SÉANCE 7

Objectif

Le but de cette septième séance est de permettre à tous ceux qui n'ont pas terminé les séances précédentes de le faire et à tous ceux qui ont des difficultés de poser des questions à votre enseignant.

Ceux qui ont terminé les séances précédentes suivront les consignes de l'enseignant et trouveront ci-dessous des exercices supplémentaires.

Exercices

Exercice 1 (Triangle de Pascal)

L'objectif de cet exercice est d'afficher les n premières lignes du triangle de Pascal en réutilisant une partie de ce que vous avez fait lors de la séance 6 concernant les matrices dynamiques. C'est l'utilisateur qui saisira le nombre de lignes qu'il veut voir afficher. Par exemple, s'il demande 9 lignes, le programme devra afficher :

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5  1
1  6 15 20 15  6  1
1  7 21 35 35 21  7  1
1  8 28 56 70 56 28  8  1
```

1. Récupérez de la séance 6 :
   - le synonyme de type `tMatrice` ;
   - la fonction `MatAllouer` ;
   - la fonction `MatLiberer`.
2. Écrivez la fonction d'en-tête :
   `tMatrice Triangle(int n)`
   qui :
   - alloue une matrice de taille $n \times n$;
   - remplit la partie triangulaire inférieure de cette matrice avec le triangle de Pascal ;
     Si on note `A` cette matrice, alors :
     `A[`i`][`j`]` $=$ `A[`$i-1$`][`$j-1$`]`$+$`A[`$i-1$`][`j`]` pour $i = 1 \cdots n-1$ et $j = 1 \cdots i-1$;
     `A[`i`][0]` $=$ `A[`i`][`i`]` $= 1$ pour $i = 0 \cdots n-1$;
   - retourne la matrice.
3. Écrivez la fonction d'en-tête :
   `void AffTriangle(tMatrice A, int NbLig)`
   qui affiche à l'écran le triangle de Pascal de `NbLig` lignes stocké dans la partie triangulaire inférieure de la matrice `A` (voir l'exemple au début de l'exercice).
4. Écrivez une fonction principale qui :
   - demande à l'utilisateur et saisit le nombre de lignes ;
   - crée la matrice contenant le triangle de Pascal ;

- affiche le triangle de Pascal ;
- libère la matrice.

☞ Exercice 2 (Tri d'un tableau d'entiers)

Pour tester les fonctions ci-dessous, vous écrirez une fonction principale.

1. Écrivez la fonction d'en-tête :
   `void Trier(int Tab[], int NbElts)`
   qui trie dans l'ordre croissant les `NbElts` valeurs de type `int` stockées dans le tableau `Tab`. Vous pouvez utiliser l'algorithme du tri à bulles qui consiste à :
   Pour $i =$ `NbElts`$-1 \cdots 1$
   Pour $j = 0 \cdots i-1$
   Si `Tab`$[j+1] <$ `Tab`$[j]$
   Permuter `Tab`$[j+1]$ et `Tab`$[j]$
2. Écrivez la fonction d'en-tête :
   `void Aff(int Tab[], int NbElts)`
   qui affiche les éléments du tableau passé en paramètre.
3. Écrivez la fonction d'en-tête :
   `void TrierPtr(int Tab[], int NbElts, int *Ptr[])`
   qui ne modifie pas les valeurs du tableau `Tab`, mais qui remplit le tableau `Ptr` avec les adresses des éléments du tableau `Tab` dans l'ordre croissant de ces éléments. Par exemple, si on déclare :
   `int Tableau[3]={3,1,2};`
   alors, après la séquence suivante :
   ```
   int *TableauP[3];
   TrierPtr(Tableau,3,TableauP);
   ```
   `TableauP[0]` contiendra l'adresse de `Tableau[1]`, `TableauP[1]` contiendra l'adresse de `Tableau[2]` et `TableauP[2]` contiendra l'adresse de `Tableau[0]`.
4. Écrivez la fonction d'en-tête :
   `void AffPtr(int *TabP[], int NbElts)`
   qui affiche les valeurs de type `int` pointées par chacun des `NbElts` éléments de `TabP`. Avec l'exemple précédent, l'appel :
   `AffPtr(TableauP,3);`
   devra produire l'affichage de :
   `1 2 3`
5. Écrivez deux nouvelles versions de la fonction `TrierPtr` dans lesquelles l'espace mémoire du tableau des pointeurs n'est pas réservé par l'appelant, mais est alloué de manière dynamique par la fonction `TrierPtr`. Dans la première de ces deux versions, le tableau des pointeurs sera retourné par la fonction. Dans la seconde version, le tableau devra être délivré en sortie de la fonction. Trouver les en-têtes de ces deux versions fait partie de l'exercice.