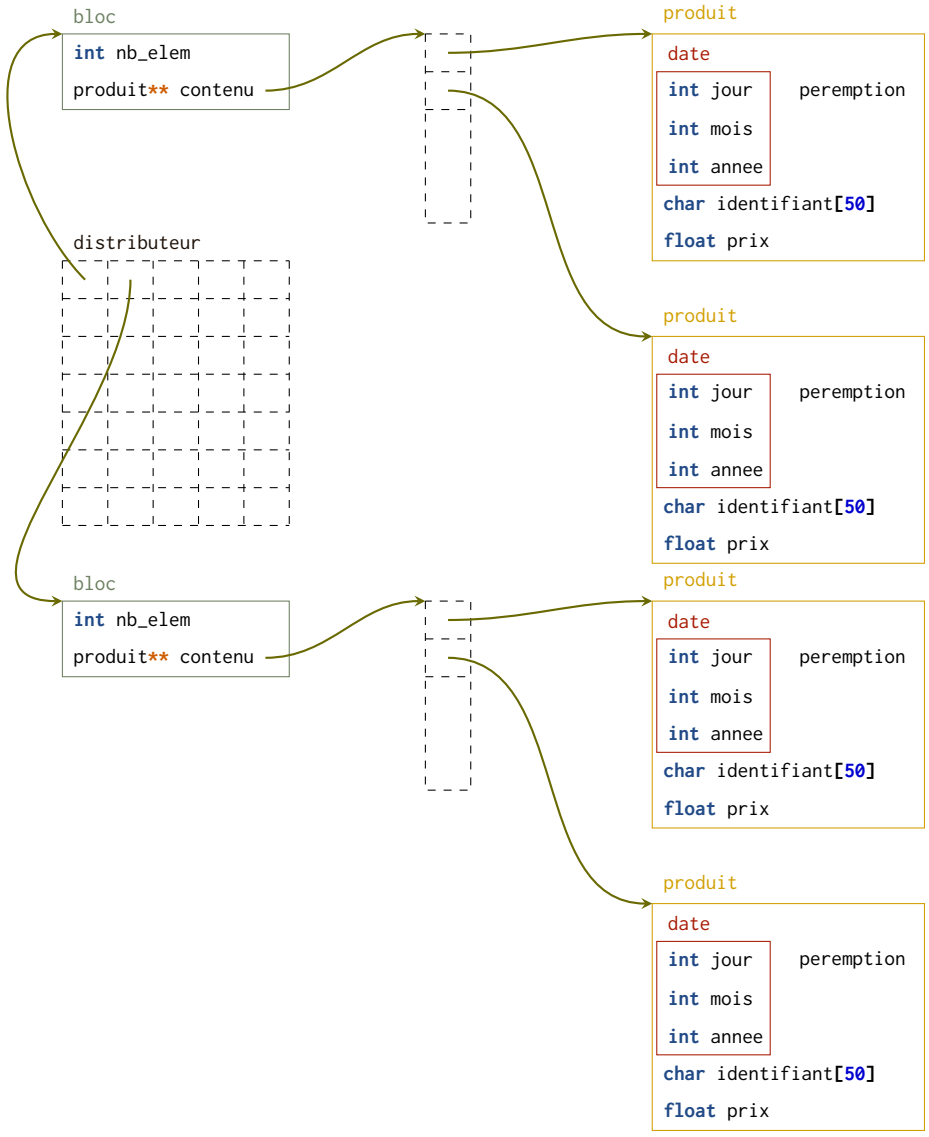
Q. 9 Définir une fonction `diff_s` permettant de calculer la différence $S_1 \setminus S_2$ de deux surfaces.

Exercice 3 : Distributeur

Dans cet exercice nous nous intéressons à la définition d'un type `distributeur` permettant de représenter le contenu d'un distributeur de nourriture/ café. Nous nous intéressons ensuite à la définition de fonctions permettant "l'entretien" du distributeur.

1. Définition des types

Le schéma ci-dessous représente les sous-composants d'un distributeur et la manière dont ceux-ci sont assemblés pour le représenter.



Un `distributeur` est une matrice (tableau à deux dimensions) de dimensions 7×5. Les cases de cette matrice contiennent un pointeur vers un `bloc`. Un `bloc` représente une des cases du distributeur (contenant plusieurs produits). Un `bloc` sera donc représenté en mémoire par un entier `nb_elem` indiquant le nombre d'éléments dans le bloc et par un pointeur `contenu` vers la première case d'un tableau. Un `bloc` contiendra toujours au plus 10 produits (le distributeur n'a pas une profondeur infinie). Ce tableau (qui sera de taille `nb_elem`) contiendra des pointeurs vers tous les produits stockés dans la case (le `bloc`) en question. Un `produit` sera représenté par une `date` (elle-même représentée par un jour, un mois, une année), par un `identifiant` (une chaîne de caractères de longueur au plus 50) et finalement par son prix, un flottant. Attention notre distributeur est donc un peu spécial, il est possible (pour l'instant) d'avoir dans la même case des produits n'ayant pas le même nom, ou pas le même prix. **ATTENTION** : Lorsqu'une case (i, j) (un `bloc`) du distributeur est vide, plutôt que d'allouer de la mémoire pour un `bloc` contenant 0 pour valeur `nb_elem`, on écrira le pointeur spécial `NULL` dans la case (i, j).

Q. 1 Définir les types `date`, `produit`, `bloc` et finalement `distributeur`.

Q. 2 Définir une fonction `produit* cree_produit(date peremption, char id[50], float prix)` retournant un pointeur vers un `produit` alloué dynamiquement et rempli avec les valeurs `peremption`, `id` et `prix`.

Q. 3 Définir une fonction `bloc* cree_bloc(int nb_elem)` retournant un pointeur vers un `bloc` alloué dynamiquement. Cette fonction devra initialiser le champs `nb_elem` à la valeur passée

en argument et le champs `contenu` à un pointeur vers une zone mémoire allouée dynamiquement pour l'occasion et pouvant contenir `nb_elem` pointeurs vers des `produits`.

Q. 4 Définir une fonction `void place_produit_dans_bloc(int pos, produit* pdt, bloc* bloc)` écrivant le produit `pdt` dans la case d'indice `pos` du bloc `bloc`.

2. Affichage du distributeur

Q. 5 Définir des fonctions :

- `void print_date(date date)`
- `void print_produit(produit* pst)`
- `void print_bloc(bloc* bloc, int i, int j)`
- `void print_distributeur(distributeur b)`

permettant l'affichage d'une `date`, d'un `produit`, d'un `bloc` et finalement d'un `distributeur`. Vous êtes complètement libre de choisir la manière dont les affichages sont faits. À titre d'exemple voici un extrait de ce qui est imprimé par mon programme.

```
…
====== 2 x 0 =======
  id: cafe
  peremption: 13/1/2029
  prix: 0.450000
====================
  id: cafe
  peremption: 16/1/2051
  prix: 0.450000
====================
====== 2 x 2 =======
  id: cafe
  peremption: 19/12/2034
  prix: 0.450000
====================
  id: sram
  peremption: 29/5/2031
  prix: 1.200000
====================
…
```

À ce stade vous pouvez récupérer sur moodle les deux fonctions `void init_distributeur_exemple(distributeur d)` et `void init_distributeur_petit_exemple(distributeur d)` permettant deux initialisations différentes d'un distributeur `d`. Vous pourrez utiliser les distributeurs ainsi générés pour faire des tests de vos fonctions. Vous trouverez aussi dans ce fichier la définition de deux constantes : `char liste_produits[6][50]` et `float prix_neuf[6]`, ces deux constantes donnent les noms (resp. le prix) des produits manipulés dans les exemples (libre à vous d'en ajouter).

Attention Dans la suite de cet exercice on ne précise plus chaque fonction auxiliaire devant être déclarée pour répondre à une question.

3. Lecture de l'état du distributeur

Q. 6 Définir une fonction `int nb_produit(distributeur d)` retournant le nombre de produits présents dans le distributeur.

Q. 7 Définir une fonction `float valeur_distributeur(distributeur d)` calculant la valeur totale du distributeur (la somme des valeurs) de tous les produits.

Q. 8 Définir un type `pdt_manquant` contenant deux entiers : un représentant le rang d'un produit (l'indice de l'identifiant de ce produit dans le tableau `liste_produits`) et l'autre représentant une quantité. Utiliser ce type pour définir une fonction `pdt_manquant trouve_produit_manquant(dis` `d, int nb_pdt)` retournant le rang d'un produit dont la quantité dans le distributeur est inférieure à `nb_pdt` et le nombre de tel produit à acheter pour atteindre une quantité de `nb_pdt`. Si aucun produit n'est manquant on retournera le rang -1.

Q. 9 Définir une fonction `produit** liste_produit(distributeur d)` prenant en argument un distributeur et retournant un tableau dynamiquement alloué de tous les produits contenus dans le distributeur. On fera attention à ne pas recopier tous les produits (qui existent déjà quelque part en mémoire) mais à seulement faire une copie du pointeur vers le produit.

4. Modification des blocs

Q. 10 Définir une fonction `void ajout_produit(int i, int j, produit* pdt, distributeur d)` permettant l'ajout du produit `pdt` dans la case de coordonnées $(\texttt{i}, \texttt{j})$ du distributeur (Indication : on créera un nouveau bloc d'une taille bien choisie).

Q. 11 Définir une fonction `void enleve_produit(int i, int j, int k, distributeur d)` permettant la suppression du `k`-ième produit de la case de coordonnées $(\texttt{i}, \texttt{j})$ (Indication : on créera un nouveau bloc d'une taille bien choisie).

5. Gestion des produits périmés

Q. 12 Définir une fonction `void reduction_produit_perime(distributeur d, date ajd)` appliquant une réduction de 50% sur tous les produits périmés dans le distributeur.

Q. 13 Définir un type `position` contenant 3 entiers `i`, `j` et `k` permettant de représenter la position d'un produit dans le distributeur. Les indices `i` et `j` représentent la case de la matrice distributeur et l'entier `k` l'indice du produit dans le tableau `contenu` du bloc en question. Utiliser ce type pour définir une fonction `position trouve_produit_perime(distributeur d, date ajd)` retournant la position d'un produit périmé du distributeur. Si aucun produit n'est périmé on retournera une position ayant un champs `i` à -1.

Q. 14 En déduire une fonction `void enleve_produit_perime(distributeur d, date ajd)` supprimant tous les produits périmés du distributeur.

6. Remplissage du distributeur

Dans un premier temps nous remplissons le distributeur sans nous soucier de rassembler les produits de même étiquette dans la même case.

Q. 15 Définir une fonction `produit* achete_produit(int pdt_id)` retournant un pointeur vers un produit nouvellement créé, ce produit aura l'identifiant (resp. le prix) stocké à l'indice `pdt_id` dans `liste_produits` (resp. `prix_neuf`). La date de péremption du produit sera tirée au hasard entre le `01/01/2022` et le `30/12/2050` (on supposera que tous les mois ont 30 jours).

Q. 16 Définir une fonction `void remplis_distributeur(distributeur d, int nb_pdt)` ajoutant des produits au distributeur afin que chaque produit soit présent `nb_pdt`. Chaque produit sera créé au moyen de la fonction `achete_produit` et ajouté dans la première case du distributeur ayant encore de la place (on rappelle que seulement 10 produits sont autorisés par case).

7. Rangement du distributeur

Q. 17 Définir une fonction `void range_distributeur(distributeur d)` assurant qu'une case du distributeur ne contient que des produits ayant le même identifiant. Vous êtes libre dans cette question de choisir votre méthode de rangement, on pourra par exemple :

- vider tout le distributeur et ranger les produits dans un tableau temporaire ;
- calculer pour chaque identifiant combien de produits ont cet identifiant ;
- en déduire le nombre N de cases qu'il est nécessaire de réserver pour les produits d'un tel identifiant ;
- choisir N cases libres dans le distributeur et y ranger les produits de l'identifiant en question.