Licence Informatique
Université Bordeaux I

Initiation à la Programmation
Session de Septembre 2002

L'épreuve dure 3 heures et aucun document n'est autorisé.
Les programmes sont exclusivement à rédiger en langage C - ansi. On respectera le mode d'indentation utilisé en cours. On utilisera systématiquement la macro-fonction `assert()` *pour signaler toute anomalie d'utilisation. La notation tiendra compte de la correction des programmes (en particulier de la bonne utilisation de la mémoire dynamique) mais surtout de la qualité du codage : lisibilité, maintenabilité, extensibilité et concision.*
Barème indicatif : 3 4 5 8.

Exercice 1. Écrire le texte que produirait l'exécution du programme suivant :

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

int
f (int a, int *b)
{
  static int c = 0;
  c = c + a + 1;
  a = *b + c;
  *b = c;
  return a + 1;
}

int
main (void)
{
  int a[2], *c = a + 1;
  for (a[0] = a[1] = 1; a[1] < 7; a[1]++)
    {
      printf ("%d␣%d␣", a[0], a[1]);
      printf ("%d␣", f (a[0], a));
      printf ("%d\n", f (c[0], c));
    }
  return EXIT_SUCCESS;
}
```

Exercice 2. Écrire le programme correspondant à la commande

`occurence -<entier non nul> <liste de paramètres>`

qui affiche les paramètres apparaissant exactement un certain nombre de fois, pour exemples :

```
> occurence -1 les poules du couvent couvent
les poules du
> occurence -2 les poules du couvent couvent
couvent
> occurence -3 les poules du couvent couvent
> occurence les poules du couvent couvent
occurence: nombre d'occurences absent
```

Exercice 3. On considère des listes simplement chaînées d'entiers, écrire

1. la déclaration du type `lce` correspondant aux listes simplement chaînées d'entiers ;
2. la fonction `lce implante_12345()` qui retourne une liste chaînée contenant les entiers 1,2,3,4,5 ;
3. la fonction `int est_croissante(lce liste)` qui vérifie si la liste passée en paramètre est croissante ou non.
4. la fonction `void renverse(lce *liste)` qui renverse la liste passée en paramètre.

Problème Il s'agit de présenter un module générique traitant des ensembles d'objets, c'est à dire des ensembles de pointeurs.

Les fonctionnalités du module ensemble sont : la création et la destruction d'un ensemble ; l'ajout, la suppression et la recherche d'un élément dans un ensemble ; l'application d'une fonction paramétrée à tous les éléments d'un ensemble ; l'union, la différence et l'intersection de deux ensembles et le calcul de la cardinalité d'un ensemble.

1. En appliquant le principe du masquage de l'implémentation définir l'interface du module ensemble.
2. En supposant que le programme `voitures.c` fasse appel au module ensemble écrire son fichier makefile.
3. Écrire les déclarations de type du fichier source du module ensemble.
4. Écrire le code des fonctions de création et de destruction d'un ensemble.
5. Écrire le code des fonctions suppression (d'un élément), union et intersection.
6. Une application utilisant ce module a le comportement cyclique suivant : elle ajoute beaucoup d'éléments aux ensembles puis travaille intensivement avec ceux-ci sans ne plus utiliser la fonction d'ajout. Indiquer une implémentation efficace du module pour ce type d'application et discuter de son efficacité en la comparant aux solutions classiques (liste chaînée, tableau, arbre, table de hachage).

Annexes

```
void assert (int expression);                                      <assert.h>
int strcmp(const char *s1, const char *s2);                        <string.h>
void *malloc(size_t size);                                         <stdlib.h>
void free(void *ptr);                                              <stdlib.h>
void *realloc(void *ptr, size_t size);                             <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);       <stdlib.h>
```

— Fin —

Correction

Exercice 1.

Remarques :
- 2 lignes suffisent,
- le correcteur n'a lu aucun commentaire.

```
1 1 4 6
2 5 10 19
```

Exercice 2.

Remarques :
- on préférera usage() à assert() pour signaler les erreurs dues à l'utilisateur ;
- la fonction d'usage termine le programme à l'aide de la fonction exit() ... elle ne doit pas retourner.
- à peine un étudiant sur dix a produit un algorithme correct (...),
- l'utilisation de pointeur de pointeur de la fonction `strtol` n'est pas maîtrisée (le test de débordement n'était pas demandé).

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

void
usage (char *argv0)
{
  fprintf (stderr, "%s␣:␣erreur␣d'utilisation", argv0);
  fprintf (stderr, "%s␣-<entier␣non␣nul>␣<liste␣de␣èparamtres>", argv0);
  exit (EXIT_FAILURE);
}

int
main (int argc, char *argv[])
{
  long nb;
  char *endptr;
  size_t i;

  if (argc == 1)
    usage (argv[0]);

  nb = -strtol (argv[1], &endptr, 10);
  if (nb < 1 || *endptr != '\0' || errno == ERANGE)
    usage (argv[0]);

  // on remplace par NULL les occurences multiples afin de ne plus en
  // tenir compte par la suite

  for (i = 2; i < argc; i++)
    if (argv[i] != NULL)
      {
        size_t j, nb_occ;
        for (j = i + 1, nb_occ = 1; j < argc; j++)
          if (!strcmp (argv[j], argv[i]))
            {
              argv[j] = NULL;
              nb_occ++;
            }
```

```
        if (nb_occ == nb)
          printf ("%s␣", argv[i]);
      }
  printf ("\n");
  return EXIT_SUCCESS;
}
```

Exercice 3.

Remarques :
- Peu d'étudiants ont correctement déclaré le type (mauvaise influence de java ?)
- la procédure renverser est un algorithme classique, vraiment difficile à bien implémenter sans s'aider de schémas et de variables locales significativement identifiées.

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

typedef struct maillon *lce;

struct maillon
{
  int valeur;
  lce suivant;
};

lce
cons (int valeur, lce suivant)
{
  lce l = malloc (sizeof (struct maillon));
  assert (l != NULL);

  l->valeur = valeur;
  l->suivant = suivant;

  return l;
}

lce
implante_12345 ()
{
  int i;
  lce l;

  for (l = NULL, i = 5; i > 0; i--)
    l = cons (i, l);

  return l;
}

int
est_croissante (lce l)
{
  if (l == NULL)
    return 1;

  for (; l->suivant != NULL && l->valeur < l->suivant->valeur; l = l->suivant)
    ;
  return l->suivant == NULL;
}
```

```c
void
renverse (lce * l)
{
  lce pred, courant, succ;
  assert (l != NULL);

  if (*l == NULL)
    return;

  pred = NULL;
  courant = *l;
  succ = courant->suivant;

  for (;;)
    {
      courant->suivant = pred;
      if (succ == NULL)
        break;
      pred = courant;
      courant = succ;
      succ = succ->suivant;
    };

  *l = courant;
}
```

Problème

Fichier en-tête du module ensemble

```c
#ifndef ENSEMBLE_H
#define ENSEMBLE_H

typedef struct ensemble *ensemble;

typedef void (*appliquer) (void *elt, void *parametre);


extern ensemble ens_creer ();
extern void ens_detruire (ensemble e);

extern void ens_ajouter (ensemble e, void *elt);
extern void ens_supprimer (ensemble e, void *elt);
extern int ens_appartient (ensemble e, void *elt);

extern ensemble ens_union (ensemble a, ensemble b);
extern ensemble ens_instersection (ensemble a, ensemble b);
extern ensemble ens_difference (ensemble a, ensemble b);

extern int ens_cardinal (ensemble e);

#endif
```

Fichier makefile

```makefile
CC = gcc                        #nom du compilateur
CFLAGS = -Wall -pedantic -g     #options de la compilation
LDFLAGS =                       #options du linker

SRC = ensemble.c vecteur_extensible.c ens-test.c
OBJ = $(SRC:.c=.o)
```

```
ens-test: $(OBJ)

ensemble.o: ensemble.h ensemble.c

vecteur_extensible.o: vecteur_extensible.h vecteur_extensible.c

ens-test.o: ens-test.c ensemble.h

clean:
	rm -f *.o *~ *.d ens-test
```

Corps du module ensemble

NB. On présente ici une solution à la dernière question : clairement, cette proposition étant plutôt digne d'un projet que d'un examen écrit, le correcteur n'attendait pas des étudiants une réponse aussi élaborée. Le module est disponible sur le Web du cours.

On pratique ici l'insertion *paresseuse*. Afin de favoriser les opérations ensemblistes (union, intersection et différence) on a choisi de conserver les éléments dans un tableau ordonné. La technique est paresseuse dans le sens où les séquences d'insertions (ou les séquences de suppressions) sont retardées au maximum (i.e. jusqu'à l'exécution d'une opération d'un autre type) : un tableau (souvent appelé *cache*) non ordonné annexe conserve les éléments a insérer. Lorsque les insertions sont inévitables (pour exécuter une opération ensembliste, par exemple) on normalise la structure en (a) triant le cache puis en (b) fusionnant le cache tableau et le tableau contenant les autres éléments.

Complexité extrême de l'insertion (à la louche[1]) : si le programme effectue p alternances de q insertions on a une complexité de l'ordre de $pq\log(q) + p^2q$ affectations.

Complexité du pire des cas $p = n, q = 1$: le programme alterne n fois une insertion et une opération ensembliste) la complexité de cette solution paresseuse est de l'ordre de n^2 affectations ; à l'opposé $p = 1, q = n$: le programme réalise d'abord les n insertions, la complexité est alors de l'ordre de $n\log(n)$ affectations.

p	q	$pq\log_2(q) + p^2q$
1	1000000	20 931569
10	100000	26 609640
100	10000	113 287712
1000	1000	1009 965784
10000	100	10006 643856
100000	10	100003 321928
500000	2	500001 000000
1000000	1	1 000000 000000

Comportement de la fonction $pq\log_2(q) + p^2q$ pour $pq = 1\ 000\ 000$

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

#include "ensemble.h"
#include "vecteur_extensible.h"

enum
{ TAILLE_INITIALE = 56 };

struct ensemble
{
  void **elements;
```

[1]Les enseignants du module ASD se feront une joie d'analyser la complexité amortie de cette structure de données

```
  size_t cardinal;
  vex delta;
  enum
  { NORMALISE, AUGMENTE, RESTREINT }
  etat;
};

static int compare_adresse (const void *a, const void *b);
static void trier (void **elements, size_t cardinal);
static void difference (void ***res, size_t * card_res,
                        void **a, size_t card_a, void **b, size_t card_b);
static void reunion (void ***res, size_t * card_res,
                     void **a, size_t card_a, void **b, size_t card_b);
static void ens_normaliser (ensemble e);

ensemble
ens_creer ()
{
  ensemble e = malloc (sizeof (struct ensemble));
  assert (e != NULL);
  e->elements = NULL;
  e->cardinal = 0;
  e->delta = vex_creer (TAILLE_INITIALE, 2, realloc);
  e->etat = NORMALISE;
  return e;
}

void
ens_detruire (ensemble e)
{
  assert (e != NULL);
  realloc (e->elements, 0);
  vex_detruire (e->delta);
  free (e);
}

void
ens_ajouter (ensemble e, void *elt)
{
  assert (e != NULL);
  if (e->etat == RESTREINT)
    ens_normaliser (e);
  vex_ajouter_element (e->delta, elt);
  e->etat = AUGMENTE;
}

void
ens_supprimer (ensemble e, void *elt)
{
  assert (e != NULL);
  if (e->etat == AUGMENTE)
    ens_normaliser (e);
  vex_ajouter_element (e->delta, elt);
  e->etat = RESTREINT;
}

int
ens_appartient (ensemble e, void *elt)
```

```
{
  assert (e != NULL);
  ens_normaliser (e);
  return bsearch (&elt, e->elements, e->cardinal,
                  sizeof (void *), compare_adresse) != NULL;
}

ensemble
ens_union (ensemble a, ensemble b)
{
  ensemble e;
  assert (a != NULL && b != NULL);
  ens_normaliser (a);
  ens_normaliser (b);
  e = ens_creer ();
  reunion (&e->elements, &e->cardinal,
           a->elements, a->cardinal, b->elements, b->cardinal);
  return e;
}

ensemble
ens_difference (ensemble a, ensemble b)
{
  ensemble e;
  assert (a != NULL && b != NULL);
  ens_normaliser (a);
  ens_normaliser (b);
  e = ens_creer ();
  difference (&e->elements, &e->cardinal,
              a->elements, a->cardinal, b->elements, b->cardinal);
  return e;
}

ensemble
ens_instersection (ensemble a, ensemble b)
{
  ensemble e;
  assert (a != NULL && b != NULL);
  ens_normaliser (a);
  ens_normaliser (b);
  e = ens_creer ();
  e->elements = malloc ((a->cardinal > b->cardinal ?
                         b->cardinal : a->cardinal) * sizeof (void *));
  {
    size_t i, j, p;
    for (i = j = p = 0; i < a->cardinal && j < b->cardinal;)
      if (a->elements[i] == b->elements[j])
        {
          e->elements[p] = a->elements[i];
          p++, i++, j++;
        }
      else if (a->elements[i] > b->elements[j])
        j++;
      else
        i++;
    e->cardinal = p;
  }
  e->elements = realloc (e->elements, e->cardinal * sizeof (void *));
```

```
  return e;
}

static void
ens_normaliser (ensemble e)
{
  void **delta, **norme;
  size_t nb_delta, nb_norme;
  if (e->etat == NORMALISE)
    return;
  nb_delta = vex_cardinal (e->delta);
  delta = vex_vers_vecteur (e->delta);
  trier (delta, nb_delta);
  if (e->etat == RESTREINT)
    difference (&norme, &nb_norme, e->elements, e->cardinal, delta, nb_delta);
  else
    reunion (&norme, &nb_norme, e->elements, e->cardinal, delta, nb_delta);
  free (e->elements);
  free (delta);
  e->elements = norme;
  e->cardinal = nb_norme;
  e->delta = vex_creer (TAILLE_INITIALE, 2, realloc);
  e->etat = NORMALISE;
}

static int
compare_adresse (const void *a, const void *b)
{
  return (*(void **) a > *(void **) b) - (*(void **) b > *(void **) a);
}

static void
trier (void **elements, size_t cardinal)
{
  qsort (elements, cardinal, sizeof (void *), compare_adresse);
}

static void
difference (void ***res, size_t * card_res,
            void **a, size_t card_a, void **b, size_t card_b)
{
  size_t i, j, p;
  *res = malloc (card_a * (sizeof (void *)));
  for (i = j = p = 0; i < card_a && j < card_b;)
    if (a[i] == b[j])
      i++, j++;
    else if (a[i] > b[j])
      j++;
    else
      (*res)[p++] = a[i++];
  if (i < card_a)
    {
      memcpy (*res + p, a + i, (card_a - i) * (sizeof (void *)));
      *card_res = p + card_a - i;
    }
  else
    *card_res = p;
  *res = realloc (*res, *card_res * sizeof (void *));
```

```
}

static void
reunion (void ***res, size_t * card_res,
         void **a, size_t card_a, void **b, size_t card_b)
{
  size_t i, j, p;
  *res = malloc ((card_a + card_b) * (sizeof (void *)));
  for (i = j = p = 0; i < card_a && j < card_b; p++)
    {
      if (a[i] > b[j])
        (*res)[p] = b[j++];
      else
        {
          (*res)[p] = a[i];
          if (a[i] == b[j])
            j++;
          i++;
        }
    }
  if (i < card_a)
    {
      memcpy (*res + p, a + i, (card_a - i) * (sizeof (void *)));
      *card_res = p + card_a - i;
    }
  else if (j < card_b)
    {
      memcpy (*res + p, b + j, (card_b - j) * (sizeof (void *)));
      *card_res = p + card_b - j;
    }
  else
    *card_res = p;
  *res = realloc (*res, *card_res * sizeof (void *));
}

int
ens_cardinal (ensemble e)
{
  assert (e != NULL);
  ens_normaliser (e);
  return e->cardinal;
}
```