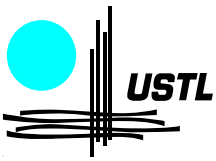


Licence d'informatique
Module de Pratique du C

Travaux dirigés

Manipulation de références et pointeurs

Philippe MARQUET

Novembre 2004

Les exercices proposés illustrent l'utilisation de références et de pointeurs. La représentation des tableaux à plusieurs dimensions est détaillée. On introduit les différentes classes d'allocation mémoire, dont l'allocation dynamique de mémoire. On traite aussi de la définition de types et de la manipulation de structures autoréférentes.

Exercice 1 (Passage de paramètres)
Écrivez une fonction qui échange les valeurs de deux variables entières.

(• ***Éléments de solution 1***

- On explique qu'une fonction de prototype

```
void will_no_swap(int a, int b);
```

  ne peut rien faire, si ce n'est des effets de bords. Les paramètres sont passés par valeur, donc copiés de l'appelant à l'appelé.
- On passe donc à une fonction telle la suivante

```
void
swap_int(int *pa, int *pb)
{
    int c = *pa ;
    *pa = *pb ;
    *pb = c ;
}
```

  en passant, par valeur, une référence sur une valeur entière, on peut modifier cette valeur entière.
- On utilisera cette fonction comme ceci :

```
int a, b;

swap_int(&a, &b);
```

•)

Exercice 2 (Copie de tableau/chaîne de caractères, allocation dynamique de mémoire)
Question 2.1 Donnez le code C de la fonction, non normalisée ANSI, de la bibliothèque `string`

```
char *strdup(const char *str);
```

qui retourne une copie fraîchement allouée de la chaîne de caractère `str`.

On fera appel à la fonction de bibliothèque

```
#include <stdlib.h>
void *malloc(size_t size);
```

qui renvoie une référence sur une nouvelle zone mémoire de longueur de `size` octets. On pourra aussi utiliser les autres fonctions de la bibliothèque `string`.

Que retourne cette fonction `strdup()` en cas de problème ?

(• ***Éléments de solution 2.1***

- La fonction n'est pas ANSI (le ISO C89 que le compilateur leur affiche de temps à autre...) ; elle n'est donc pas fournie si on compile avec `gcc -ansi` ; ce que l'on fait pourtant systématiquement pour assurer une certaine portabilité à notre code C.

– Le `const` assure que les valeurs référencées par `str` ne seront pas modifiées.
– Le type `size_t` est le type entier utilisé par les bibliothèques standard pour mémoriser des tailles, par exemple la différence entre deux pointeurs ou la longueur d'un tableau. On a peut être jamais parlé de définition de type, mais...
– La fonction `strdup()` peut retourner une erreur en cas d'impossibilité d'allouer de la mémoire. On retourne alors `NULL`, valeur illégale pour une référence.
– Le code, le code !

```
char *
strdup(const char *str)
{
    size_t siz;
    char *copy;

    siz = strlen(str) + 1;          /* +1 pour le \0 */
    if ((copy = malloc(siz)) == NULL)
        return NULL;
    memcpy(copy, str, siz);
    return copy;
}
```

– Il n'y a plus de raison depuis longtemps (ANSI C) de voir des coercitions (aussi nommé conversion ou cast) de la valeur retournée par `malloc()` telle : `(type_t *) malloc(..)` , merci `void *`
– On peut préférer `strncpy()` à `memcpy()`, mais on évitera `bcopy()` qui n'est pas normalisée ANSI...
– Une petite explication sur l'utilisation de `strdup()` est nécessaire. La valeur retournée pourra/devra être utilisée comme argument de `free()`.

•)

Question 2.2 Écrivez une fonction qui renvoie une copie « fraîche » d'un tableau de `n` entiers passé en paramètre.

(• ***Éléments de solution 2.2***

– On se souvient de la manière de désigner un tableau de taille variable.
– C'est du même acabit que `strdup()`... on note la nécessaire utilisation de l'opérateur `sizeof`.
– Une version tout à la main

```
int *
itabcpy(const int *tab, int n)
{
    int *res;

    res = malloc(n*sizeof(int));
    if (res) {
        int *p = res ;
        while (n--)
            *p++ = *tab++;
    }
    return res;
}
```

moins efficace à coup sûr que

```
memcpy(res,tab,n*sizeof(int));
```

•)

Exercice 3 (Noms temporaires, allocations automatique, statique ou dynamique)
Donnez la définition d'une fonction qui retourne à chaque invocation une chaîne de caractères différente, par exemple en vue de l'utiliser comme un nom de fichier temporaire.

(• ***Éléments de solution 3***

– On veut par exemple retourner `temp0`, puis `temp1`, `temp2`, etc. Pour simplifier, le suffixe reste sur deux chiffres maximum, puis rebouclage...
– Étant donné le prototype de la fonction

```
char *newname(void);
```

se pose de suite le problème de l'allocation du résultat : allocation statique ou dynamique ?

– Voyons d'abord les autres points. Dans tous les cas on aura quelque chose du genre

```
char *
newname(void)
{
    .... char name ...;      /* allocation dependant */
    static int sequence = -1;

    if (++sequence > 99)
        sequence = 0;
    strcpy(name, "tmp");
    name[3] = (sequence/10) + '0';
    name[4] = (sequence%10) + '0';
    name[5] = '\0';

    return name;
}
```

– On note la déclaration `static` de `sequence`, variable dite rémanente.

– On peut légitimement préférer une utilisation de `sprintf()` :

```
sprintf(name, "tmp%02d", sequence);
```

– Si l'on réalise une **allocation automatique** de la chaîne `name` :

```
char *
newname(void)
{
    char name[6];
    ....

    return name;
}
```

on obtient, dans le meilleur des cas un résultat erroné, `name` étant allouée *automatiquement*, sur la pile, à l'entrée de la fonction, mais aussi désallouée *automatiquement* à la terminaison de la fonction. Le code

```
char *tmpstr;
tmpstr = newname();
```

affecte `tmpstr` d'une adresse sur la pile qui n'est plus valide au moment de l'affectation ; elle est « au dessus » de la pile !

– On passe donc à une **allocation statique** de la variable `name` :

```
char *
newname(void)
{
    static char name[6];
    ....

    return name;
}
```

qui résout le problème précédent. Cependant le comportement dans le cas d'utilisation suivant

```
char *tmpstr1, *tmpstr2;
tmpstr1 = newname();
tmpstr2 = newname();
```

ne sera pas celui attendu, les deux valeurs `tmpstr1` et `tmpstr2` étant égales...

– On en vient donc à l'**allocation dynamique** :

```
char *
newname(void)
{
```

```
        char *name = malloc(6);

        if (!name)
            return NULL;
        ....

        return name;
    }
```

à charge pour l'appelant de libérer la mémoire allouée par `newname()`.

- De nombreuses fonctions des bibliothèques standard utilisent cette allocation dynamique pour retourner leur résultat. Qui a dit « vivement le garbage collector en C » ?
- On notera finalement qu'une telle fonction normalisée ANSI existe dans la bibliothèque C standard : `tmpnam()`.

•)

Exercice 4 (Tableaux à plusieurs dimensions)
Soit la déclaration d'un tableau

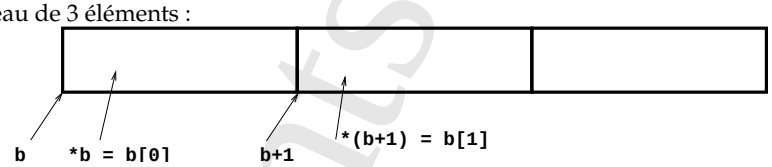
```
int b[3][5];
```

En considérant que l'allocation du tableau se fait linéairement en mémoire (les 3 « tranches » de `b` sont allouées à des adresses contiguës), donnez l'état du tableau `b` après l'exécution du code C suivant :

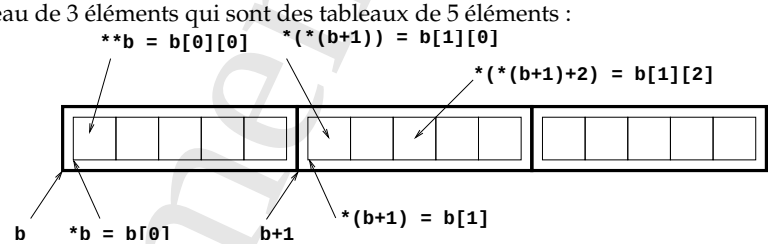
```
int b[3][5];
int *a = *b, i;
for (i=0 ; i<15 ; *a++ = i++)
    ;
**b = 15;          **(b+1) = 16;       *(b[0]+1) = 17;
*(*b+8) = 18;      *(b[1]+2) = 19;     *(*(b+1)+5) = 20;
*(b[2]+3) = 21;    *(*(b+2)+2) = 22;
```

(• ***Éléments de solution 4***

- Le petits dessins suivants expliquent l'allocation de `b`. En C, les tableaux sont rangées « ligne par ligne ».
- `b` est un tableau de 3 éléments :



- `b` est un tableau de 3 éléments qui sont des tableaux de 5 éléments :



- `a` est une référence sur le premier élément entier de `b` ;
- On découvre l'expression `*a++`, nouvel idiome du C qui retourne la valeur référencée par `p` puis incrémente la référence `p` ;
- La boucle produit donc la valeur suivante pour `b` :

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

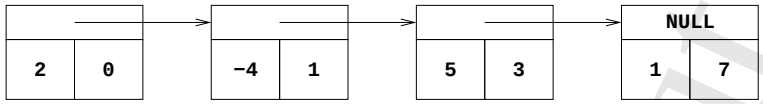
- On note ensuite que le `*` (déréférence) est prioritaire sur l'addition `+`. Donc `*p+i` est lu `*(p)+i`.
- L'arithmétique des pointeurs joue à plein : une incrémentation d'une référence considère le type de l'objet pointé
- La suite d'instructions produit finalement les valeurs :

```
{15, 17, 2, 3, 4, 16, 6, 19, 18, 9, 20, 11, 22, 21, 14}
```

•)

Exercice 5 (Manipulation de polynômes)
Un polynôme est une liste de monômes. Un monôme est caractérisé par un coefficient (considéré entier) et un degré. On ne rangera en mémoire que les monômes de coefficient différent de zéro.

Par exemple le polynôme $x^7 + 5x^3 - 4x + 2$ constitué de 4 monômes sera représenté par :



Question 5.1 Donnez des définitions des types de données `monome_t` et `polynome_t`.
(• ***Éléments de solution 5.1***

- Un monôme est caractérisé par un coefficient et un degré. On en fait une structure :

```
struct monome_s {
    int coeff;
    int degre;
};
```

- On définit alors simplement le type `monome_t`. La première utilisation de `typedef`. On note le nommage des types. J'utilise systématiquement le suffixe `_t`.

```
typedef struct monome_s monome_t;
```

- Je rencontre parfois des partisans du mélange :

```
typedef struct monome_s {
    int coeff;
    int degre;
} monome_t;
```

  je sais lire leur code, mais n'en produit point de semblable...
- Un polynôme est une liste chaînée de monômes. Le type `polynome_t` est une référence sur un chaînon :

```
typedef struct poly_s *polynome_t;
struct poly_s {
    monome_t mono;
    polynome_t next;
};
```

•)

Question 5.2 Écrivez une fonction qui incrémente le degré de tous les monômes d'un polynôme.
(• ***Éléments de solution 5.2*** Un parcours de liste. On note le caractère « calcul en place » de la fonction

```
void
inc_poly_r (polynome_t p)
{
    if (!p)
        return;
    p->mono.degre++;
    inc_poly_r(p->next);
}

void
inc_poly_i (polynome_t p)
{
    while (p) {
        p->mono.degre++;
        p = p->next;
```

```
    }
}
```

on note la première utilisation de l'opérateur `->`. •)

Question 5.3 Écrivez une fonction qui évalue un polynôme pour une valeur passée en paramètre.
(• ***Éléments de solution 5.3*** Un simple parcours de liste, simple en récursif :

```
#include <math.h>               /* pour pow() */

int
val_poly (polynome_t p, int x)
{
    if (p)
        return p->mono.coeff * pow(x, p->mono.degre)
            + val_poly(p->next, x);
    else
        return 0;
}
```

•)

Question 5.4 Écrivez une fonction qui imprime sur la sortie standard la représentation d'un polynôme.
(• ***Éléments de solution 5.4*** À nouveau un simple parcours de liste, facile en récursif si on affiche comme ça vient...

```
void
print_poly_r (polynome_t p)
{
    if (!p)
        return;
    /* le monome */
    printf("%dx^%d", p->mono.coeff, p->mono.degre);
    /* la suite */
    if (p->next)
        printf(" + ");
    print_poly_r(p->next);
}
void
print_poly (polynome_t p)
{
    print_poly_r(p);
    printf("\n");
}
```

•)

Question 5.5 Écrivez une fonction qui libère la mémoire occupée par un polynôme.
(• ***Éléments de solution 5.5***

- On suppose bien entendu que tous les maillons ont été alloués dynamiquement par `malloc()`
- Il s'agit juste de faire les choses dans le bon ordre ; on ne peut plus accéder à `p->next` après un `free(p)` :

```
void
free_poly (polynome_t p)
{
    if (!p)
        return;
    free_poly(p->next);
```

```
    free(p);
}
```

•)

Question 5.6 Écrivez une fonction qui lit sur l'entrée standard un polynôme donné sous la forme d'une suite de coefficients des monômes de degrés décroissants. Le premier entier entré représente le coefficient du monôme de degré le plus élevé ; le nombre de coefficients entrés est égal au degré du polynôme plus 1. Par exemple, le polynôme de l'exemple est entré par la suite :

```
1 0 0 0 5 0 -4 2
```

(• ***Éléments de solution 5.6***

- Création de monômes de degré zéro, que l'on accroche en tête du polynôme en cours de création dont on incrémente les degrés.

```
polynome_t
read_poly (void)
{
    polynome_t p = NULL;
    int coeff;
    int status;

    while ((status = scanf("%d", &coeff)) != EOF) {
        if (!status){ /* non int : skip */
            getchar();
            continue;
        }
        if (coeff) {
            polynome_t new = malloc(sizeof(struct poly_s));
            if (!new) {
                free_poly(p);
                return NULL;
            }
            new->mono.coeff = coeff;
            new->mono.degre = -1; /* va etre incremente */
            new->next = p;
            p = new;
        }
        inc_poly_r(p); /* incrementation de tous les degres */
    }
    return p;
}
```

- Oui c'est une création en $\mathcal{O}(n^2)$...
- On peut simplifier l'appel à `scanf()` en supposant que l'entrée fournie est correcte...

•)

Question 5.7 Écrivez une fonction qui additionne un monôme à un polynôme. Attention, on ne stocke jamais les monômes dont le coefficient est nul.

(• ***Éléments de solution 5.7***

```
<ton code ici>
```

•)

Question 5.8 Soyez rassurés, on ne vous demande pas de donner la définition d'une fonction qui additionne (ou multiplie !) deux polynômes. Donnez simplement les prototypes de telles fonctions.

(• ***Éléments de solution 5.8***

•)

Exercice 6 (Gestion de piles)
Il s'agit de donner une implantation des fonctions suivantes de manipulation d'une pile d'entiers

```
int is_empty(pile_t p);
int push(pile_t *pp, int val);
int pop(pile_t *pp, int *val);
```

Chacune de ces fonctions retourne une valeur non nulle en cas d'erreur.

La spécification précise que la taille de la pile n'est pas bornée, aussi utilisera-t-on une allocation dynamique de mémoire.

Question 6.1
- Donnez la définition d'un type `pile_t`.
- Précisez la représentation de la pile vide.
- Expliquez les prototypes donnés des fonctions.

(• ***Éléments de solution 6.1***
- Une pile est représentée par une liste chaînée de maillons. Une valeur de type pile est une référence vers un maillon. La pile vide est donc représentée par le pointeur `NULL`.
- J'écris, et dans cette ordre :

```
typedef struct elem_s *pile_t;
struct elem_s {
    int val;
    pile_t next;
};
```

- Je fais remarquer aux partisans du mélange

```
typedef struct elem_s {
    int val
    struct elem_s *next;
} *pile_t;
```

  que l'utilisation du type `pile_t` (i.e. le `.h`) ne requière que la définition du type :

```
typedef struct elem_s *pile_t;
```

  le reste pouvant rester cacher dans mon implémentation (le `.c`).
- Une macro pour la pile vide

```
#define EMPTY_PILE ((pile_t) 0)
```

- Et les valeurs pour les retours des fonctions :

```
#define RETURN_FAILURE  (-1)
#define RETURN_SUCCESS  (0)
#if RETURN_FAILURE >= 0
#  error "RETURN_FAILURE must be negative"
#endif
```

•)

Question 6.2 Donnez les définitions des fonctions `is_empty()`, `push()`, et `pop()`.
(• ***Éléments de solution 6.2*** Qu'est ce que vous voulez de plus que le code ? Un dessin ou mieux une animation avec les allocations, modifications des chaînages... Bon voila le code !

```
int
is_empty(pile_t p)
{
    return p == EMPTY_PILE;
}

int
push(pile_t *pp, int val)
```

```
{
    pile_t new;

    /* allocation */
    new = malloc(sizeof(struct elem_s));
    if (!new)
        return RETURN_FAILURE;

    /* initialisation */
    new->val = val;
    new->next = *pp;

    /* attachement en tete */
    *pp = new;

    /* done */
    return RETURN_SUCCESS;
}

int
pop(pile_t *pp, int *val)
{
    pile_t first;

    if (is_empty(*pp))
        return RETURN_FAILURE;

    /* le premier element */
    first = *pp;

    /* positionnement des valeurs retournées */
    *val = first->val;
    *pp = first->next;

    /* liberation du premier element */
    free(first);

    /* done */
    return RETURN_SUCCESS;
}
```

•)

Question 6.3 Identifiez les fonctions supplémentaires que requière l'utilisation de valeurs de type `pile_t`. Donnez le fichier d'entête d'une bibliothèque de manipulation de piles.

(• ***Éléments de solution 6.3***

- Il faut des fonctions de création/initialisation et de destruction de piles.
- Une version simple peut être :

```
#ifndef _PILE_H_
#define _PILE_H_

typedef struct elem_s *pile_t;

#define EMPTY_PILE ((pile_t) 0)

extern void delete(pile_t p);

extern int is_empty(pile_t p);
extern int push(pile_t *pp, int val);
```

```
extern int pop(pile_t *pp, int *val);

#endif
```

•)

Question 6.4 Soit la fonction `next_token()` suivante

```
/* next_token() retourne un token et positionne val en cas de token INT_TK */
enum token_e {INT_TK,           /* un entier */
              PRINT_TK,         /* affichage */
              ADD_TK,           /* addition  */
              MUL_TK,           /* produit */
              EOF_TK,           /* EOF */
              NONE_TK};         /* erreur */
static int val;
enum token_e next_token(void);
```

Écrivez un programme pour une simple calculette postfixée munie des opérations d'addition et de multiplication.

(• ***Éléments de solution 6.4***

– Juste au cas où, voici une implantation possible de `next_token()`. On ne demande pas d'écrire ce code.

```
enum token_e
next_token(void)
{
    switch (scanf("%d", &val)) {
        case EOF :
            return EOF_TK;
        case 1:
            return INT_TK;
        default: /* non int = print ! */
            getchar(); /* skip the non int */
            return PRINT_TK;
    }
    return NONE_TK;
}
```

– Il s'agit d'une simple illustration de l'utilisation des fonctions définies

```
#include "pile.h"

int
main()
{
    pile_t p = EMPTY_PILE;
    enum token_e tk;
    int i,j;

    while ((tk = next_token()) != EOF_TK) {
        switch (tk) {
            case INT_TK :
                if (push(&p, val)<0)
                    fatal();
                break;
            case PRINT_TK :
                if (pop(&p, &i))
                    fatal();
                printf("%d \n", i);
                break;
            case ADD_TK:        /* idem pour MUL_TK */
```

```
                if (pop(&p, &i))
                    fatal();
                if (pop(&p, &j))
                    fatal();
                if (push(&p, i+j)<0)
                    fatal();
                break;
            default:
                fatal();
        }
    }

    exit(EXIT_SUCCESS);
}
```

•)

Exercice 7 (Tableaux grandissants)
Il s'agit de gérer des tableaux, définis comme une structure de données avec des accès indexés à partir de zéro aux données, dont la taille augmente avec les accès aux éléments grandissants.

L'interface de cette structure de données est, par exemple pour des grands tableaux d'entiers :

```
int ga_set(struct ga_s *ga, unsigned int index, int val);
int ga_get(struct ga_s *ga, unsigned int index, int *val);
struct ga_s ga_init(void);
int ga_destroy(struct ga_s *ga);
```

Question 7.1 Donnez la définition de l'implantation d'une telle structure par un tableau alloué dynamiquement dont on conserve la taille.

Quels sont les avantages et inconvénients d'une telle implantation ?

(• ***Éléments de solution 7.1***

- De manière générale pour l'exercice, il s'agit donc de mémoriser au sein d'une structure la taille maximale actuelle et une référence vers une zone de mémoire allouée dynamiquement dont la taille pourra varier en fonction des accès en écriture.
- Une solution simple peut reposer sur l'utilisation de `realloc()`
- Avantage : accès direct. Inconvénient : peut nécessiter des recopies (opérées par `realloc()`) lors de l'agrandissement de la structure.

```
#include <stdlib.h>

void * realloc(void *ptr, size_t size);
```

dont il est précisé qu'en cas d'erreur elle retourne `NULL`. , mais que la mémoire référencée par `ptr` est toujours valide.

```
enum {GA_OK = 0,
      GA_ENOMEM,              /* can not allocate memory */
      GA_EOOB,                /* access out of bound */
      GA_ENULL,               /* invalid NULL pointer */
      GA_ELAST
};

#define GA_INITIAL_CHUNK_SIZE 8

/* malloc_good_size() is not ANSI */
#define malloc_good_size(size) (size)

struct ga_s {
    unsigned int ga_size;
    int *ga_array;
};
```

```
struct ga_s
ga_init(void)
{
    struct ga_s ga = {0,(int *) 0};

    return ga;
}

int
ga_destroy(struct ga_s *ga)
{
    if (!ga)
        return GA_ENULL;
    if (! ga->ga_size)
        free(ga->ga_array);
    return GA_OK;
}

int
ga_get(struct ga_s *ga, unsigned int index, int *val)
{
    if (!ga || !val)
        return GA_ENULL;
    if (index >= ga->ga_size)
        return GA_EOOB;

    *val = ga->ga_array[index];

    return GA_OK;
}

int
ga_set(struct ga_s *ga, unsigned int index, int val)
{
    if (!ga)
        return GA_ENULL;

    if (ga->ga_size == 0) {     /* first allocation */
        int size = malloc_good_size(max(GA_INITIAL_CHUNK_SIZE, index));
        ga->ga_array = malloc(size * sizeof(int));
        if (!ga->ga_array)
            return GA_ENOMEM;
        ga->ga_size = size;
    }

    if (index >= ga->ga_size) { /* extended allocation */
        int size = malloc_good_size(max(2 * ga->ga_size, index));
        int *new = realloc(ga->ga_array, size * sizeof(int));
        if (!new)
            return GA_ENOMEM;
        ga->ga_array = new;
        ga->ga_size = size;
    }

    /* set the ga element */
    ga->ga_array[index] = val;
```

```
    return GA_OK;
}
```

il est possible de raffiner le traitement des erreurs de débordement en mémorisant non seulement la taille de la zone d'allocation, mais aussi le plus grand index des éléments écrits

```
struct ga_s {
    unsigned int ga_size;
    unsigned int ga_upper_bound;
    int *ga_array;
};
```

•)

Question 7.2 Les inconvénients de la première implantation peuvent être supprimés en multipliant les zones mémoires nécessaires au fur et à mesure de l'agrandissement de la structure. Il s'agit alors de définir une structure de données pour regrouper ces zones mémoires. Donnez la définition d'une nouvelle implantation basée sur ce principe.

(• ***Éléments de solution 7.2***

- Il s'agit de supprimer les éventuelles copies lors d'agrandissement de la structure.
- Une solution est de chaîner entre-eux plusieurs « chunk » d'éléments alloués au fur et à mesure :

```
#define GA_CHUNK_SIZE
struct ga_chunk_s {
    struct ga_chunk_s ga_next;
    int ga_array[GA_CHUNK-SIZE];
};
struct ga_s {
    unsigned int ga_size;
    struct ga_chunk_s ga_first_chunk;
};
```

  on perd bien entendu l'accès direct aux éléments...
- Une autre proposition est de mémoriser dans un tableau alloué dynamiquement les adresses de chunks alloués au fur et à mesure. Seul ce tableau de pointeurs vers des chunks doit être réalloué de temps. S'il subsiste de possibles recopies dues aux réallocations, elle ne concernent pas les valeurs du tableau, mais les seuls références de chunks. On note que les chunk non encore utilisés (jamais écrits) ne sont pas alloués, on a donc une structure creuse.

•)

Exercice 8 (Allocation via un pool)

Plutôt que d'allouer dynamiquement au fur et à mesure de nombreuses petites structures, on peut gérer, au niveau applicatif, un pool de structures. Les requêtes d'allocation sont alors satisfaites en retournant des structures de ce pool, les désallocations venant à nouveau nourrir le pool, une allocation dynamique effective n'ayant lieu que lorsque le pool est vide.

Proposez une interface et son implantation pour la gestion d'un pool de valeurs d'un type `struct_t` connu.

(• ***Éléments de solution 8***

- L'interface comprend deux fonctions centrales d'allocation/désallocation. Des fonctions de création/destruction du pool sont aussi nécessaires. Si on ne veut pas mentionner le pool en paramètre aux appels d'allocation/désallocation, on utilisera une variable statique dans la bibliothèque.

```
struct_t *new_struct(void);
void free_struct(struct_t *s);

void create_pool(void);
void destroy_pool(void);
```

- On peut bien entendu gérer le pool sous la forme d'une bête liste chaînée d'objets.
- Le message à faire passer est l'utilisation des `struct_t` au sein du pool comme des maillons de notre liste chaînée.
- L'utilisation d'une `union` serait possible. On utilise ici une simple coercition de pointeurs.

– On définit un type pour nos maillons de chaînage :

```
struct pool_s {
    struct pool_s *next;
};
```

il faut s'assurer que le type `struct_t` est de taille suffisante pour stocker un `struct pool_s`. Ce ne peut qu'être le cas si `struct_t` est le maillon d'une structure chaînée....
On ne peut malheureusement pas écrire

```
#if sizeof(struct_t) < sizeof(struct pool_t)
#  error "UNABLE TO POOL PICOLO..."
#endif
```

et c'est justifié par le fait que ceci serait évalué lors de la compilation, sur la machine locale, et poserait des problèmes lors de cross-compilation... On se contente donc de

```
if (sizeof(struct_t) < sizeof(struct pool_s))
    fatal("Unable to pool picolo...");
```

– La création du pool ne fait qu'initialiser à `NULL` la variable statique `pool` :

```
static struct pool_s *pool;

void
create_pool(void)
{
    if (sizeof(struct_t) < sizeof(struct pool_s))
        fatal("Unable to pool picolo...");

    pool = (struct pool_s *)0;
}
```

– La destruction consiste en une réelle désallocation de notre liste chaînée.

```
void
destroy_pool(void)
{
    struct pool_s p;

    while (pool) {
        p = pool->next;
        free(pool);
        pool = p;
    }
}
```

Attention, on ne peut écrire

```
    while (pool) {
        p = pool;
        free(pool);
        pool = p->next;
    }
```

ou autre code accédant à la mémoire suite à sa restitution...

– Dans l'allocation, le point central est le fait que les appels nécessaires à `malloc()` se font en vue d'allouer des `struct_t` et non des `struct pool_t` :

```
struct_t *
new_struct(void)
{
    struct_t *res;
    if (pool) {
        /* return the first pool_s */
        res = (struct_t *) pool;
        pool = pool->next;
        return res;
```

```
    } else {
        /* no more pool_s, mallocate one */
        return malloc(sizeof(struct_t));
    }
}
```

– La désallocation chaîne en tête le `struct_t` donné :

```
void
free_struct(struct_t *s)
{
    ((struct pool_s *)s)->next = pool;
    pool = (struct pool_s *)s;
}
```

toutes les parenthèses sont nécessaires dans `((struct pool_s *)s)->next`

– On peut aussi allouer, lors des appels à `malloc()`, des blocs de `struct_t` et ne rendre par `new_struct()` ces `struct_t` un à un. On gère alors une structure chaînée de blocs de `struct_t` en mentionnant dans chacun des blocs le nombre de `struct_t` libres. Lors de la désallocation, on ne se préoccupe pas de recoller les `struct_t` adjacents en blocs :

```
struct pool_s {
    unsigned int n_struct_t;
    struct pool_s *next;
};

void
free_struct(struct_t *s)
{
    ((struct pool_s *)s)->next = pool;
    ((struct pool_s *)s)->n_struct_t = 1;
    pool = (struct pool_s *)s;
}

#define N_STRUCT_PER_BLOC ((1<<12)/sizeof(struct_t))

struct_t *
new_struct(void)
{
    struct_t *res;
    if (!pool) { /* no more pool_s, mallocate one bloc */
        pool = malloc(sizeof(struct_t) * N_STRUCT_PER_BLOC);
        if (!pool)
            return (struct_t *)0;
        /* initialization */
        pool->n_struct_t = N_STRUCT_PER_BLOC;
        pool->next = (struct pool_s *)0;
    }

    if (pool->n_struct_t ==1) { /* the last struct_t of the bloc */
        res = (struct_t *) pool;
        pool = pool->next;
        return res;
    } else { /* one less struct_t in the bloc */
#ifdef RETURN_THE_FIRST_STRUCT_T
        res = (struct_t *) pool;
        /* adjust pool */
        (struct_t *)pool++;
        /* update new bloc info */
        pool->n_struct_t = ((struct pool_s*) res)->n_struct_t - 1;
        pool->next = ((struct pool_s*) res)->next;
#else
```

```
        /* adjust pool */
        pool->n_struct_t--;
        /* the last struct_t */
        res = ((struct_t *)pool) + pool->n_struct_t;
#endif
        return res;
    }
}
```

•)