

Examen, INFO505 – Programmation C II, Session 1

Durée : 1h30

Documents autorisés : tous documents du cours/td/tp, notes manuscrites (nb : pas de livres)

Les exercices sont indépendants. Le barème est indicatif, et dépasse volontairement 20. La durée est de 2h pour les étudiants bénéficiant d'un 1/3 temps.

1 Connaissance du C et pile d'exécution (/7)

On vous donne le programme C suivant :

```
#include <stdlib.h>
#include <stdio.h>
char* f( char* d, char* s );
char* mystere( char* b, int n )
{
    // (1) état début de la fonction
    int k = f( b, b ) - b;
    char* p = (char*) malloc( k * n + 1 );
    char* q = p;
    // (2) état avant la boucle
    for ( ; n != 0; n-- )
        q = f( q, b );
    *q = 0;
    // (3) état en fin de boucle
    return p;
}

char* f( char* d, char* s )
{
    while ( *s != 0 ) *d++ = *s++;
    return d;
}

int main()
{
    char p[] = "Yop";
    char* q = mystere( p, 4 );
    // (4) état après calcul
    printf( "%s\n", q );
    free( q );
    return 0;
}
```

(sp) n	4						
b		•					
(fp) @ret. mystere							
q		?					
p[0]		'Y'	←				
p[1]		'o'					
p[2]		'p'					
p[3]		0					

1. (/1) Que fait la fonction `f` ?
2. (/2) Qu'affiche le programme et plus généralement que fait la fonction `mystere` ?
3. (/3) On affiche l'état de la pile d'exécution à l'étape (1). Afficher la pile d'exécution (et le tas d'exécution) de ce programme aux lignes (2), (3) et (4), comme précisé dans le programme.
4. (/1) Ecrivez la ligne `int k = f(b, b) - b;` autrement pour faire le même calcul.

2 Mot dans un texte (/5)

Ecrivez une fonction `char* recherche_mot(char* texte, char* mot)` qui recherche si un mot `mot` donné est présent dans le texte `texte` donné, et retourne son adresse de départ dans le texte si trouvé ou `NULL` sinon. On considère que toute lettre différente de 'a'-'z' ou 'A'-'Z' est un séparateur de mot. De plus le mot cherché peut être partiellement en majuscule. Ainsi

```
char* texte= "C'est un bon chien : tu es mon YouKI ! Tu vas avoir ton nonos.";
char* mot = "youki";
char* p = recherche_mot( texte, mot );
```

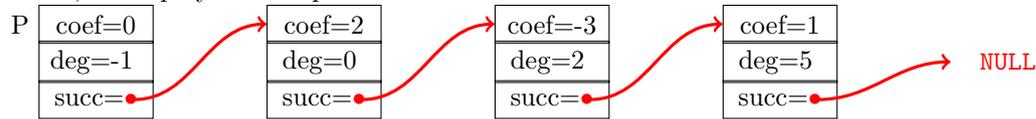
met `texte+31` dans `p`.

NB : Faites une fonction auxiliaire qui transforme un caractère en sa version majuscule si possible ou retourne le caractère 0 pour tous les autres (i.e. séparateur).

3 Polynômes (/12)

On va représenter les polynômes, e.g. $P(x) = x^5 - 3x^2 + 2$, sous forme de liste simplement chaînée de monômes triés en partant de celui de degré le plus petit, donc sur l'exemple précédent : $2, -3x^2, x^5$.

Chaque monôme contiendra deux valeurs (le coefficient et le degré) et un pointeur vers le monôme de degré supérieur ou NULL si c'était déjà le monôme de degré maximal. Sur l'exemple précédent, notre polynôme P peut être modélisé ainsi :



On en déduit les types C suivants :

```

typedef struct SMonome {
    double coef;
    int deg;
    struct SMonome* succ;
} Monome;
typedef Monome Polynome;
  
```

Tout polynôme commence par un monôme fictif (degré -1), ce qui permet d'éviter les cas particuliers d'insertion ou de suppression dans une liste. Par convention, le polynôme $P(x) = 0$ est représenté par le seul monôme fictif $\{0.0, -1, \text{NULL}\}$. La fonction d'initialisation s'écrit donc :

```
void P_init( Polynome* P ) { P->coef = 0.0; P->deg=-1; P->succ=NULL; }
```

- (/1) Ecrire un petit programme qui crée les variables représentant le polynôme $P(x) = x^5 - 3x^2 + 2$ sans allocation dynamique.

- (/1) On va écrire ci-dessous le code des fonctions manipulant un polynôme :

```

void P_affiche( Polynome* P ); // affiche le polynôme P
double P_eval ( Polynome* P, double x ); // évalue P en x
Monome* P_cherche( Polynome* P, int d ); // cherche le monôme avant x^d
void P_ajout ( Polynome* P, double c, int d ); // ajoute le monôme c*x^d
void P_termine( Polynome* P ); // remet le polynôme à P(x)=0
  
```

Pourquoi passe-t-on les polynômes par adresse ? Aurait-on pu passer dans certaines fonctions les polynômes par valeur ? Si oui, listez ces fonctions.

- (/1,5) Ecrire la fonction qui affiche un polynôme (de manière simple).

```
void P_affiche ( Polynome* P );
```

Sur le polynôme précédent, cela afficherait $(2*x^0)+(-3*x^2)+(1*x^5)$.

- (/1,5) On veut maintenant créer les polynômes en leur ajoutant (dynamiquement) des monômes. On veut donc écrire une fonction qui rajoute un monôme à un polynôme existant. Ce n'est malheureusement pas si simple car ajouter un monôme peut parfois enlever un monôme de la liste ! (Ex : On rajoute le monôme $3x^2$ à $P(x)$ au-dessus.)

On va donc d'abord écrire une fonction qui cherche le monôme qui *précède* un monôme d'un certain degré d dans P . Il peut retourner le monôme fictif si le premier monôme réel a un degré supérieur ou égal à d , où si le polynôme est 0.

```
Monome* P_cherche( Polynome* P, int d );
```

Par exemple `P_cherche(P, 2)` retourne l'adresse du monôme $2x^0$ sur l'exemple au-dessus.

- (/3) Ecrire maintenant la fonction qui rajoute un monôme à un polynôme existant, en utilisant la fonction précédente.

```
void P_ajout( Polynome* P, double c, int d );
```

NB : Cette fonction peut créer un monôme, supprimer un monôme ou modifier un monôme.

- (/2) On veut maintenant évaluer le polynôme en un point x donné. Faites-le de la manière la plus efficace possible selon vous.

```
double P_eval( Polynome* P, double x );
```

NB : vous pouvez utiliser la fonction `double pow(double x, int n)` qui calcule x^n, \dots ou faire autrement.

- (/2) Ecrire enfin la fonction qui réinitialise le polynôme au polynôme 0.

```
void P_termine( Polynome* P );
```