Licence Informatique
Université Bordeaux I

Initiation à la Programmation
Session de janvier 2000

L'épreuve dure 3 heures.
Les programmes sont à rédiger en langage C - ansi.
On respectera (un minimum) le mode d'indentation utilisé en cours.
On utilisera systématiquement la macro-fonction `assert()` *pour signaler toute erreur de programmation ou anomalie d'utilisation.*
La notation tiendra compte de la correction des programmes mais aussi de la qualité du codage (lisibilité, maintenabilité et extensibilité).

Exercice 1. Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

static int i;

static int
f(int *k)
{
  static int n = 1;
  static int p = 1;

  n = n + p * 2 + 1;
  p = p + 1;

  *k = (n / p) + 2;
  return n;
}

int
main ()
{
  for (i = 2; i < 8; i++)
    {
      printf ("pour %d on a ", i);
      printf ("%d\n", f (&i));
    }
  return EXIT_SUCCESS;
}
```

Exercice 2. La commande unix `wc` affiche le nombre de lignes, de mots et de caractères présents sur l'entrée standard ; les options `-l` ,`-w` et `-c` permettent d'afficher respectivement le nombre de lignes, de mots et de caractères. Implémenter la commande `wc` et ses options.

Indications

Un mot est une séquence de caractères de longueur maximale ne comportant pas d'espace au sens de la fonction `isspace`.

```
# wc
123456789[ctrl-d]       0       1       9
# wc
123456789[retour chariot]
```

```
[ctrl-d] 1       1      10
# wc -c -w
bizarre bizarre !
Comme c'est etrange !
     7      40
# wc -w -c
bizarre bizarre !
Comme c'est etrange !
     7      40
```

Exercice 3. On s'intéresse aux arbres binaires de recherche[1] (a.b.r.) contenant des objets de type `double`. On utilisera les déclarations suivantes :

```
typedef struct noeud *noeud ;
typedef struct arbre *arbre ;

struct noeud
{
  double valeur ;
  noeud g;
  noeud d;
};

struct arbre
{
  noeud racine ;
};
```

1. Implémenter une version itérative de la fonction `double abr_max(arbre a)` qui retourne l'élément maximal de l'a.b.r. `a` ;
2. implémenter la fonction `double abr_moyenne(arbre a)` qui retourne la valeur moyenne des éléments de l'a.b.r. `a` ;
3. implémenter la fonction `double abr_k_ieme(arbre a, unsigned long k)` qui retourne la valeur du k-ième élément de l'a.b.r. `a`.

Exercice 4. Il s'agit de proposer un module traitant des polynômes, ce module proposera les fonctionnalités suivantes :

- créer un polynôme nul ;
- détruire un polynôme ;
- évaluer un polynôme en un double donné ;
- modifier un polynôme en le multipliant par un monôme ;
- modifier un polynôme en lui ajoutant un monôme ;
- faire la somme de deux polynômes ;
- faire le produit de deux polynômes ;
- dupliquer un polynôme.

Afin d'implémenter des polynômes via des *tableaux dynamiques* on utilise les types suivants:

```
struct monome
{
  double coefficient;
  unsigned long degre;
};

struct polynome
{
  unsigned long taille;
  struct monome *les_monomes;
};
```

1. Rappel : pour chaque sommet d'un a.b.r. les valeurs des sommets du sous-arbre gauche sont inférieures à la valeur du sommet considéré qui est elle même inférieure à toutes les valeurs des sommets du sous-arbre droit.

On dira qu'un objet de type `polynome` est *normalisé* si son champ taille correspond au nombre de ses monômes non nuls et si ses monômes sont ordonnés de façon strictement croissante suivant leur degré. Exemples :

– tout polynôme nul sera représenté par `{0,NULL}` ;
– le polynôme $-3x^4 + 5x$ sera représenté par un objet `{2,ADR}` où `ADR` désigne l'adresse d'un objet contenant `{{5 , 1}, {-3 , 4}}` ;
– l'objet `{3,ADR}` où `ADR` désigne l'adresse de `{{5 , 1}, {0 , 2}, {-3 , 4} }` représente aussi le polynôme $-3x^4 + 5x$ mais cette représentation n'est pas normalisée.

Toutes les fonctions du module polynôme ne travailleront, en entrée comme en sortie, qu'avec des objets de type `polynome` normalisés.

1. Proposer un fichier en-tête pour le module polynôme.
2. En supposant que le programme `mon_polynome.c` (contenant une fonction main) fasse appel au module `polynome`, écrire un fichier makefile.
3. Écrire les fonctions créer et détruire.
4. Écrire la fonction qui modifie un polynôme en lui ajoutant un monôme.
5. Écrire la fonction somme de deux polynômes. Afin de conserver une complexité raisonnable, cette fonction fera un seul appel à malloc (pour réserver dès le départ un espace suffisant au résultat) et un seul appel à realloc (pour normaliser le résultat).
6. Écrire la fonction produit de deux polynômes, évaluer sa complexité.

Annexes

Les définitions des fonctions suivantes sont extraites du «Linux Programmer's Manual».

```
int isspace (int c);                                         <ctype.h>
checks for white-space characters.  In the "C" and "POSIX" locales,
these are: space, form-feed ('\f'), newline ('\n'), carriage return
('\r'), horizontal tab ('\t'), and vertical tab ('\v').

int getchar(void);                                             <stdio.h>
reads the next character from stdin and returns it as an unsigned char
cast to an int, or EOF on end of file or error.

int strcmp(const char *s1, const char *s2);                  <string.h>
The strcmp() function compares the two strings s1 and s2.  It returns
an integer less than, equal to, or greater than zero if s1 is found,
respectively, to be less than, to match, or be greater than s2.

void *malloc(size_t size);                                   <stdlib.h>
malloc() allocates size bytes and returns a pointer to the allocated
memory.  The memory is not cleared.

void free(void *ptr);                                         <stdlib.h>
free() frees the memory space pointed to by ptr, which must have been
returned by a previous call to malloc(), calloc() or realloc().
Otherwise, or if free(ptr) has already been called before, undefined
behaviour occurs.  If ptr is NULL, no operation is performed.

void *realloc(void *ptr, size_t size);                         <stdlib.h>
realloc()  changes the size  of the memory  block pointed to by ptr to
size bytes.  The contents will be unchanged to the minimum of the old
and new sizes; newly allocated mem- ory will be uninitialized.  If ptr
is NULL, the call is equivalent to malloc(size); if size is equal to
zero, the call is equivalent to free(ptr).  Unless ptr is NULL, it
must have been returned by an earlier call to malloc(), calloc() or
realloc().
```

```
Exemple de fichier Makefile

CC = gcc
CFLAGS = -g -Wall
OBJ= test_proc.o proc.o gqueue.o

all:	test_proc

test_proc:	$(OBJ)
	$(CC) $(CFLAGS) -o test_proc $(OBJ)

test_proc.o: test_proc.c proc.h

proc.o: proc.c proc.h gqueue.h

gqueue.o: gqueue.c gqueue.h
```

Corrigé

Barème : 3 - 3 - 6 (2 + 2 + 2) - 8 (1 + 1 + 1 + 2 + 2 + 1)

- 8 points étaient reservés aux exercices de base (questions 1 - 3.1 - 4.1, 4.2 et 4.3)
- 6 points aux questions d'un niveau intermédiaire (questions 2 - 3.2 et 4.6)
- 6 points aux questions plus difficiles (questions 3.3 - 4.4 et 4.5)

Exercice 1

```
pour 2 on a 4
pour 5 on a 9
pour 6 on a 16
pour 7 on a 25
```

Erreurs courantes : ne pas tenir compte soit du passage par adresse du paramètre (&k), soit du caractère statique des variables (n et p) de la fonction f.

Exercice 2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#define CAR_CHAINE "-c"
#define MOT_CHAINE "-w"
#define LIGNE_CHAINE "-l"

void compte(long *caracteres, long *mots, long *lignes) ;

void usage(char *argv0) ;

int
main(int argc, char *argv[])
{
  long ligne, caractere, mot ;
  int opt_ligne, opt_mot, opt_car ;

  if (argc == 1)
    opt_car = opt_mot = opt_ligne = 1 ;
  else
    {
      int i ;
      opt_car = opt_mot = opt_ligne = 0 ;

      for(i = 1 ; i < argc ; i++)
        if (!strcmp(argv[i], CAR_CHAINE))
          opt_car = 1 ;
        else if (!strcmp(argv[i], MOT_CHAINE))
          opt_mot = 1 ;
        else if (!strcmp(argv[i], LIGNE_CHAINE))
          opt_ligne = 1;
        else usage(argv[0]);

    }

  compte(&caractere, &mot, &ligne);

  if (opt_ligne)
    printf("\t%ld", ligne);
  if (opt_mot)
    printf("\t%ld", mot);
  if (opt_car)
    printf("\t%ld", caractere);
  return EXIT_SUCCESS ;
}
```

```
void
usage(char *argv0)
{
  assert(argv0);
  fprintf(stderr,"usage : %s [-l][-w][-c]", argv0);
  exit(EXIT_FAILURE);
}

void
compte(long *caracteres, long *mots, long *lignes)
{
  int car_courant ;
  short dans_un_mot = 0 ;

  *caracteres = *mots = *lignes = 0 ;
  while((car_courant = getchar()) != EOF)
    {
      (*caracteres)++ ;
      if (!isspace(car_courant))
        {
          dans_un_mot = 1 ;
          continue ;
        }
      if (dans_un_mot) /* terminaison d'un mot */
        (*mots)++ ;
      if (car_courant == '\n')
        (*lignes)++ ;
      dans_un_mot = 0 ;
    }
  *mots += dans_un_mot ;
}
```

Erreurs courantes : confusion entre les mots sur l'entrée standard et ceux formant les arguments de l'appel de la commande ; utilisation d'un tableau de taille indeterminée ; comparaison directe de chaines de caractères.

Commentaires sur la correction : la reconnaissance des options n'est pas vraiment extensible ; avec du temps, on aurait pu utiliser la structure suivante :

```
typedef struct
{
  char *nom_option ;
  int validee ;
} option ;

option  les_options[] = {{"-c",0}, {"-m",0},{"-l",0}} ;

typedef enum {CARACTERE = 0 , MOT , LIGNE} code_option ;
```

Exercice 3

```
#include <assert.h>
#include <stdlib.h>

#define ARBRE_NON_VIDE(a) assert(a != NULL && a->racine != NULL)

double
abr_max(arbre a)
{
  noeud n;

  ARBRE_NON_VIDE(a);
  for (n = a->racine; n->d != NULL; n = n->d)
    ;
  return n->valeur;
}

static void somme_et_compte(noeud n, double *somme, long *nb_noeuds);

double
abr_moyenne(arbre a)
```

```
{
  double somme = 0;
  long nb_noeuds = 0;
  ARBRE_NON_VIDE(a);
  somme_et_compte(a->racine, &somme, &nb_noeuds);
  return somme / nb_noeuds;
}

static void
somme_et_compte(noeud n, double *somme, long *nb_noeuds)
{
  *somme += n->valeur;
  (*nb_noeuds)++;
  if (n->d != NULL)
    somme_et_compte(n->d, somme, nb_noeuds);
  if (n->g != NULL)
    somme_et_compte(n->d, somme, nb_noeuds);
}


double k_ieme(noeud n, long *k); /* correction - merci olivier */

double
abr_k_ieme(arbre a, long k)
{
  double val;
  ARBRE_NON_VIDE(a);
  assert(k > 0);
  val = k_ieme(a->racine, &k);
  assert(k <= 0);               /*l'arbre contient moins de k noeuds */
  return val;
}

double
k_ieme(noeud n, long *k)
{
  double val;

  if (n->g != NULL)
    val = k_ieme(n->g, k);
  (*k)--;
  if (k < 0)
    return val;
  if (k == 0)
    return n->valeur;
  if (n->d != NULL)
    return k_ieme(n->d, k);
  return 0;
}
```

Erreurs courantes : confusion entre le type arbre et le type noeud [si a désigne une variable de type arbre l'expression `a->d` n'est pas définie] ; déréférencement du pointeur null (aller un cran trop loin) ; utilisation d'une variable statique pour un accumuler un résultat sur un arbre (la fonction ne peut servir qu'une seule fois) ; utilisation d'un tableau de taille indeterminée (Q.3) ; sortie d'une fonction de type non void sans retour de résultat (Q.3).

Exercice 4

```
/* module traitant de polynomes a coefficient double et de degre unsigned long */
#ifndef POLYNOME
#define POLYNOME

typedef struct polynome *polynome;


/* creation / destruction */
extern polynome pol_creer_nul();
extern void pol_detruire(polynome *p);
extern polynome pol_dupliquer(polynome p);
```

```
/* operations standards */

extern double pol_evaluer(polynome p, double x) ;

extern void pol_multiplier(polynome p, double coefficient, unsigned long degre);
/* multiplie p par le monome "coefficient * x ^ degre" */

extern void pol_ajouter(polynome p, double coefficient, unsigned long degre);
/* ajoute a p le monome "coefficient * x ^ degre" */

extern polynome pol_somme(polynome p, polynome q) ;
/* p et q restent inchanges */

extern polynome pol_produit(polynome p, polynome q) ;
/* p et q restent inchanges */

#endif /* POLYNOME */
```

Makefile :

```
CC = gcc
CFLAGS = -g -Wall
OBJ = mon_polynome.o polynome.o

mon_polynome : $(OBJ)
	$(CC)  $(CFLAGS) -o mon_polynome $(OBJ)

mon_polynome.o : mon_polynome.c polynome.h

polynome.o : polynome.c  polynome.h
```

polynome.c :

```
#define RETAILLER(p) (struct monome *)\
  realloc(p->les_monomes, p->taille * sizeof(struct monome))
#define COEF(p,i) p->les_monomes[i].coefficient
#define DEGRE(p,i) p->les_monomes[i].degre

polynome
pol_creer_nul()
{
  polynome p = (polynome) malloc(sizeof(struct polynome));
  assert(p != NULL);
  *p = (struct polynome) {0, NULL};
  return p;
}

void
pol_detruire(polynome * p)
{
  assert(p != NULL && *p != NULL);
  if ((*p)->taille != 0)
    free((*p)->les_monomes);
  free(*p);
  *p = NULL;
}

static long
position(unsigned long degre, polynome p)
{
  long i;
  for (i = 0; i < p->taille; i++)
    if (DEGRE(p, i) >= degre)
      return i;
  return p->taille;
}

static void
etendre(polynome p, long indice)
{
  long i;
  p->taille++;
```

```
  RETAILLER(p);
  for (i = p->taille - 1; i > indice; i--)
    p->les_monomes[i] = p->les_monomes[i - 1];
}

static void
comprimer(polynome p, long indice)
{
  long i;
  for (i = indice; i < p->taille - 1; i++)
    p->les_monomes[i] = p->les_monomes[i + 1];
  p->taille--;
  RETAILLER(p);
}

void
pol_ajouter(polynome p, double coefficient, unsigned long degre)
{
/* Quatre cas sont  consideres :
 *   - le coef est nul : rien a faire
 *   - p est le polynome nul : allouer un tableau de taille 1
 *   - p possede un monome de meme de degre :
 *     * soit les coef. s'annulent, il faut alors comprimer
 *     * soit les 2 monomes s'ajoutent
 *   - p ne possede pas de monone de meme degre :
 *      insertion du nouveau monome
 */
  long pos;

  assert(p != NULL);
  if (coefficient == 0)
    return;
  if (p->taille == 0)
    {
      *p = (struct polynome){1, malloc(sizeof(struct monome))};
      assert(p->les_monomes != NULL);
      *(p->les_monomes) = (struct monome){coefficient, degre};
      return;
    }

  pos = position(coefficient, p);

  if (pos < p->taille && p->les_monomes[pos].degre == degre)
    {
      if (COEF(p, pos) == -coefficient)
        comprimer(p, pos);
      else
        COEF(p, pos) += coefficient;
      return;
    }

  etendre(p, pos);
  p->les_monomes[pos] = (struct monome){coefficient, degre};
}

polynome
pol_somme(polynome p, polynome q)
{
  /* il s'agit d'un algorithme de fusion particulier :
   * deux monomes peuvent s'ajouter ou s'annuler.
   */

  assert(p != NULL && q != NULL);

  if (p->taille == 0)
    return pol_dupliquer(q);
  if (q->taille == 0)
    return pol_dupliquer(p);

  {
    int i_p, j_q;
```

```
    polynome res = pol_creer_nul();
    res->les_monomes = (struct monome *)
      malloc((p->taille + q->taille) * sizeof(struct monome));
    assert(res->les_monomes != NULL);

    for (i_p = j_q = 0; i_p < p->taille && j_q < q->taille;)
      {
        if (DEGRE(p, i_p) == DEGRE(q, j_q))
          {
            if (COEF(p, i_p) != -COEF(q, j_q))
              COEF(res, res->taille++) = COEF(p, i_p++) + COEF(q, j_q++);
            else
              i_p++, j_q++;
          }
        if (DEGRE(p, i_p) > DEGRE(q, j_q))
          COEF(res, res->taille++) = COEF(p, i_p++);

        if (DEGRE(p, i_p) < DEGRE(q, j_q))
          COEF(res, res->taille++) = COEF(q, j_q++);
      }

    while (i_p < p->taille)
      COEF(res, res->taille++) = COEF(p, i_p++);
    while (j_q < q->taille)
      COEF(res, res->taille++) = COEF(q, j_q++);
    RETAILLER(res);
    return res;
  }
}

polynome
pol_produit(polynome p, polynome q)
{
  polynome res, tmp, res_precedent ;
  long i;

  assert(p != NULL && q != NULL);
  if (p->taille == 0 || q->taille == 0)
    return pol_creer_nul();

  for (res = pol_dupliquer(p), i = 0; i < q->taille; i++)
    {
      tmp = pol_dupliquer(p);
      pol_multiplier(tmp, COEF(q, i), DEGRE(q, i));
      res_precedent = res ;
      res = pol_somme(tmp, res);
      pol_detruire(&res_precedent);
      pol_detruire(&tmp);
    }
  return res;
}

/* complexite en ecriture et a la louche
 *  q->taille * (p->taille + p->taille + res->taille)
 *  comme res->taille est éborn par p->taille * q->taille
 *  on a q->taille * (p->taille + p->taille * q->taille)
 *  soit O( p->taille * q->taille ^2) ecritures
 *  De plus on fait 2 * q->taille malloc (evitables).
 */
```

Erreurs courantes (NC = erreur non comptabilisée) :

1. oublier le ifndef ; donner l'implémentation des types (la liste des champs composant structure) ; masquer l'implémentation sans utiliser de pointeurs ; omettre la liste des fonctions ;
2. omettre la compilation de polynome.o ; ajouter un progamme testpoly.c non demandé ;
3. ne pas libérer les_monomes ; libérer, sans vérification, les_monomes (NC) ;
4. confondre tableau dynamique et liste chainée ; ne pas rechercher si un monôme de même degré existe déjà ; ne pas considérer l'ajout du premier monôme (NC) ; insérer des monômes nuls (NC) ;

5. utiliser la fonction `ajouter_monome`; insérer des monômes nuls (NC); ne pas considérer le cas de paramètre nul (NC) ;
6. oublier de libérer la mémoire de travail.