Licence Informatique | **Initiation à la Programmation**
Université Bordeaux I | **Session de janvier 2001**

L'épreuve dure 3 heures et aucun document n'est autorisé.
Les programmes sont 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 mais surtout de la qualité du codage : lisibilité, maintenabilité, extensibilité et concision.
Barème indicatif : 3 - 5 - 4 - 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)
{
  int c;
  c = a + 1;
  a = *b + 1;
  *b = c;
  return a;
}

int
main(void)
{
  int a[2], *c = a;
  for (a[0] = a[1] = 0; a[0] < 5; a[1]++ )
    {
      printf("%d %d ", a[0], a[1]);
      printf("%d ", f(a[0], a));
      printf("%d\n", f(c[1], c + 1));
      a[0] = f(c[0], a + 1) + 1;
    }

  return EXIT_SUCCESS;
}
```

Exercice 2. Par définition, une distance est un élément de $I\!N \cup \{+\infty\}$.

Le but de l'exercice est de proposer un module offrant les fonctionnalités suivantes : créer/détruire une distance, tester si une distance est finie ou non, donner la valeur entière d'une distance finie et les fonctions somme, min et max de deux distances. On appliquera le principe de masquage de l'implémentation.

1. Proposer un fichier en-tête `distance.h`.
2. Proposer un fichier source (on pourra ommettre les fonctions somme, min et max).
3. En supposant que le fichier `mes_distances.c` contienne une fonction principale (`main`), écrire un fichier `Makefile` facilitant la compilation de ce programme.

Exercice 3. Écrire une fonction `unsigned long fibonacci(unsigned int n)` qui retourne F_n, le n-ième nombre de fibonacci ($F_0 = 0$, $F_1 = 1$ et $F_{n+1} = F_n + F_{n-1}$) tout en respectant les contraintes suivantes :

- le calcul doit être juste, pour cela on utilisera `assert` et la constante `ULONG_MAX` de `limits.h` ;
- les calculs doivent être mémorisés : on effectuera ni précalcul, ni recalcul.

Problème Soient les types suivants :

```
typedef enum {somme, difference, produit, quotient, valeur} sorte;
typedef struct expression *expression;
struct expression
{
  sorte s;
  expression droite, gauche; /* definies ssi s != valeur */
  int val;                   /* definie ssi s == valeur */
};
```

1. Écrire une fonction `expression expr1()` qui implante l'expression $(3+4)*(1-3)$.
2. Écrire une fonction `int evaluer(expression expr)` qui calcule la valeur d'une expression.
3. Écrire une fonction `char *image(expression expr)` qui retourne une chaîne de caractères représentant l'expression `expr` sous sa forme usuelle (forme infixe, comme dans cet énoncé). On soignera la gestion dynamique de la mémoire et, à cette fin, on aura intérêt à définir quelques fonctions annexes. Enfin, on pourra utiliser la fonction `char *itoa(int i)` qui retourne une chaîne (allouée par `malloc`) contenant l'image de l'entier `i`.

Au lieu d'utiliser un discriminant (`sorte s`) typant le noeud de l'expression puis une fonction d'évaluation réalisant l'appel à la fonction spécifique, on utilise un pointeur de fonction `evalue_moi` contenant l'adresse de la fonction d'évaluation spécifique à ce noeud et un objet :

```
typedef void *objet;
typedef struct expression *expression;
typedef int (*f_eval)(expression);
struct expression
{
  f_eval evalue_moi;
  objet argument;
};
```

Aussi la fonction `evaluer` s'écrit simplement :

```
int
evaluer(expression expr)
{
  return expr->evalue_moi(expr);
}
```

4. Écrire une fonction `expression creer_feuille(int i)` qui implante l'expression i.
   Cette fonction retourne un pointeur désignant une structure contenant :
   - un pointeur `evalue_moi` désignant une fonction `eval_feuille` de type `f_eval`;
   - un pointeur `argument` désignant un entier de valeur i.
5. Écrire une fonction `expression expr2()` qui implante $ite(3+1; 2+4; 6)$ où $ite(x; y; z)$ est la fonction if-then-else qui retourne la valeur de y si $x \neq 0$ et celle de z sinon.
6. On désire généraliser l'approche aux objets quelconques, c'est à dire faire en sorte que le résultat de l'évaluation d'une expression soit, non plus un entier, mais un objet (par exemple une distance, une pile, une matrice...).
   Quelle est la principale difficulté de programmation rencontrée?
   Proposer un ensemble de types et un exemple significatif de fonction d'évaluation (on pourra proposer une expression de distances, cf. Exercice 2).

Extraits du «Linux Programmer's Manual»

void assert (int expression); <assert.h>
assert() prints an error message to standard output and terminates the program by calling abort() if expression is false (i.e., compares equal to zero). This only happens when the macro NDEBUG is undefined.

size_t strlen(const char *s); <string.h>
The strlen() function calculates the length of the string s, not including the terminating '\0' character.

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 memory 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                      #nom du compilateur
CFLAGS = -Wall -pedantic -g   #options de la compilation
LDFLAGS  =                    #options du linker


convertir_en: convertir_en.o devise.o

convertir_en.o: convertir_en.c devise.h

devise.o:  devise.h devise.c

clean:
	rm -f *.o *~
```

Fin

Corrigé

Exercice 1.

```
0 0 1 1
3 3 4 4
```

Exercice 2.

1. Fichier entête
   Erreurs rencontrées :
   - Oublier de masquer l'implémentation (au plus 2 points de pénalité).
   - Ne pas initialiser via un paramètre la distance lors de sa création : créer une distance «vide» n'a aucun sens (au plus 1 point de pénalité).

```
#ifndef DISTANCE_H
#define DISTANCE_H

typedef struct distance *distance;

/* CONSTRUCTEURS / DESTRUCTEUR */

extern distance dist_creer_finie (unsigned long d);
extern distance dist_infinie ();
extern void dist_detruire (distance d);

/* INSPECTEURS */

extern int dist_est_infinie (distance d);
extern unsigned long dist_vers_ul (distance finie);
/* NB. finie doit representer une distance FINIE */

/* OPERATEURS */
/* ces éoprations écrent de nouvelles distances */

extern distance dist_somme (distance a, distance b);
extern distance dist_min (distance a, distance b);
extern distance dist_max (distance a, distance b);

#endif
```

2. Fichier source.
   Erreurs rencontrées :
   - inventer des données (extrémités, noms,...) ;
   - utiliser un même entier pour coder l'infini et la distance (aucune solution satisfaisante de ce type n'a été proposée : ce choix complique les programmes, de plus il faut en prévenir l'utilisateur ce qui va à l'encontre du masquage de l'implémentation.

```
#include <assert.h>
#include <malloc.h>
#include <limits.h>
#include "distance.h"

struct distance
  {
    unsigned long val;
    int infini;
  };
```

```
static distance
creer_distance (unsigned long d, int infini)
{
  distance a = (distance) malloc (sizeof (struct distance));
  assert (a != NULL);
  a->val = d;
  a->infini = infini;
  return a;
}

distance
dist_creer_finie (unsigned long d)
{
  return creer_distance (d, 0);
}

distance
dist_infinie ()
{
  return creer_distance (0, 1);
}

void
dist_detruire (distance d)
{
  assert (d != NULL);
  free (d);
}

int
dist_est_infinie (distance d)
{
  assert (d != NULL);
  return d->infini;
}

unsigned long
dist_vers_ul (distance finie)
{
  assert (!dist_est_infinie (finie));
  return finie->val;
}

/* HORS SUJET */

distance
dist_somme (distance a, distance b)
{
  assert(a != NULL && b != NULL);
  if (dist_est_infinie (a) && dist_est_infinie (b))
    return dist_infinie ();
  assert (a->val <= ULONG_MAX - b->val);
  return creer_distance (a->val + b->val, 1);
}

distance
dist_min (distance a, distance b)
```

```
{
  assert(a != NULL && b != NULL);
  if (!dist_est_infinie (a) && !dist_est_infinie (b))
    return creer_distance (a->val > b->val ? b->val : a->val, 0);
  if (a->infini)
    return creer_distance (b->val, b->infini);
  return creer_distance (a->val, a->infini);
}

distance
dist_max (distance a, distance b)
{
  assert(a != NULL && b != NULL);
  return
    creer_distance (a->val < b->val ? b->val : a->val,
                    a->infini || b->infini);
}
```

3. Fichier Makefile :
   Erreur commises : oublier la compilation du fichier `distance.c`, ne pas mettre en premier la cible `mes_distances` .

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

mes_distances: mes_distances.o distance.o
mes_distances.o: mes_distances.c distance.h
distance.o: distance.h distance.c

clean:
        rm -f *.o *~
```

Exercice 3.

Calculer le n-ième nombre de fibonacci en C est un exercice abordé dès la deuxième semaine du cours d'IP du DEUG MIAS de notre université. La difficulté de l'exercice était de réaliser ce calcul tout en respectant les deux consignes. Sur le plan technique, on peut noter quelques erreurs :

```
unsigned long x;
...
assert(x => 0);
assert(x <= ULONG_MAX);
 /* vérifications inutiles par définition */

static int *a = malloc(10) ;
 /* impossible : la valeur doit etre connue lors de la compilation */

int a[10] ;
a = realloc(a,10);
/* impossible a est constant */

int *a = {10,12};
a = realloc(a,10);
/* impossible a n'a pas été initialisé par un malloc */
```

Style de solution attendue :

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

unsigned long
fibonacci (unsigned int n)
{
  static unsigned long non_calcule = 0;
  static unsigned long *valeurs = NULL;
  if (n >= non_calcule)
    {
      size_t taille = (n > 1) ? n + 1 : 2;
      valeurs = (unsigned long *) realloc (valeurs, taille * sizeof (unsigned long));

      assert (valeurs != NULL);
      if (non_calcule == 0)
        {
          valeurs[0] = 0;
          valeurs[1] = 1;
          non_calcule = 2;
        }

      for (; non_calcule <= n; non_calcule++)
        {
          assert (ULONG_MAX - valeurs[non_calcule - 2] > valeurs[non_calcule - 1]);
          valeurs[non_calcule] = valeurs[non_calcule - 1] + valeurs[non_calcule - 2];
        }
    }
  return valeurs[n];
}
```

Problème

Première partie (sur 6 points) :

Erreurs commises :

- produire une solution non lisible, non extensible et longue ;
- oublier d'allouer de l'espace pour chacune des sous expressions ;
- confondre valeur d'un type énuméré et chaine de caractères (`produit` et `"produit"`) ;
- ne pas allouer assez d'espace mémoire ;
- ne pas désallouer l'espace mémoire (en particulier celui utilisé par la fonction `itoa`) ;
- utiliser l'affectation ou la somme pour concaténer deux chaines.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <malloc.h>
#include <assert.h>

typedef enum  { somme, difference, produit, quotient, valeur} sorte;

typedef struct expression *expression;

struct expression
  {
    sorte s;
    expression gauche, droite;  /* definies ssi s != valeur */
    int val;                    /* definie ssi s == valeur */
```

```
  };

expression
exp (sorte s, expression gauche, expression droite, int val)
{
  expression e = (expression) malloc (sizeof (struct expression));
  e->s = s;
  e->droite = droite;
  e->gauche = gauche;
  e->val = val;
  return e;
}

#define PLUS(a,b) exp (somme,(a),(b),0)
#define MOINS(a,b) exp (difference,(a),(b),0)
#define MUL(a,b) exp (produit,(a),(b),0)
#define DIV(a,b) exp (quotient,(a),(b),0)
#define VAL(a) exp (valeur,NULL,NULL,(a))

expression
expr1 ()                        /* construire (3+4)*(1-3) */
{
  return MUL (PLUS (VAL (3), VAL (4)), MOINS (VAL (1), VAL (3)));
}

int
evaluer (expression e)
{
  if (e->s == valeur)
    return e->val;
  {
    int g = evaluer (e->gauche);
    int d = evaluer (e->droite);

    switch (e->s)
      {
      case somme:
        return g + d;
      case difference:
        return g - d;
      case produit:
        return g * d;
      case quotient:
        assert (d != 0);
        return g / d;
      default:
        assert ("erreur" == NULL);
        return 0;
      }
  }
}

/* image nous vous proposons deux solutions */
/* èpremire solution rapideà é crire mais peu performante
   mais surtout infaisable en DS */

char *
format2str (char *fmt,...)
```

```
{
  char *p;
  va_list ap;
  va_start (ap, fmt);
  p = malloc (vsnprintf (NULL, 0, fmt, ap) + 1);
  assert (p != NULL);
  va_end (ap);
  va_start (ap, fmt);
  vsprintf (p, fmt, ap);
  va_end (ap);
  return p;
}

char *
image_court (expression e)
{
  static char *op_image[] = {"+", "-", "*", "/"};
  if (e->s == valeur)
    return format2str ("%d", e->val);
  else
    {
      char *res, *g, *d;
      g = image_court (e->gauche);
      d = image_court (e->droite);
      res = format2str ("(%s%s%s)", g, op_image[e->s], d);
      free (g);
      free (d);
      return res;
    }
}

char *
itoa (int i)                    /* fonction édonne */
{
  return format2str ("%d", i);
}

/*************************************************************/
/* Solution attendue */

typedef struct chaine
{
  char *texte;
  size_t longueur;
  size_t taille_allouee;
} *chaine;

void
ajouter (chaine c, char *s)
{
  size_t lg = strlen (s);
  if (c->longueur + lg > c->taille_allouee)
    {
      c->taille_allouee = (c->longueur + lg + 1) * 2;
      c->texte = realloc (c->texte, c->taille_allouee * sizeof (char));
    }
  strcpy (c->texte + c->longueur, s);
  c->longueur += lg;
```

```
}

void
construire_image (chaine c, expression e)
{
  static char *op_image[] = {"+", "-", "*", "/"};
  if (e->s == valeur)
    {
      char *s = itoa (e->val);
      ajouter (c, s);
      free (s);
      return;
    }
  ajouter (c, "(");
  construire_image (c, e->gauche);
  ajouter (c, op_image[e->s]);
  construire_image (c, e->droite);
  ajouter (c, ")");
}

char *
image (expression e)
{
  struct chaine c = (struct chaine) {NULL, 0, 0};
  construire_image (&c, e);
  c.texte = realloc (c.texte, (c.longueur + 1) * sizeof (char));
  return c.texte;
}

int
main ()
{
  char *c;
  c = image_court (expr1 ());
  printf ("%s = %d = %s", c, evaluer (expr1 ()), image (expr1 ()));
  return EXIT_SUCCESS;
}
```

Seconde Partie (sur 2 points) :

Erreurs commises :

- ne pas fournir la fonction `eval_feuille` ;
- ne pas créer l'espace mémoire pour l'entier.

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

typedef struct expression *expression;
typedef int (*f_eval) (expression);
typedef void *objet;

struct expression
{
    f_eval evalue_moi;
    objet argument;
};
```

```
int evaluer(expression e)
{
  return e->evalue_moi (e) ;
}

/* premiere question */

expression
exp_creer (f_eval f, objet o)
{
  expression expr;
  expr = (expression) malloc (sizeof (expression));
  expr->evalue_moi = f;
  expr->argument = o;
  return expr;
}

int
eval_feuille (expression expr)
{
  return *((int *) expr->argument);
}

expression
exp_feuille (int i)
{
  int *p;
  p = (int *) malloc (sizeof (int));
  assert (p);
  *p = i;
  return exp_creer (eval_feuille, p);
}

/* seconde question */

#define EVAL_ARG(expr,i) evaluer(((expression *)((expr)->argument))[(i)])

int
eval_plus (expression expr)
{
  return  EVAL_ARG(expr,0) + EVAL_ARG(expr,1);
}

int
eval_ite (expression expr)
{
  return EVAL_ARG(expr,0) ? EVAL_ARG(expr,1) : EVAL_ARG(expr,2);
}

expression
exp_binaire (f_eval f, objet arg1, objet arg2)
{
  objet *tab;
  tab = malloc (sizeof (2 * sizeof (objet)));
  tab[0] = arg1;
  tab[1] = arg2;
  return exp_creer (f, tab);
}
```

```
/* plutot que d'ecrire expression ternaire je prefere generaliser */

expression
exp (f_eval f, size_t arite,...)
{
  if (arite == 0)
    return exp_creer (f, NULL);
  {
    va_list ap;
    objet *tab;
    int i;
    tab = (objet) malloc (sizeof (arite * sizeof (objet)));
    va_start (ap, arite);
    for (i = 0; i < arite; i++)
      tab[i] = (objet) va_arg (ap, void *);
    va_end (ap);
    return exp_creer (f, tab);
  }
}

#define PLUS(a,b) exp_binaire (eval_plus, (a), (b))
#define ITE(a,b,c) exp (eval_ite, 3, (a), (b), (c))

expression
expr2()                /* ite(3+1;2+4;6) */
{
  return ITE ( PLUS( exp_feuille (3), exp_feuille (1)),
               PLUS( exp_feuille (2), exp_feuille (4)),
               exp_feuille (6));
}

int
main ()
{
  printf ("%d\n", evaluer(expr2()));
  return EXIT_SUCCESS;
}
```

Troisième partie : question 6. (hors barème)

En réponse à cette question, quelques étudiants ont évoqué l'absence de typage, une des principale difficulté de la programmation C (ici le typage est réalisé par la fonction d'évaluation). L'autre travers du C concerne la gestion de la mémoire ; c'est ici la principale *nouvelle* difficultée. Il nous faut définir une politique de gestion de la mémoire, en voici quelques une :

- Libérer systématiquement les résultats intermédiaires produit lors d'une évaluation. Ceci implique que toute fonction d'évaluation doit retourner un objet allouée par son action – en particulier, la fonction d'évaluation d'une feuille devra dupliquer la valeur de celle-ci.
- Associer un *mode d'emploi* à tout résultat intermédiaire, ce mode d'emploi contiendra une fonction libération (qui peut très bien ne rien libérer du tout, dans le cas d'une feuille par exemple) et une fonction de duplication (utile pour retourner le résultat d'une expression).
- Automatiser la gestion mémoire grâce à l'utilisation d'un ramasse miettes (*garbage collector*).