Les Fonctions – L'allocation Dynamique de mémoire

(2 séances de TP)

Exercice 1

Rappel :

```
// En langage C, une fonction est un sous-programme qu'on appelle depuis la fonction main() ou depuis
// une autre fonction. Le langage fournit ses propres fonctions comme 'printf()' et il est possible
// d'en écrire d'autres afin de bâtir et de structurer un programme.
// Une fonctions se situe dans trois contextes différents :
//  - La déclaration, annonce l'existence de la fonction et comment on communique avec elle
//  - L'implémentation, fournit le code qui décrit le traitement à réaliser
//  - L'appel, est l'utilisation effective d'une fonction depuis une autre fonction
// Caractéristiques principales d'une fonction :
//  - Elle porte un nom sensible à la casse (ex : Myfunction() )
//  - Elle reçoit des arguments, quand on l'appelle, de manière conforme (en nombre et en type) à la liste
//    des paramètres qui ont été définis lors de sa déclaration.
//  - Une fonction renvoie une valeur d'un certain type
//  - Une fonction qui ne renvoie pas de valeur est déclarée avec le type 'void'
//  - Les variables déclarées dans la fonction sont dites 'locales' à la fonction et ne sont pas visibles
//    à l'extérieur de celle-ci.
//  - Les variables déclarées à l'extérieur de la fonction principale 'main()' sont dites 'globales'
//  - Les variables locales n'existent que le temps de l'exécution de la fonction, sauf si elles
//    sont déclarées 'static' et dans ce cas conservent leur valeur entre les différents appels.
//  - Les fonctions reçoivent leurs arguments soit par 'valeur' soit par 'adresse', seul le passage par
//    adresse permet de modifier la valeur, dans la fonction appelante, des variables passées en argument.
//
//      Exemple de fonction :
//      Déclaration : float Volume( float, float, float);
//      Implémentation :
//              int Volume( float Hauteur, float Largeur, float Profondeur) {
//                      float resultat = -1;
//                      if ((Hauteur > 0) && (Largeur > 0) && (Profondeur > 0)) {
//                      resultat = Hauteur * Largeur * Profondeur ;
//                      }
//
//                      return( resultat);
//              }
//      Appel :
//              vol = Volume( 10, 20, 5.5 );
```

- Créer une nouvelle solution nommée TP5
- Déclarer les prototypes des fonctions dans le fichier main.c au lieu de créer un fichier .h
- Les fonctions suivantes sont implémentées dans le fichier main.c, elles font appel aux variables globales que lorsque c'est nécessaire. Elles prévoient la réception d'arguments erronés et renvoient un code d'erreur (Par défaut 0, négatif si il y a une erreur, ou, positif pour un code de retour spécifique).
  Les fonctions sont documentées à l'aide de commentaires simples et pertinents.
- Le programme principal fait appel aux fonctions dans un scénario d'utilisation typique. Le test des fonctions se fait à l'aide de "jeux d'essais" afin de vérifier que la fonction fournit le comportement attendu. Dans certains cas, où lorsque la fonction est de type void, on ne teste pas le code de retour et on espère que le traitement est réalisable sans provoquer d'erreur. Certains langages "lèvent une exception" en cas d'erreur, celle-ci doit être interceptée et traitée. Ce n'est pas le cas du langage C qui rend plus complexe la gestion des erreurs en alourdissant le code.

1. **Fonction avec passage des arguments par valeur**
   Ecrire la Fonction (cf TP 2) :

   ```
   int valideTailleBagage( float dim1, float dim2, float dim3)
   ```

   Valeur de retour :
   0 : Bagage refusé
   1 : Bagage accepté

   La fonction ne s'occupe pas de la saisie clavier des dimensions ni de l'affichage du résultat, c'est le programme appelant qui s'en charge.
   Dim1, *Dim2* et *Dim3* sont des noms de paramètres formels utilisés par l'implémentation de la fonction, comme s'il s'agissait de variables locales. Lors de l'appel de la fonction, le programme appelant fournit des arguments sous la forme de ses propres variables (paramètres réels) dont les valeurs sont recopiées vers les paramètres formels.

   Ecrire la fonction **somme** qui calcule la somme des n premiers entiers naturels non nul en utilisant une boucle :

   ```
   unsigned short int somme(unsigned short int n)
   ```

   Valeur de retour :
   0 : dépassement de capacité ou argument non valide
   autre : résultat du calcul
   Tester la fonction pour n= 361 et n=362

2. **Fonction récursive**

   Ecrire une version récursive de la fonction somme(), c'est –à-dire une fonction qui fait appel à elle-même jusqu'à atteindre une condition d'arrêt

   ```
   unsigned short int sommeRecurs(unsigned short int n)
   ```

   Valeur de retour :
   0 : dépassement de capacité ou argument non valide

3. **Fonction avec passage des arguments par adresse**

   Lors d'un passage des paramètres par adresse, on fournit une adresse ou un pointeur vers la donnée afin de pouvoir en modifier la valeur et conserver l'effet des modifications après l'appel de la fonction. Du point de vue du langage C, le pointeur lui-même reste un paramètre passé par valeur, on ne peut le modifier. C'est la valeur qu'il pointe qui est modifiable.

   Ecrire une fonction qui échange la valeur de deux variables passées en arguments :

   ```
   void permuter(int *a, int *b)
   ```

   Le code de la fonction doit vérifier la présence de pointeur NULL.

   Ecrire une fonction qui affiche une structure HEURE (cf TP 4) passée par adresse :
   void AfficheHeure(HEURE * UneHeure);

Penser à utiliser la notation '->' à la place du '.' pour accéder aux champs d'une structure pointée.

4. Fonction avec passage d'un tableau en argument

Il est contraignant de passer un tableau par valeur. Parce que cela revient à recopier le tableau de valeurs sur la pile (Stack) avant l'appel de la fonction, avec les problèmes de performance et de consommation de mémoire que cela induit. Donc les langages utilisent un passage par adresse des tableaux, quitte à interdire l'écriture dans le tableau. En langage C, on passe simplement le nom du tableau, celui-ci étant un pointeur sur le premier élément. Les chaînes de caractères sont implémentées à l'aide de tableaux de caractères, on les manipule de la même manière, ce sont des pointeurs.

Déclarer dans le main() un tableau nommé MyTab1 de 10 valeurs de type int.
Ecrire une une fonction qui permet de remplir un tableau de 10 valeurs int avec des zéros.

```
int initTab1( int * tab)
```

Valeur de retour : 0 si tab est un pointeur NULL, la taille du tableau sinon.

Remarque : MyTab1 est un tableau de taille fixe.

Ecrire une fonction qui affiche le tableau sur une ligne de la console.

```
int afficheTab1(int *tab)
```

Plutôt que d'être déclaré avec une taille fixe au moment de la compilation, un tableau peut-être dimensionné de façon dynamique au moment de l'exécution du programme en demandant au système de nous fournir un pointeur sur un nouvel espace de mémoire réservé et de la taille souhaitée. Si le système accepte d'allouer la mémoire demandée, il faut aussi penser à restituer cette mémoire quand toutes les fonctions n'en ont plus besoin.

Exemple de déclaration et d'utilisation un tableau dynamique en trois étapes :

a. Définition de constantes utiles :
```
#define TAILLEINITIALE 100
#define TAILLEAJOUT 50
```
b. Déclaration du pointeur et des variables qui mémorisent la taille du tableau et le nombre de valeurs qui y sont rangées :
```
int *MyTab2 = NULL;
int TabSize = TAILLEINITIALE;
int NbElts = 0;
```
c. Allocation de la mémoire :
```
MyTab2 = (int *) malloc( TAILLEINITIALE * sizeof(int) );
if ( MyTab2 != NULL) { initTab2( MyTab2, TabSize); }
else { printf("mémoire insuffisante"); return(-1); }
```

Restitution au système de la mémoire **devenue inutile** (vérifier que ce soit le cas…) :

```
free(MyTab2);
```

Ecrire la fonction initTab2() qui remplit de zéros un tableau d'entiers, de taille Size, après avoir créé au préalable ce tableau MyTab2 **de manière dynamique** dans le programme principal:

```
void initTab2( int *tab, int Size) ;
```

Ecrire la fonction qui affiche les valeurs contenues dans le tableau :

```
void afficheTab2( int *tab, int NbElts );
```

5. Agrandissement d'un tableau alloué dynamiquement :

La fonction realloc() permet à partir d'un pointeur déjà alloué et d'une nouvelle taille de modifier une zone de mémoire utilisée par un tableau dynamique, donc d'agrandir le tableau sans perdre son contenu.

Nous allons écrire une fonction qui permet d'ajouter un élément à la suite dans le tableau. Si le tableau est rempli, il faut l'agrandir. Les opérations d'allocation sont coûteuses en temps CPU, il faut donc anticiper la demande et allouer suffisamment de place pour éviter de le faire à chaque nouvel ajout.

La fonction realloc() va fournir un nouveau pointeur sur une zone plus grande. La fonction d'ajout doit donc pouvoir modifier le pointeur qui correspond à la position du tableau. C'est pour cette raison que la fonction ne va pas prendre comme argument la valeur du pointeur mais plutôt son adresse. Ce qui se traduit par la syntaxe ****tab** dans la liste des paramètres de la fonction d'ajout, il s'agit d'un pointeur de pointeur !

Ecrire la fonction : (exercice facultatif mais conseillé, niveau expert)

Int ajoutElementDansTableau(int **tab, int *Size, int *NbElements, int Element);

Elle ajoute un nombre entier à la suite des valeurs déjà entrées et met à jour le nombre d'éléments stockés ainsi que la capacité réelle du tableau. Si le tableau est trop petit, il doit être agrandi de TAILLEAJOUT éléments.

Valeur de retour :

-1 : ajout impossible
>=0 : Nombre d'éléments dans le tableau

Exercice 2

Il s'agit d'améliorer la gestion des tableaux en créant une sorte d'objet tableau à l'aide d'une structure. Le tableau d'entiers est en fait un des champs de la structure TABLEAU et il est facile de le créer et d'en modifier la taille par des allocations dynamiques.

Créer un type structuré nommé TABLEAU pour manipuler les tableaux d'entiers :

```
typedef struct Tableau {
	int *elt; // le tableau d'entiers
	int size;  // la taille de ce tableau d'entiers
	int eltsCount; // le nombre d'éléments dans le tableau
} TABLEAU;
```

Ecrire les fonctions suivantes :

1 - crée un nouveau TABLEAU en allouant une taille initiale pour les données.

// renvoie une structure TABLEAU, avec un pointeur Elt = NULL si allocation a échoué.
TABLEAU NewArray( );

2 - modifie la taille du tableau

// renvoie 0 si erreur, ou la nouvelle taille si OK

int IncrementArraySize(TABLEAU * tab , int IncrementValue);

3 - Ecrit un élément à une position donnée (premier indice = 1) sans insertion

// Si l'élément n'est pas après le dernier élément, il faut agrandir le tableau et créer des éléments à
// zéros entre les deux.

// renvoie 0 si erreur, sinon position de l'élément inséré
int SetElement(TABLEAU * tab, int pos, int Element);

4 - Affiche une portion du tableau de l'indice début à l'indice fin

// renvoie 0 si erreur, sinon 1
int DisplayElements(TABLEAU *tab, int StartPos, int EndPos);

5 - Supprime des éléments avec compactage du tableau

// Met à jour le nombre d'éléments dans le tableau et diminue la taille du tableau

// renvoie 0 si erreur, sinon nouvelle taille du tableau
int DeleteElements(TABLEAU *tab, int StartPos, int EndPos);

6 - Déclarer un tableau de type TABLEAU dans le main et tester les fonctions