*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 3 8 (2 points restent à distribuer)*

Exercice 1. Décrire, en 10 lignes maximum au total, les fonctions `malloc`, `free` et `realloc`. Schématiser une structure de données et décrire des algorithmes simples illustrant le fonctionnement interne de ces fonctions.

Exercice 2. La commande `filtrer` applique successivement plusieurs filtres à une même image. Chaque filtre nécessite un *nombre prédéfini* d'arguments entiers, comme l'illustrent les exemples suivants :

```
> filtrer -laplacien -seuiller 240 in.pgm out.pgm
> filtrer -perturber 1 20 -median image.pgm brouille.pgm
> filtrer -perturber 1 -median image.pgm brouille.pgm
-median n'est pas un entier
usage : filtrer {<filtre> <parametres>} fichier_entree fichier_sortie
liste des filtres :
    -laplacien
    -seuiller seuil
    -perturber a b
    -median
```

En utilisant les modules donnés ci-dessous :

1. Définir un tableau rassemblant toutes les informations nécessaires à l'exécution de chacun des filtres (nom du filtre, commentaires, pointeur de fonction, etc).
2. Écrire le fichier `filtrer.c`.

Exercice 3. Il s'agit d'implanter en mémoire des tableaux d'entiers dynamiques à 3 dimensions (`tab3D`) de telle façon à pouvoir les utiliser le plus *naturellement* possible, i.e. si possible comme les tableaux déclarés statiquement dans le code (e.g. `int tab[2][6][2]`).

1. Dessiner l'implantation mémoire d'un tableau dynamique 3D, définir le type `tab3D`.
2. Écrire le code de `tab3D tab3D_creer(size_t m, size_t l, size_t c)`.
3. Écrire le code de `void tab3D_detruire(tab3D t)`.

N.B. Une réponse simplifiée au cas 2D pourra rapporter des points.

Problème Il s'agit de présenter un module générique `curliste` traitant des listes avec curseur. Un curseur désigne une position entre deux éléments dans une liste. Les fonctionnalités de ce module sont :

- la création et la destruction d'une liste avec curseur ;
- la gestion du déplacement du curseur (aller au début ou à la fin de la liste, avancer ou reculer d'un cran dans la liste) ;
- l'insertion d'un élément devant le curseur et la suppression de l'élément suivant le curseur ;
- de donner la valeur de l'élément suivant le curseur ;
- de donner des renseignements sur position du curseur (être au début, être à la fin) ;
- de donner le cardinal de la liste.

1. En appliquant le principe du masquage de l'implémentation, définir l'interface du module `curliste`.
2. En supposant que le programme `listing.c` fasse appel au module `curliste` écrire son fichier `makefile`.

On souhaite implémenter les curlistes à l'aide de liste circulaire doublement chaînées. Au départ, la liste sera composée d'un élément initial ayant pour successeur le début de la liste (initialement lui-même) et pour prédécesseur la fin de la liste (initialement lui-même).

3. Après avoir dessiné la structure de données implantant une curliste à 4 éléments (ne pas oublier le curseur), écrire les déclarations des types du fichier source du module `curliste`.
4. Écrire le code des fonctions de création et de destruction d'une curliste.
5. Écrire le code de la fonction d'insertion d'un élément devant le curseur.
6. Écrire le code de la fonction de suppression de l'élément suivant le curseur.
7. On désire travailler avec plusieurs curseurs. Proposer une interface et une structure de données adaptée à ce problème. Préciser les algorithmes de suppression d'un élément et de destruction d'une curliste.

– FIN –

CORRECTION

Barème effectif : 4 - 6 - 4 - 8

Exercice 1.

Le module `malloc.h` gère, dans le tas, une structure de données permettant l'allocation dynamique de mémoire. Les fonctions de ce module manipulent des blocs (zones contiguës de données) au travers de l'adresse de leur premier octet utile.

L'appel `malloc(taille)` (équivalent à `realloc(NULL, taille)`) réserve dans le tas un bloc d'au moins `taille` octets et retourne son adresse.

L'appel `free(adr)` (équivalent à `realloc(adr,0)`) restitue le bloc (préalablement réservé) désigné par `adr`.

L'appel `realloc(adr, t)` porte si possible à `t` la taille du bloc désigné et retourne son adresse, autrement (le bloc ne pas être étendu en place) `realloc` déplace les données du bloc désigné par `adr` dans un nouveau bloc d'au moins `t` octets, restitue l'ancien bloc et retourne la nouvelle adresse.

Une structure de données possible est (1) de maintenir une liste chaînée des blocs libres et (2) de préfixer tout bloc d'un en-tête contenant la taille du bloc :

- La fonction `malloc()` cherche dans la liste des blocs libres un bloc de taille suffisante (un appel système sera effectué au besoin pour agrandir le tas) puis, au besoin, scinde le bloc choisi en deux : un a attribuer, l'autre a conserver dans la liste des blocs libres.
- La fonction `free` insère (ou fusionne) le bloc restitué dans la liste des blocs libres.
- la fonction `realloc` peut être réalisée en
 - combinant `malloc`, `memcpy`, `free` pour allouer un bloc plus grand par exemple,
 - scindant le bloc en 2 pour en libérer une partie,
 - ne rien faisant : il n'est pas forcément utile (ni possible) de libérer 3 octets ici et 20 octets là !

Exercice 2.

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

#include "image.h"
#include "filtre.h"
#include "libfiltre.h"

typedef struct fonctionnalite
{
    char *commande;
    char *description;
    filtre f;
    size_t nb_parametres;
    size_t filtreX;
    size_t filtreY;
}
*fonctionnalite;

struct fonctionnalite lesFiltres[] = {
    {"PerturbationAleatoire", "uu", PerturbationAleatoire00, 2, 0, 0},
    {"Contraction", "u", Contraction00, 2, 0, 0},
    {"Median", "uuuuu", Median11, 0, 3, 3},
    {"Laplacien", "uuuu", Laplacien11, 0, 3, 3},
    {"Moyenne", "uu", Moyenne11, 0, 3, 3},
    {"Inversion", "uuu", Inversion00, 0, 1, 1},
};
```

```

const int NBOPTIONS = sizeof (lesFiltres) / sizeof (struct fonctionnalite);

enum
{ MAXPARAMETRES = 2 };

void
usage (char *argv0)
{
    size_t i;

    fprintf (stderr, "usage : %s {filtre parametre}"
             "<éichier_entre><fichier_sortie>\n", argv0);
    for (i = 0; i < NBOPTIONS; i++)
        fprintf (stderr, "\t %s : %s\n",
                 lesFiltres[i].commande, lesFiltres[i].description);

    exit (EXIT_FAILURE);
}

fonctionnalite
TrouverFiltre (char *arg)
{
    size_t i;
    for (i = 0; i < NBOPTIONS; i++)
        if (!strcmp (arg, lesFiltres[i].commande))
            return lesFiltres + i;
    return NULL;
}

long
argVersEntier (char *argv0, char *s)
{
    char *endptr;
    long l = strtol (s, &endptr, 10);

    if (*endptr != '\0' || errno == ERANGE)
        {
            fprintf (stderr, "%s n'est pas un entier valide\n", s);
            usage (argv0);
        }
    return l;
}

int
main (int argc, char *argv[])
{
    image Image;
    size_t arg;
    fonctionnalite opt;

    if (argc < 3)
        usage (argv[0]);

    Image = ImageLire (argv[argc - 2]);

    for (arg = 1; arg < argc - 2; arg += 1 + opt->nb_parametres)
        {
            size_t i;

```

```

    int parametres[MAXPARAMETRES];

    opt = TrouverFiltre (argv[arg]);

    if (opt == NULL)
        usage (argv[0]);

    for (i = 0; i < opt->nb_parametres; i++)
        parametres[i] = argVersEntier (argv[0], argv[arg + i + 1]);

    FiltrerImage (opt->f, Image, parametres, opt->filtreX, opt->filtreY);
}

ImageEcrire (Image, argv[argc - 1]);
ImageDetruire (Image);

return EXIT_SUCCESS;
}

```

Exercice 3.

```

#include <stdlib.h>

typedef int ***tab3D;

tab3D
tab3D_creer(size_t m, size_t l, size_t c)
{
    tab3D t;
    size_t i, j;

    t = (tab3D) malloc (m * sizeof (int **)) ;
    t[0] = (int **) malloc (m * l * sizeof (int *)) ;
    t[0][0] = (int *) malloc (m * l * c * sizeof (int)) ;

    for (i = 0; i < m; i++)
    {
        t[i] = t[0] + i * l ;
        for (j = 0; j < l; j++)
            t[i][j] = t[0][0] + i * l * c + j * c;
    }
    return t;
}

void
tab3D_detruire(tab3D t)
{
    free(t[0][0]);
    free(t[0]);
    free(t);
}

```

Problème

Fichier en-tête

```

#ifndef CURLIST_H
#define CURLIST_H

#include <stdlib.h>

typedef struct curliste *curliste;

```

```

typedef enum
{ auDebut, enAvant, enArriere, alaFin }
mouvement;

typedef void (*destruction_f) (void *objet, void *parametre);

curliste cl_creer ();
void cl_detruire (curliste cl, destruction_f f, void *parametre);

size_t cl_cardinal (curliste cl);

void cl_inserer_devant_curseur (curliste cl, void *element);
void *cl_supprimer_apres_curseur (curliste cl);
void *cl_valeur_apres_curseur (curliste cl);

void cl_deplacer_curseur (curliste cl, mouvement mvt);
int cl_est_au_debut (curliste cl);
int cl_est_en_fin (curliste cl);
#endif

```

Fichier makefile

```

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

SRC = curliste.c listing.c
OBJ = $(SRC:.c=.o)

listing: $(OBJ)

curliste.o: curliste.h curliste.c

listing.o: listing.c curliste.h

clean:
    rm -f *.o *~ listing

```

Fichier source

Pour faciliter l'écriture des fonctions de gestion des listes bilatères, il est commode (sinon d'usage) de gérer cette liste comme une liste circulaire : on crée la liste à partir d'un maillon initial qui ne comportera jamais d'élément mais sert uniquement à marquer le début et la fin de la liste..

```

#include "curliste.h"

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

typedef struct maillon *maillon;

struct maillon
{
    void *element;
    maillon pred;
    maillon succ;
};

struct curliste
{
    struct maillon liste;

```

```

    maillon curseur;
    size_t cardinal;
};

curliste
cl_creer ()
{
    curliste cl = (curliste) malloc (sizeof (*cl));
    cl->curseur = cl->liste.pred = cl->liste.succ = &cl->liste;
    cl->cardinal = 0;
    return cl;
}

void
cl_detruire (curliste cl, destruction_f f, void *parametre)
{
    maillon m = cl->liste.succ;
    while (m != &cl->liste)
    {
        maillon tmp = m;
        m = m->succ;
        if (f != NULL)
            f (tmp->element, parametre);
        free (tmp);
    }
    free (cl);
}

size_t
cl_cardinal (curliste cl)
{
    return cl->cardinal;
}

void
cl_insérer_devant_curseur (curliste cl, void *element)
{
    maillon m = (maillon) malloc (sizeof (*m));
    m->element = element;
    m->pred = cl->curseur->pred;
    m->succ = cl->curseur;

    m->pred->succ = m;
    cl->curseur->pred = m;
    cl->cardinal++;
}

void *
cl_supprimer_apres_curseur (curliste cl)
{
    maillon vieux_curseur = cl->curseur;
    void *tmp = vieux_curseur->element;

    assert (!cl_est_en_fin (cl));

    vieux_curseur->pred->succ = vieux_curseur->succ;
    vieux_curseur->succ->pred = vieux_curseur->pred;
    cl->curseur = vieux_curseur->succ;
}

```

```

    cl->cardinal--;

    free (vieux curseur);

    return tmp;
}

void *
cl_valeur_succ curseur (curliste cl)
{
    assert (!cl_est_en_fin (cl));
    return cl->curseur->element;
}

void
cl_deplacer curseur (curliste cl, mouvement mvt)
{
    switch (mvt)
    {
        case auDebut:
            cl->curseur = cl->liste.succ;
            return;
        case enAvant:
            assert (!cl_est_en_fin (cl));
            cl->curseur = cl->curseur->succ;
            return;
        case enArriere:
            assert (!cl_est_au_debut (cl));
            cl->curseur = cl->curseur->pred;
            return;
        case alaFin:
            cl->curseur = &cl->liste;
            return;
    }
}

int
cl_est_au_debut (curliste cl)
{
    return cl->curseur == cl->liste.succ;
}

int
cl_est_en_fin (curliste cl)
{
    return cl->curseur == &cl->liste;
}

```

Question 7. Il s'agit d'introduire plusieurs curseurs sans alourdir l'interface (sans ajouter des pointeurs ou numeros de curseur partout) et de pouvoir conserver la cohérence des structures de données lors de la suppression d'un élément (il ne faut pas qu'un curseur traîne sur un élément supprimé). On ajoute les fonctions suivantes :

```

curliste cl_dupliquer curseur (curliste cl);
void cl_detruire curseur (curliste cl);

```

et on modifie structure de données ainsi :

```

struct curliste

```

```

{
    maillon curseur;
    struct liste *l; // structure partagée
}

struct liste
{
    struct maillon la_liste;
    curliste **les curseurs; // tableau de pointeurs de curliste
    size_t nb curseurs;
    size_t cardinal;
};

```

La création d'un curseur consiste à dupliquer la structure curseur et à insérer le nouveau curseur dans le tableau des curseurs. On balaye ce tableau pour *déplacer* les curseurs pointant sur le maillon supprimé.

La destruction du dernier curseur pose problème car il faut alors détruire proprement la liste... dans ce cas il est bon de lever une exception.