Université Savoie Mont Blanc, UFR SceM
2022-2023

INFO505, L3 INFO, Session 1
15 novembre 2022

Corrigé 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. La durée est de 2h pour les étudiants bénéficiant d'un 1/3 temps.

1 Mot dans un texte (/5)

Ecrivez une fonction `char* recherche_mot( char* texte, char* mot )` qui recherche si un `mot` donné est présent dans le `texte` donné, et retourne son adresse 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 qui transforme un caractère en sa version majuscule si possible ou retourne le caractère 0 pour tous les autres (i.e. séparateur).

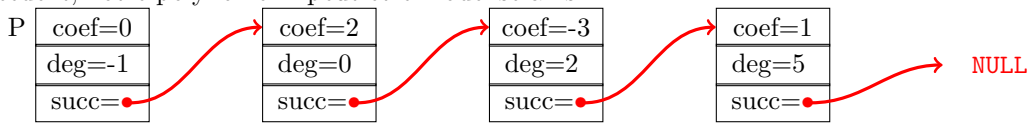
```
#include <stdio.h>

char simplifie( char c )
{
  if ( c >= 'A' && c <= 'Z' )      return c;
  else if ( c >= 'a' && c <= 'z' ) return c - 'a' + 'A';
  else return 0;
}
int verifie( char* s, char* mot )
{
  for ( ; ( *s != 0 ) && ( *mot != 0 ); s++, mot++ )
    {
      char c1 = simplifie( *s );
      char c2 = simplifie( *mot );
      if ( c1 != c2 ) return 0;
      if ( c2 == 0  ) return 1;
    }
  return 1;
}
char* recherche_mot( char* texte, char* mot )
{
  char last = 0;
  for ( char* p = texte; *p != 0; ++p )
    {
      char c = simplifie( *p );
      if ( c != 0 && last == 0 && verifie( p, mot ) )
        return p; // un mot ne commence qu'aprÃ¨s un sÃ©parateur
      last = c;
    }
  return NULL;
}
int main()
{
  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 );
  if ( p != NULL ) printf("OK: Found at position %ld\n", p - texte );
  else printf( "ERROR: NOT FOUND\n" );
  char* q    = recherche_mot( texte, "nOnOs" );
  if ( q != NULL ) printf("OK: Found at position %ld\n", q - texte );
  else printf( "ERROR: NOT FOUND\n" );
  char* r    = recherche_mot( texte, "chat" );
```

```
  if ( r != NULL ) printf("ERROR: Found at position %ld\n", r - texte );
  else printf( "OK: NOT FOUND\n" );
}
```

2 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 un liste. Par convention, le polynôme $P(x) = 0$ est représenté par le monôme `{0.0, -1, NULL}`. La fonction d'initialisation s'écrit donc :

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

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

```
Monome   m5 = { 1, 5, NULL };
Monome   m2 = {-3, 2, &m5   };
Monome   m0 = { 2, 0, &m2   };
Polynome P  = { 0,-1, &m0   };
```

2. (/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.

> On passe les structures par adresse pour limiter les copies et pour permettre les modifications. Ici on aurait pu passer par valeur `P_affiche`, `P_eval`, `P_cherche`, avec un léger surcoût (20 octets au lieu de 8 octets).

3. (/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)`.

```
void     P_affiche( Polynome* P )
{
  Monome* M = P->succ;
```

```
  if ( M == NULL ) printf( "0" );
  while ( M != NULL )
    {
      printf( "(%.f*x^%d)", M->coef, M->deg );
      M = M->succ;
      if ( M != NULL ) printf( "+" );
    }
}
```

4. (/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 l'enlever 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 dégré supérieur ou égal à d.

   ```
   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.

```
Monome* P_cherche( Polynome* P, int d )
{
  Monome* cur = P;
  Monome* next= P->succ;
  while ( next != NULL && next->deg < d )
  {
    cur  = next;
    next = next->succ;
  }
  return cur;
}
```

5. (/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 : Notez que P est passé par pointeur pour pouvoir être modifié le cas échéant. Cette fonction peut créer un monôme, supprimer un monôme ou modifier un monôme.

```
  if ( c == 0.0 ) return;
  Monome* pred = P_cherche( P, d );
  if ( pred->succ == NULL || pred->succ->deg != d ) // on crée un monôme
    {
      Monome* M  = (Monome*) malloc( sizeof(Monome) );
      M->coef    = c;
      M->deg     = d;
      M->succ    = pred->succ;
      pred->succ = M;
    }
  else
    {
      pred->succ->coef += c;
      if ( pred->succ->coef == 0 )
        {
          Monome* M  = pred->succ;
          pred->succ = M->succ;
          free( M );
        }
    }
```

6. (/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 utilisez la fonction `double pow( double x, int n)` qui calcule x^n, ... ou faire autrement.

```
double  P_eval   ( Polynome* P, double x )
{
  int      n = 0;
```

```
  double  r = 0.0;
  double xn = 1.0;
  Monome* M = P->succ;
  while ( M != NULL )
    {
      while ( n < M->deg ) { xn *= x; n++; }
      r += M->coef * xn;
      M = M->succ;
    }
  return r;
}
```

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

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

```
void     P_termine ( Polynome* P )
{
  Monome* M = P->succ;
  while ( M != NULL )
    {
      Monome* S = M;
      M         = M->succ;
      free( S );
    }
  P->succ = NULL;
}
```