Licence Informatique
Université Bordeaux I

Initiation à la Programmation
Session de Septembre 2001

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 - 4 - 9

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;
}

int
main(void)
{
  int a[2], *c = a + 1;
  for (a[0] = a[1] = 0; a[1] < 5; 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. La commande `egaux` compare ses arguments et retourne soit `EXIT_SUCCESS` (valant 0), soit `EXIT_FAILURE` (valant 1). Cette commande prend en compte les options suivantes :
`egaux -tous` teste si les arguments sont tous égaux ;
`egaux -aucun` teste si les arguments sont deux à deux distincts ;
`egaux -certains` teste si les arguments ne sont ni tous égaux, ni tous distincts.

1. écrire la fonction principale (`main`) qui, après avoir analysé la ligne de commande, appelle une fonction spécifique à chacune des options ;
2. écrire la fonction correspondant à l'option `-aucun`.

Exemples :

```
>egaux -aucun un deux un
>echo $?           # affiche la valeur de la variable status
1                  # equivalent à faux   (EXIT_FAILURE)
>egaux -aucun un deux
>echo $?
0                  # equivalent à vrai (EXIT_SUCCESS)
```

Exercice 3. Écrire une fonction `unsigned long u(unsigned long n)` qui retourne $u_n = u_{n-1}^2 - u_{n-1} + 1$ avec $u_0 = 2$ tout en respectant les contraintes suivantes :

- le calcul doit être juste, pour cela on utilisera `assert` et la constante `ULONG_MAX` de `limits.h` pour prévenir de tout débordement ;
- les valeurs successives de la suite doivent être mémorisées dans un tableau statique alloué dynamiquement, et ce afin de ne pas répéter les mêmes calculs au cours de l'exécution d'un programme.

Problème Soient les types suivants :

```
typedef struct individu *individu;
struct individu
{
  char *nom;
  char *prenom;
  int date_naissance[3]; // jour mois annee
  individu pere, mere; // NULL si inconnu
  individu *enfants;   // ensemble d'individus terminee par le pointeur NULL
};
```

Le but de l'exercice est de proposer un module offrant les 4 fonctionnalités suivantes : créer/détruire un individu, déterminer l'ensemble de sa descendance et l'ensemble de ses cousins (enfants des tantes et oncles d'un individu).

On appliquera le principe de masquage de l'implémentation.

Les ensembles d'indivus (exempts de doublons) seront représentés par un tableau d'individus terminé par le pointeur NULL (de la même façon que le caractère `'\0'` termine toute chaîne de caractères).

1. Proposer un fichier en-tête `généalogie.h` (2 points).
2. Proposer un fichier `la_famille_smurf.c` qui désigne la famille Smurf composée de 2 parents et d'un enfant (1 point).
3. Ecrire un fichier `Makefile` facilitant la compilation du fichier `la_famille_smurf.c` (1 point).
4. Proposer un fichier source `généalogie.c` contenant, *entre autre*, le code des fonctions demandées (5 points). Il est fortement conseillé de définir et d'utiliser des fonctions manipulant les ensembles d'individus (création d'un ensemble, insertion et suppression d'un élément).

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>
```

Exemple de fichier Makefile

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

convertir_en: convertir_en.o devise.o

convertir_en.o: convertir_en.c devise.h

devise.o:  devise.h devise.c
```

— FIN —

Corrigé

Exercice 1.

```
0 0 1 2
1 3 5 11
```

Exercice 2.

Erreurs commises :

- utiliser un test comme `argv[n] == NULL` pour déterminé le nombre d'arguments au lieu d'utiliser la variable (traditionnellement appelée) `argc` qui contient le nombre de mots présents sur la ligne de commande.
- utiliser un aiguillage (switch) pour comparer une chaîne de caractères (par exemple `switch (argv[1])`) au lieu d'utiliser la fonction `strcmp()`.

```
#include <stdlib.h>

typedef int (*opt_fun) (int nb_arg, char *arg[]);

typedef struct
{
  char *nom;
  char *commentaire;
  opt_fun fonction;
}
option;

int tous (int nb_arg, char *arg[]);
int aucun (int nb_arg, char *arg[]);
int certains (int nb_arg, char *arg[]);

option les_options[] = {
  {"-tous", "tous␣egaux", tous},
  {"-aucun", "tous␣2␣a␣2␣differents", aucun},
  {"-certains", "ni␣tous␣egaux,␣ni␣tous␣differents", certains}
};

#define NB_OPTIONS (sizeof (les_options) / sizeof (option))

void
usage (char *commande)
{
  size_t i;
  printf ("usage␣:␣\n");
  printf ("%␣-option␣arg1␣arg2␣...␣argn\n", commande);
  printf ("%␣teste␣si␣arg1␣arg2␣...␣argn␣sont\n", commande);
  for (i = 0; i < NB_OPTIONS; i++)
    printf ("\t␣%s␣(%s)\n", les_options[i].nom, les_options[i].commentaire);
  exit (EXIT_FAILURE);
}

int
main (int argc, char *argv[])
{
  size_t opt;
  int positon_arg = 1;
  if (argc > 1)
    for (opt = 0; opt < NB_OPTIONS; opt++)
      if (!strcmp (argv[1], les_options[opt].nom))
        return les_options[opt].fonction (argc - 2, argv + 2);
```

```
  usage (argv[0]);
  return EXIT_FAILURE;
}

int
aucun (int nb_arg, char *argv[])
{
  size_t i, j;
  for (i = 0; i < nb_arg - 1; i++)
    for (j = i + 1; j < nb_arg; j++)
      if (!strcmp (argv[i], argv[j]))
        return EXIT_FAILURE;
  return EXIT_SUCCESS;
}

int
tous (int nb_arg, char *argv[])
{
  size_t i, j;
  for (i = 1; i < nb_arg; i++)
    if (strcmp (argv[0], argv[i]))
      return EXIT_FAILURE;
  return EXIT_SUCCESS;
}

int
certains (int nb_arg, char *argv[])
{
  return !aucun (nb_arg, argv) && !tous (nb_arg, argv);
}
```

Exercice 3.

Erreurs commises :
- utiliser un test du genre `(ULONG_MAX => x)` cette assertion est toujours vraie pour tout entier long `x` ;
- ne pas mémoriser le résultat de l'appel à la fonction `realloc()`.

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

unsigned long
u (unsigned long n)  /* u_{n+1} = u_n (u_n - 1) + 1  , u_0 = 2 */
{
  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] = 2;
          non_calcule = 1;
        }
```

```
    for (; non_calcule <= n; non_calcule++)
      {
        unsigned long val = valeurs[non_calcule - 1];
        assert ((ULONG_MAX - 1) /  val >= val - 1);
        valeurs[non_calcule] = val * (val - 1) + 1;
      }
    }
  return valeurs[n];
}
```

Problème

Commentaires : peu de problème pour les fichiers `genealogie.h` et `Makefile`, catastrophique pour les deux autres.

Fichier `genealogie.h`

```
#ifndef GENEALOGIE_H
#define  GENEALOGIE_H

typedef struct individu *individu;

typedef individu *population;

population pop_creer();
void pop_detruire(population p);
population pop_ajouter(population p, individu i);
population pop_supprimer(population p,  individu i);

individu indi_creer(char *nom, char *prenom, int date_naissance[3], individu pere, individu m
void indi_supprimer(individu moi);
population indi_descendance(individu i);
population indi_cousins(individu i);


#endif // GENEALOGIE_H
```

$* \quad * \quad *$

Fichier `famille_smurf.c`

```
#include <stdlib.h>
#include "genealogie.h"

int
main ()
{
  individu pere, mere, enfant;
  int dp[] = { 10, 12, 1956 };
  int dm[] = { 11, 02, 1954 };
  int de[] = { 10, 12, 1980 };
  pere = indi_creer ("Smurf", "Maurice", dp, NULL, NULL);
  mere = indi_creer ("Frums", "Anais", dm, NULL, NULL);
  enfant = indi_creer ("Smurf", "Alice", de, pere, mere);

  indi_supprimer (mere);
  indi_supprimer (enfant);
  indi_supprimer (pere);

  return EXIT_SUCCESS;
}
```

* * *

Fichier `Makefile`

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


famille_smurf: famille_smurf.o genealogie.o

famille_smurf.o: famille_smurf.c genealogie.h

genealogie.o:  genealogie.h genealogie.c

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

* * *

Fichier `genealogie.c`

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

struct individu
{
  char *nom;
  char *prenom;
  int date_naissance[3];          // jour mois éanne
  individu parents[2];            // pere, mere (NULL si inconnu)
  population enfants;             // suite d'individus termine par le pointeur NULL
};

population pop_creer ()
{
  population p = malloc (sizeof (individu));
  assert (p != NULL);
  *p = NULL;
  return p;
}

void
pop_detruire (population p)
{
  free (p);
}

population pop_ajouter (population p, individu i)
{
  size_t n;
  assert (p != NULL);
  for (n = 0; p[n] != NULL; n++)
    if (p[n] == i)
      return p;
  p = realloc (p, (n + 2) * sizeof (individu));
  assert (p != NULL);
  p[n] = i;
  p[n + 1] = NULL;
```

```
  return p;
}

population pop_supprimer (population p, individu i)
{
  size_t n;
  for (n = 0; p[n] != NULL; n++)
    if (p[n] == i)
      {
        // translater le tableau
        for (; p[n] != NULL; n++)
          p[n] = p[n + 1];
        p = realloc (p, n * sizeof (individu));
        assert (p != NULL);
        return p;
      }
  // pas trouve
  return p;
}

individu
indi_creer (char *nom, char *prenom, int date_naissance[3],
            individu pere, individu mere)
{
  size_t n;
  individu moi = malloc (sizeof (struct individu));
  assert (moi);
                                                                            // *INDENT-OFF*
  *moi = (struct individu) { nom, prenom,
                             {date_naissance[0], date_naissance[1], date_naissance[2]},
                             {pere, mere},  pop_creer () };
                                                                            // *INDENT-ON*
  assert (moi->enfants != NULL);
  for (n = 0; n < 2; n++)
    if (moi->parents[n] != NULL)
      moi->parents[n]->enfants = pop_ajouter (moi->parents[n]->enfants, moi);
  return moi;
}

void
indi_supprimer (individu moi)
{
  size_t n;
  assert (moi != NULL);
  // se supprimer chez ses parents
  for (n = 0; n < 2; n++)
    if (moi->parents[n] != NULL)
      moi->parents[n]->enfants =
        pop_supprimer (moi->parents[n]->enfants, moi);
  // se supprimer chez ses enfants
  for (n = 0; moi->enfants[n] != NULL; n++)
    moi->enfants[n]->parents[moi->enfants[n]->parents[1] == moi] = NULL;
  pop_detruire (moi->enfants);
  free (moi);
}

static void
descendance (individu i, population * p)
```

```c
{
  size_t n;
  for (n = 0; i->enfants[n] != NULL; n++)
    {
      *p = pop_ajouter (*p, i);
      descendance (i->enfants[n], p);
    }
}

population indi_descendance (individu i)
{
  population p = pop_creer ();
  assert (i != NULL);
  descendance (i, &p);
  return p;
}

static population
grands_parents (individu i)
{
  population gp = pop_creer ();
  size_t n, m;
  for (n = 0; n < 2; n++)
    if (i->parents[n] != NULL)
      for (m = 0; m < 2; m++)
        if (i->parents[n]->parents[m] != NULL)
          gp = pop_ajouter (gp, i->parents[n]->parents[m]);
  return gp;
}

static population
oncles_et_tantes (individu i)
{
  population res = pop_creer (), gp;
  size_t n, m;
  gp = grands_parents (i);
  for (n = 0; gp[n] != NULL; n++)       // pour tous les grands parents
    for (m = 0; gp[n]->enfants[m] != NULL; m++) // et tous leurs enfants
      if (gp[n]->enfants[m] != i->parents[0]
          && gp[n]->enfants[m] != i->parents[1])
        res = pop_ajouter (res, gp[n]->enfants[m]);
  pop_detruire (gp);
  return res;
}

population indi_cousins (individu i)
{
  population cousins = pop_creer (), tatas_tontons = oncles_et_tantes (i);
  size_t n, m;
  for (n = 0; tatas_tontons[n] != NULL; n++)
    for (m = 0; tatas_tontons[n]->enfants[m] != NULL; m++)
      cousins = pop_ajouter (cousins, tatas_tontons[n]->enfants[m]);
  return pop_supprimer (cousins, i);    // je ne suis pas mon propre cousin !
}
```