TD 4 : Pointeurs et structures

Semaine du 19 février 2007

1 Chaînes de caractères

Exercice 1 Écrire une fonction

```
char *recherche(char *s, char c)
```

qui renvoie un pointeur vers la première occurrence du caractère c passé en argument. Si ce caractère n'apparaît pas dans la chaîne, la fonction devra renvoyer NULL.

Exercice 2 À l'aide de la fonction précédente, écrire une fonction

```
int compte(char *s, char c)
```

qui renvoie le nombre d'occurrences de c dans s.

2 Polynômes

On représente toujours un polynôme par un tableau de **double**. Mais on regroupe le pointeur vers les coefficients et le degré du polynôme dans une structure (*Rappel* : un polynôme de degré d a $d+1$ coefficients) :

```
struct polynome { int degre; double *coefficients; };
```

Exercice 3 Écrire des fonctions

```
struct polynome *somme_polynome(struct polynome *P, struct polynome *Q)
```

et

```
struct polynome *produit_polynome(struct polynome *P, struct polynome *Q)
```

calculant la somme et le produit de leurs arguments.

3 Matrices

Un tableau à deux dimensions peut être représenté par un tableau de tableaux. Par exemple, pour des tableaux de **double**, on utilisera le type **double** **, c'est à dire un pointeur vers un tableau dont chaque case est de type **double** *, c'est à dire contient un pointeur vers un tableau de **double**.

On peut regrouper le nombre de lignes, le nombre de colonnes et un pointeur vers des pointeurs (qui pointent chacun vers les coefficients d'une ligne de la matrice) dans une structure représentant une matrice :

```
struct matrice { int lignes, colonnes; double **coefficients; };
```

Exercice 4 Écrire une fonction

```
struct matrice *alloue_matrice(int lignes, int colonnes)
```

qui alloue une matrice lignes × colonnes. On devra utiliser la fonction malloc() pour allouer une **struct** matrice, puis un tableau de **double** * avec autant de cases que la matrice a de lignes, puis remplir chacune de ces cases avec un pointeur vers un tableau de **double** (qui contient autant de **double** que la matrice a de colonnes) que l'on allouera également avec malloc().

Exercice 5 Écrire une fonction

```
struct matrice *produit_matrice(struct matrice *A, struct matrice *B)
```

qui calcule le produit de deux matrices. Elle devra renvoyer NULL si les dimensions des matrices ne sont pas compatibles.

Exercice 6 Écrire une fonction

```
void libere_matrice(struct matrice *A)
```

qui désalloue la matrice A. On appelera la fonction free () sur le tableau contenant les pointeurs vers les lignes, sur la structure elle-même et sur les lignes de la matrice dans un ordre que vous devrez déterminer.

4 Pile

Rappel : Une pile est une structure de donnée dans laquelle les derniers éléments ajoutés sont les premiers à être récupérés.

Une première méthode pour implémenter une pile d'entiers consiste à stocker tous les éléments de la pile dans un simple tableau (le sommet de la pile se trouve dans la dernière case de ce tableau). Lorsqu'on veut ajouter (empiler) un élément, on recopie tous les éléments (et le nouvel élément) dans un tableau plus grand d'une case (de même, lorsqu'on veut enlever (dépiler) le sommet, on recopie tous les éléments sauf le sommet dans un tableau plus petit d'une case). On utilise la structure suivante :

```
struct pile_simple { int taille; int *elements; };
```

Exercice 7 Écrire des fonctions

```
void empile_pile_simple(struct pile_simple *pile, int n)
```

et

```
int depile_pile_simple(struct pile_simple *pile)
```

(On supposera que depile_pile_simple() n'est jamais appelée alors que la pile est vide).

Cette implémentation est simple mais inefficace puisque les éléments de la pile sont recopiés à chaque opération. Il est plus astucieux de garder un tableau partiellement rempli : lorsque la pile déborde, plutôt que d'allouer un tableau plus grand d'une case, on alloue un tableau deux fois plus grand et lorsque l'on dépile un élément, on ne recopie dans un tableau plus petit que si le tableau est aux trois quarts vide, auquel cas on divise sa taille par deux.

On utilise donc la structure suivante :

```
struct pile_amortie { int taille, capacite; int *elements; };
```

où capacite représente le nombre de cases du tableau elements, tandis que taille représente le nombre de cases effectivement remplies. À partir de la taille ^{ème} case, il y a des cases non utilisées.

Exercice 8 Écrire des fonctions

```
void empile_pile_amortie(struct pile_amortie *pile, int n)
```

et

```
int depile_pile_amortie(struct pile_amortie *pile)
```