CHAPITRE 1

INTRODUCTION À LA SPÉCIFICATION ET RAPPEL DE PROGRAMMATION

Pour introduire la méthode de résolution de problèmes en informatique, nous nous appuyons sur un exemple qui va nous servir tout au long du chapitre.

1 SPÉCIFICATIONS

1.1 Un (petit) exemple

Énoncé. Illustrons la démarche de résolution des problèmes que nous voulons mettre en place sur la définition de la fonction donnant x^n pour x rationnel et n entier naturel. Plutôt que de programmer l'opération en langage C, nous allons d'abord la spécifier en raffinant la spécification : tout d'abord par rapport aux opérateurs de base, puis de manière récursive directe et enfin de manière itérative pour préparer l'implantation.

Spécification formelle.
profil
puiss : Rat Nat → Rat /* fonction puissance */
précondition n : Nat ; x : Rat /* éliminer l'indéterminée 0^0 */
pré puiss(x, n) = x !=0 ou n !=0
définition par rapport aux constructeurs des entiers 0 et ()+1)
puiss(x, 0) = 1
puiss(x, n+1) = x * puiss(x, n)
définition directe *(3 multiplications pour calculer x^3)= x * x * x * 1)*
puiss(x, n) = si n == 0 alors 1 sinon x * puiss(x, n-1) fsi
définition itérative 1 *(3 multiplications pour calculer x^3= 1* x * x * x de n à 1)*
puiss(x, n) = r avec (i,r) = init (n, 1) ttq i != 0 rép (i -1, r * x) frép
définition itérative 2 *(calcul de x^3= 1 * x * x * x de 0 à n-1)*
puiss(x, n) = r avec (i,r) = init (0, 1) ttq i != n rép (i +1, r * x) frép

En C, nous pouvons écrire toutes ces versions calquer sur les spécifications :

```
Rat puiss(Rat x, Nat n) /* version récursive */
{
      return n == 0 ? 1 : x * puiss(x, (n - 1));
}
Rat puiss(Rat x, Nat n) /* version itérative 1 */
{
      Rat r; Nat i;
      i = n; r = 1;
      while(i != 0)
      {
            i = (i - 1); r = r * x;
      }
      return r;
}/*
Rat puiss(Rat x, Nat n) /* version itérative 2 */
{
      Rat r; Nat i;
      i = 0; r = 1;
      while(i != n)
      {
            i = (i + 1); r = r * x;
      }
      return r;
} ■
```

Nous ne procéderons pas systématiquement à l'écriture de toutes ses versions des spécifications mais c'est la démarche sous-jacente.

1.2 Spécification des ensembles

Nous supposons que, dans le module de spécification `BASE`, figure entre autres des spécifications des booléens et des entiers naturels, avec leurs opérateurs et propriétés habituels.

Exemple (*Spécification d'ensembles*) Une spécification `ENS0` des ensembles finis d'entiers naturels tous distincts (sinon ce sont des multi-ensembles) dont le cardinal est borné par 100, c'est-à-dire ne contenant pas plus de 100 éléments, peut s'exprimer comme suit.

spéc ENS0 **étend** BASE
sorte Ens
opérations
∅ : → Ens /* constante ensemble vide */
insertion : Ens Nat → Ens /* i ou insertion d'un élément */
suppression : Ens Nat → Ens/* s ou suppression d'un élément */
_ ϵ_ : Nat Ens → Bool /* appartenance à un ensemble */
vide : Ens → Bool /* test de vacuité d'un ensemble */
|_|: Ens → Nat /* cardinal d'un ensemble */
minimum : Ens → Nat /* plus petit élément */
préconditions e : Ens ; x : Nat
pré insertion(e, x) = ⅂ x ϵ e et |e|< 100
pré suppression(e, x) = x ϵ e
pré minimum(e) = ⅂ vide(e)
axiomes e : Ens ; x, y : Nat
(e0) insertion(insertion(e, x), y) = insertion(insertion(e, y), x) /*
permutativité de i */
(e1) x ϵ ∅ = faux
(e2) x ϵ insertion(e, y) = x == y ou x ϵ e
(e3) vide(∅) = vrai
(e4) vide(i(e, x)) = faux
(e5) |∅| = 0
(e6) |insertion(e, x)| = |e| + 1
(e7) suppression(insertion(e, x), y) = **si** x == y **alors** e
sinon insertion(suppression(e, y), x) **fsi**
(e8) minimum(insertion(e, x)) = **si** v(e) **alors** x
sinon si x < minimum(e) **alors** x **sinon** minimum(e) **fsi fsi**
ou bien en séparant en 2 axiomes le cas v(e) **ou pas :**
(e8) minimum(insertion(∅, x)) = x
(e9) minimum(insertion(insertion(e, x), y)) =
si minimum(insertion(e,x)) < y
alors minimum(insertion(e,x)) **sinon** y **fsi**
fspéc

Mettre au clair le fait que dans insertion(e, x), suppression(e, x), minimum(e), e est une variable muette. Par exemple, la précondition sur suppression(e, x) s'applique sur e qui peut ensuite être remplacé par insertion(e, x) dans l'axiome (e7). La notation e =insertion(e', x) peut éclaircir ...
Pour la programmation nous allons respecter les profils et les préconditions des ensembles mais sans procéder au raffinement sytématique du récursif à l'itératif.

2 PROGRAMMATION

2.1 Utilisation de types prédéfinis

`Booléens ou Bool`

Le langage C possède le type booléen `_Bool`. Celui-ci peut aussi être simulé, par exemple par les entiers naturels du type `unsigned char` (8 bits : 0 à 255) en **déclarant un nouveau type** :

```
typedef unsigned char Bool;
```

Les symboles de constantes faux et vrai sont alors représentés par `0` et `1`. Pour garder les notations, il est commode de définir deux constantes nouvelles. Elles peuvent l'être par des *fonctions*, mais plus efficacement par des *macro-définitions* **(#)**, ou *macros* tout court, du langage C :

```
#define faux 0
#define vrai 1
```

Il s'agit donc d'une implantation du système canonique {vrai, faux} des booléens. Variante possible correspondant au type booléen en C :

```
#define faux 0
#define vrai (!faux)
```

`Entiers naturels ou Nat`

En langage C le type entiers naturels que nous noterons *`Nat`* peut être interpréter comme le type `unsigned int`, tout en sachant que cela peut poser des problèmes pour les grands entiers à cause du risque de débordement. Prendre un type d'entiers plus longs, comme `unsigned long`, reporterait le problème sans le supprimer. Il suffit de déclarer :

```
typedef unsigned int Nat;
```

De la même manière, nous déclarons pour les entiers relatifs, les rationnels, les réels et les caractères :

```
typedef int Ent;
```

```
typedef float Rat, Reel;
typedef char Car;
```

Notons que les *réels* des langages de programmation sont en réalité des rationnels. Le langage C permet bien sûr les notations de constantes numériques comme `0`, `1`, `2`, ... , `100`, etc., ou caractères comme `'a'`, `'b'`, `'?'`, etc. Nous les utilisons par la suite en supposant connu le noyau de types et opérations *prédéfinis* de C et de ses bibliothèques standard. ∎

Sans même parler de spécification, il est intéressant dans un programme C de *renommer* grâce à `typedef` ou à `#define` tout type prédéfini utilisé. En effet, ceci permet d'en changer facilement, par une simple réécriture du `typedef` ou `#define`, c'est-à-dire sans avoir à parcourir tout le programme pour modifier individuellement chaque déclaration. Rien d'autre ne devra en principe être ré-écrit, mais le programme devra être recompilé. L'ensemble de ces types prédéfinis sera défini dans un fichier nommé `base.h`.

2.2 Renommage des types de base

Au module de spécification BASE correspond un fichier C appelé `base.h`. Ce fichier peut être complété progressivement avec de nouvelles notions générales. Au stade où nous sommes, son contenu minimum peut être par exemple le suivant :

```
#include <stdlib.h>
#include <stdio.h>
#define vrai (0==0)
#define faux (!vrai)
typedef unsigned char Bool;
typedef unsigned int Nat;
typedef int Ent;
typedef float Reel;
typedef float Rat;
typedef char Car;
typedef Car * Chaine;

/* Macros définies plus tard */
#define SIZEOF(x)       ((Nat)(sizeof(x)))
#define MALLOC(type)    ((type*)malloc(sizeof(type)))
#define MALLOCN(type, n) ((type*)malloc(n * sizeof(type)))
```

```
#define CALLOCN(type, n) ((type*)calloc(n, sizeof(type)))
#define REALLOC(t, type, n) ((type*)realloc(t, n * sizeof(type)))
#define FREE(t) free(t)
```

Ce renommage des types permet par exemple de passer des `float` au `double` sans rien changer dans l'écriture des fonctions. Ainsi, le texte de ce fichier est inclus et réutilisé dans n'importe quel autre qui contient la directive :

```
#include "base.h"
```

Une alternative à la définition de types est l'écriture de macros. Par exemple, pour les caractères et les chaînes on peut écrire à la place des `typedef` :

```
#define Car (char)
#define Chaine (Car*)
```

Enfin, pour implanter ses spécifications, le programmeur doit respecter scrupuleusement les définitions de types et opérations du langage C. Comme toujours, il faut prendre garde qu'une chaîne C se termine toujours par le caractère supplémentaire '\0". Rappelons que en C les caractères sont codés comme des entiers en C, de 65 à 97 du 'A' au 'a' et de 48 à 57 pour les chiffres '0' à '9'. Ainsi pour transformer le caractère c = '0' en l'entier 0, il suffit d'effectuer l'opération c – 48 ou c – '0' et pour transformer l'entier x en caractère, un cast suffit (car) x.

2.3 Structure de données

Dans le cas d'objets compliqués, il faut fabriquer soi-même un ou plusieurs types de données adaptés. Il faut faire appel par exemple à des tableaux, des structures et des pointeurs.

Exemple Nous prenons des ensembles d'au plus 100 entiers naturels. Nous retenons une implantation avec des tableaux. Nous décidons de représenter en langage C un ensemble :

- soit comme une *structure* composée d'un tableau `v` de `100` entiers et d'un entier `n`. Un type de structures `struct strens`, noté aussi `Ens`, est alors défini (Fig. 1) :

```
typedef struct strens {Nat v[100]; Nat n;} Ens;
```

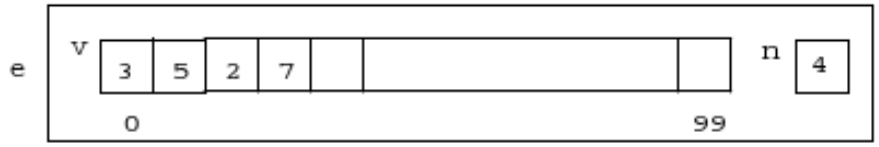

Figure 1 : Un ensemble e représenté par une structure

La valeur de `n` donne le cardinal de l'ensemble dont les éléments sont rangés dans les cases indicées de `0` à `n - 1` du tableau `v`. Une telle représentation est directe, mais nous verrons qu'elle oriente plutôt vers des passages de paramètres *par valeurs*, peu efficaces pour des objets de grande taille ;

- soit comme un *pointeur* sur une structure comme la précédente (Fig. 2) :

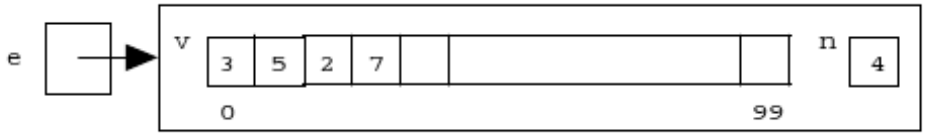

Figure 2 : Un ensemble e représenté par un pointeur sur une structure ■

Les pointeurs servent entre autres à rendre plus efficaces les passages de paramètres des fonctions, en les simulant *par adresses*.

2.4 Programmation fonctionnelle

Nous programmons tout d'abord les opérations en langage C sous forme de *fonctions* laissant intacts leurs arguments après un appel, elles se comportent exactement comme des fonctions mathématiques.

Structures (sans pointeurs)

La faculté des compilateurs C de permettre la transmission de paramètres-structures par valeurs est utilisée ici. Ceci permet une programmation directe agréable, mais pas toujours efficace.

Exemple Nous prenons les ensembles représentés par :

```
typedef struct strens {Nat v[100]; Nat n;} Ens;
```

Nous programmons les opérations correspondant à cette implantation. Nous nous plaçons dans l'hypothèse où tout appel d'opération n'a lieu que lorsque la précondition correspondante est vérifiée. Les opérations `ensvide` et `i` s'écrivent :

```
Ens ensvide()     /* ensemble vide */
{
      Ens e;      /* e:nouvel ensemble */
      e.n = 0;    /* cardinal à 0 */
      return e;   /* renvoi du resultat e */
}

Ens insertion(Ens e, Nat x) /* insertion de x dans e */
{
      Ens e1 = e;       /* e1:nouvel ensemble */
      e1.n = e.n + 1;   /* cardinal de e1 */
      e1.v[e.n] = x;    /* insertion effective *
      return e1;        /* renvoi du resultat e1 */
}
```

A cause du passage de paramètre *par valeur*, la recopie de l'ensemble est automatique lors de l'appel de la fonction, il est donc inutile de définir `e1`.

```
Ens insertion(Ens e, Nat x)/* insertion de x dans e */
{
      e.v[e.n] = x;     /* insertion effective */
      e.n++;            /* nouveau cardinal */
      return e;
}
```

A noter que le passage *par valeur* de la structure `e` à chaque appel récursif est lourd mais nécessaire pour une programmation fonctionnelle où les ensembles sont tous conservés.

Nous donnons maintenant trois formes de la programmation de `app`, obtenues par *transformations* successives. Les deux premières utilisent une fonction récursive, et la dernière est itérative. Cette manière de faire correspond à un raffinement de la fonction.

Cette première forme de `app` est écrite directement de manière *récursive* (en calquant la spécification) :

```
Bool app(Ens e, Nat x)  /* forme récursive */
{
      if (e.n == 0) return faux;    /* cas e vide */
      else  if (e.v[e.n - 1] == x) return vrai;
                  else {e.n--; return app(e, x);}
}
```

En effet, comme le montre la structure formelle des termes ensemblistes, quand `e` n'est pas vide, il est obtenu par insertion d'un élément à la première position libre `e.n - 1` de `e.v`.

Il est désagréable de modifier `e` pour un simple parcours, même si cela reste interne à la fonction `app`. Une *fonction auxiliaire* `app1`, est définie avec un paramètre supplémentaire de position `k`, sur lequel opère la récursivité.

```
Bool app1(Ens e, Nat x, Nat k)/* fonction auxiliaire */
/* vrai ssi x est en position <= k-1 dans e.v */
{
      if (k == 0) return faux;
      else  if (e.v[k - 1] == x) return vrai;
                  else return app1(e, x, k - 1);
}

Bool app(Ens e, Nat x)  /* forme intermédiaire */
{
      return app1(e, x, e.n); /* appel auxiliaire */
}
```

Cette fonction auxiliaire avec son paramètre supplémentaire prépare également au passage de la forme récursive à la forme itérative. La forme itérative est ainsi obtenue en remplaçant `app1` par une *itération* :

```
Bool app(Ens e, Nat x)  /* forme itérative */
{
      Nat k;      /* pour parcours */
      for(k = e.n; 0 < k && e.v[k - 1] != x; k--);
      return k != 0;
}
```

La condition de continuation `0 < k && e.v[k - 1] != x` est correcte parce que `e.v[k - 1] != x` n'y est évalué que si `0 < k` est `vrai`, ce qui annule le risque de débordement du tableau `e.v` quand `k == 0`. Comme dans toute la suite du chapitre, nous avons choisi d'utiliser une boucle `for`, mais nous pourrions utiliser une boucle `while`. Notons enfin que nous pourrions supprimer `k - 1` redondant avec `k--`, mais peut-être au détriment de la clarté.

Le test de vacuité et le cardinal d'un ensemble sont obtenus par les fonctions `v` et `card`, cette dernière correspondant à l'opérateur |_| de la spécification :

```
Bool vide(Ens e)  /* test de vacuite de e */
{     return e.n == 0;}

Nat card(Ens e)   /* cardinal de e */
{     return e.n;}
```

Pour l'opération de `suppression` nous proposons directement la forme itérative suivante, où, pour éviter de créer des trous, le dernier élément de l'ensemble prend la place de l'élément supprimé :

```
Ens suppression(Ens e, Nat x) /* précondition x ∈ e */
{
      Nat k;            /* pour parcours */
      e.n = e.n - 1;    /* cardinal de e1 */
      for(k = e.n; e.v[k] != x; k--);
                        /* k:place de x dans e.v */
      e.v[k] = e.v[e.n];/* remplacement de x */
      return e;
}
```

Dans l'opération `s`, l'ancien ensemble `e` est passé par valeur, comme dans `i`, et n'est donc pas modifié à l'extérieur de `s`. L'élément `x` est recherché, localisé en `k`, et remplacé par le dernier élément de `e.v`, qui est éventuellement `x` lui-même. La précondition de `s` impose que `x` *soit toujours présent dans le tableau* `e.v`. Il est donc toujours trouvé.

Enfin, le `minimum` d'un ensemble non vide peut être programmé de différentes manières. Nous donnons juste une forme itérative :

```
Nat minimum(Ens e)      /* minimum de e non vide */
{
      Nat p = e.v[e.n - 1];   /* p:minimum courant */
      Nat k;                  /* pour parcours */
      for(k = e.n - 1; 0 < k;k--)
                              /* recherche du min p */
            if (e.v[k - 1] < p) p = e.v[k - 1];
      return p;
}
```

Pour éviter deux des trois calculs de `k - 1`, cette fonction peut être écrite :

```
Nat minimum(Ens e)      /* minimum de e non vide */
{
      Nat k = e.n - 1;  /* p:minimum courant */
      Nat p = e.v[k];   /* pour parcours */
      for(; 0 < k;)
            if (e.v[--k] < p) /* recherche du min p */
            p = e.v[k];
      return p;
}
```

Mais cette version est assez obscure parce que la notation `k` n'y a jamais la même signification. ∎

Pourquoi les préconditions sont-elles testées en-dehors des fonctions ?

Il est impératif est de tester les préconditions. Nous utilisons une fonction booléenne testant les préconditions, qui est appellée avant la fonction :

```
Bool precondition-insertion(Ens e, Nat x)
Ens insertion(Ens e, Nat x)

if precondition-insertion(e, x) insertion(e, x)
else printf(" insertion : precondition non satisfaite");
```

ainsi nous pouvons éviter le test lorsque nous savons par le contexte du programme qu'elles sont satisfaites.

2.5 Rappel sur les pointeurs

Simulation du passage par adresse

Nous cherchons à remplacer la transmission d'un paramètre structure par son adresse. En C, si une variable `x` est déclarée du type `t` par

```
t x;
```

alors, *l'adresse* de la zone de mémoire allouée à `x` pour y ranger une valeur du type `t` est notée `&x`. Ainsi, `x` en paramètre dans une opération peut être remplacé par `&x`, pour réaliser un passage de `x` par adresse. Si le résultat de l'opération doit être l'objet `x` qui a *muté*, alors, il est courant en C de faire de l'opération une fonction à résultat `void`, c'est-à-dire une *procédure*, au sens habituel des langages de programmation, avec un *effet de bord* sur `x`. Mais on peut aussi donner à ces opérations une forme fonctionnelle en déclarant

```
t* x;
```

où cette fois `x` est de type pointeur c'est-à-dire que son contenu sera une adresse, adresse d'une zone mémoire qu'il faudra crée par allocation dynamique.

Allocation dynamique

En langage C, la fonction `malloc(n)` permet *d'allouer* une zone de `n` caractères et de renvoyer un *pointeur* sur cette zone. Par ailleurs, l'opération `sizeof(t)` fournit le nombre d'octets d'une zone de mémoire nécessaire pour contenir un objet du type `t` fourni en paramètre.

```
int* x;
x = (int *) malloc(sizeof(int))
```

Pour allouer une zone de mémoire destinée à un objet de type `t`, il est désagréable de devoir appeler `malloc(sizeof(t))` en mentionnant la taille de la zone. Pour l'éviter, nous définissons une macro `MALLOC` :

```
#define MALLOC(t) ((t *) malloc(SIZEOF(t)))
```

On remarquera l'usage du *forceur* de type, ou *cast* `(t *)`, qui impose que le résultat de la macro soit du type `(t *)` et sur lequel nous reviendrons.

Exemple La déclaration et l'allocation d'un ensemble `e` représenté comme dans la deuxième version ci-dessus se font alors par :

```
Ens e = MALLOC(Strens); ∎
```

Pour être homogène et récupérer une taille dont le type est toujours compatible avec `malloc`, nous définissons la macro :

```
#define SIZEOF(t) ((Nat) sizeof(t))
```

Pour allouer une zone de `n` fois `sizeof(t)` octets destinée à contenir un tableau de `n` objets du type `t`, nous définissons de manière analogue deux macros, `CALLOC` et `MALLOCN` :

```
#define CALLOC(t, n) ((t *) calloc(n, sizeof(t)))
#define MALLOCN(t, n) ((t *) malloc(n * sizeof(t)))
```

Leur rôle est le même, mais la première initialise à `0` les octets de la zone allouée, ce que ne fait pas la deuxième.

Enfin, un pointeur qui ne repère aucune zone de mémoire allouée a traditionnellement une valeur d'*adresse fictive* désignée par `nil`, ou `NIL`. En C, cette adresse fictive est `0`, qui ne correspond à l'adresse d'aucun objet valide. Elle est désignée aussi par `null`, ou `NULL` grâce à la macro :

```
#define NULL (void *) 0
```

Une allocation qui échoue renvoie `NULL`. Il faut le tester systématiquement.

Le type `void` est un *type fictif* et `(void *)` un type d'*adresses génériques*, c'est-à-dire un type compatible avec tous les pointeurs C. La macro impose donc `0` à être de ce type. Elle figure normalement dans le fichier de bibliothèque `<stdio.h>`.

Pour *désallouer* une zone allouée comme précédemment et repérée par un pointeur `p`, il suffit en C d'appeler la fonction `free(p)`. Pour homogénéiser, nous notons en majuscules `FREE(p)` l'appel de la macro :

```
#define FREE(p) (free(p))
```

qui a pour effet de désallouer la zone repérée par `p`.

Exemple La désallocation de l'ensemble `e` alloué ci-dessus s'écrit :

```
FREE(e); ■
```

Il existe aussi un `REALLOC` pour changer la taille d'une zone déjà allouée (à éviter !):

```
MAN realloc:(void *) realloc(void *t, size)
#define REALLOC(t, type, n) ((type*)realloc(t, n * sizeof(type)))
t = REALLOC(t, type, n)
```

Une allocation qui échoue renvoie `NULL`. Il faut le tester systématiquement en particulier pour le `realloc` et il faut donc conserver l'adresse avant reallocation.

Dans toute la suite, les macros ci-dessus sont utilisées dans les programmes manipulant des pointeurs.

2.6 Programmation avec mutations

Des fonctions, allouant ou copiant à chaque appel des structures de données pour garder les valeurs de paramètres effectifs, sont rarement satisfaisantes. On paie en effet en temps et en espace cette conservation, alors qu'elle est inutile dans beaucoup d'applications.

Il vaut mieux alors accepter une *modification des paramètres effectifs*, on dit aussi une *mutation*, par ce que l'on appelle un *effet de bord*. Après l'appel d'une opération avec mutation, les anciennes valeurs des paramètres effectifs sont perdues. Il est agréable cependant de laisser à ces opérations un aspect de fonction, en signalant nettement l'effet de bord.

Mais là encore, plusieurs versions sont possibles selon qu'on travaille sur les structures directement ou par l'intermédiaire de pointeurs.

Pointeurs sur structures

Exemple Nous déclarons cette fois, en plus du type de structures `struct strens`, noté aussi `Strens`, et un type de pointeurs sur de telles structures, noté `Ens` :

```
typedef struct strens {Nat v[100]; Nat n;} Strens;
```

```
typedef struct strens {Nat v[100]; Nat n;} * Ens;
```

Dans cette déclaration le type ensemble `Ens` est cette fois un pointeur `*` sur la structure. Les deux déclarations peuvent être faites en une seule instruction :
```
typedef struct strens {Nat v[100]; Nat n;} Strens, *Ens;
```

L'opération `ensvide` crée par un `MALLOC` la nouvelle structures de données. Nous remplaçons les opérations `insertion` et `suppression`, par des opérations qui *modifient* l'ensemble passé en argument. Pour ce faire, nous utilisons l'*adresse* de la structure ensemble. Les autres opérations peuvent rester quasiment identiques à ce qu'elles étaient auparavant en remplacant l'opérateur . par `->` pour l'accès aux éléments de la structure `Strens`.

Toutes les opérations gardent une apparence de *fonctions* en C.

```
Ens ensvide()/* ensemble vide : mutation */
{
      Ens e;
      e = MALLOC(StrEns);
      e->n = 0;
      return e;
}

Ens insertion(Ens e, Nat x)
/* insertion de x dans e modifié : mutation */
{
      e->v[e->n] = x;
      (e->n)++;
      return e;
}

Ens s(Ens e, Nat x)
/* suppression de x de e modifié : mutation */
{
      Nat k;
      (e->n)--;
      for(k = e->n; e->v[k] != x; k--);
                  /* k est la place de x dans e.v */
      e->v[k] = e->v[e->n];
      return e;
} ■
```

Ces nouvelles opérations sont efficaces, rapides et économes en espace, avec l'avantage d'une notation fonctionnelle et, à l'extérieur des opérations de base, un camouflage des notations d'adresses et de pointeurs.

Notons que les langages de programmation fonctionnelle privilégient les implantations fonctionnelles, mais admettent aussi les mutations pour des raisons d'efficacité, en les mettant nettement en évidence par des déclarations appropriées.

****

Deux autres formes peuvent être données pour `ensvide`, `insertion et suppression` en conservant la structure de données sans pointeurs. Les autres opérations qui ne modifient pas l'ensemble, restent identiques.

Exemple Pour nos ensembles représentés directement par des structures avec : `typedef struct strens {Nat v[100]; Nat n;} Ens;`

La *forme* 1 dite forme *procédurale* est écrite avec des *procédures*, c'est-à-dire des fonctions C avec résultat du type `void` et effet de bord.

```
void ensvide(Ens* e)    /* ensemble vide : forme 1 */
{     e->n = 0;}

void insertion(Ens* e, Nat x)
/* insertion de x dans *e modifié : forme 1 */
{
      e->v[e->n] = x;
      (e->n)++;
}

void suppression(Ens* e, Nat x)
/* suppression de x de *e modifié : forme 1 */
{
      Nat k;
      (e->n)--;
      for(k = e->n; e->v[k] != x; k--);
            /* k :place de x dans e.v */
      e->v[k] = e->v[e->n];
}
```

Utilisons ces opérations sur un ensemble `e` déclaré comme suit :

```
Ens e;
```

Nous voulons par exemple successivement initialiser `e`, y insérer les éléments `3` et `7`, puis supprimer l'élément `3`. Alors, avec ces formes 1, nous appelons : `ensvide(&e); insertion(&e, 3); insertion(&e, 7); suppression (&e, 3);`
où `&e` désigne l'adresse de la zone de mémoire correspondant à l'identificateur `e`.

Ces appels de procédure étant peu pratique, nous proposons une ***forme* 2, dans laquelle nous laissons aux opérations un *aspect fonctionnel*,** avec des fonctions renvoyant explicitement l'adresse de l'ensemble. Mais attention !, cet ensemble est modifié. Ces fonctions sont donc bien avec mutations.

```
Ens* ensvide()/* ensemble vide : forme 2 */
{
      Ens* e;
      e = MALLOC(Ens);
      e->n = 0;
      return e;
}
Ens* insertion(Ens* e, Nat x)
/* insertion de x dans *e modifie : forme 2 */
{
      e->v[e->n] = x;
      (e->n)++;
      return e;
}
Ens* suppression(Ens* e, Nat x)
/* suppression de x de *e modifie : forme 2 */
{
      Nat k;
      (e->n)--;
      for(k = e->n; e->v[k] != x; k--);
            /* k:place de x dans e.v */
      e->v[k] = e->v[e->n];
      return e;
}
```

Avec les fonctions de la forme 2 sur le même exemple d'utilisation qu'au-dessus, nous allons passer en paramètre par un pointeur sur l'ensemble `e` :

```
Ens* e;
```

Ainsi, nous pouvons appeler les opérations et les composer de manière *fonctionnelle* :

```
e = suppression(insertion(insertion(ensvide(), 3), 7), 3);
```

Dans les deux formes, la structure ensemble est modifiée exactement de la même manière. ∎

Bien que les types d'écritures ci-dessus soient très courants dans la littérature C, nous les éviterons dans la suite parce que l'apparition explicite d'adresses ou de pointeurs avec `&` et * va à l'encontre de la recherche d'*abstraction* dans la programmation car nous voulons pouvoir conserver le même *profil* pour les opérateurs qu'ils soient fonctionnels ou par mutations et ainsi que le même nom de *type* quelque soit la structure de données retenue pour les ensembles.