Types abstraits de données (TAD)
CTD 1

J.P. Bahsoun, M.C. Lagasquie, M.Paulin

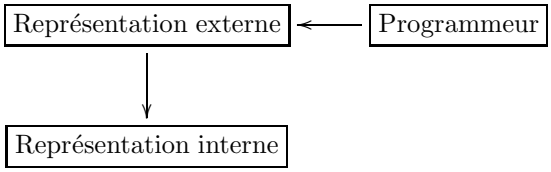
16 août 2011

Table des matières

1 Introduction

Ici, on va travailler la manière de spécifier un type abstrait, en illustrant sur le TAD Pile, puis on étudiera les diverses implémentations possibles (statiques et dynamiques).

Dans une programmation "en large" il est nécessaire d'identifier et de spécifier les données assez tôt dans le processus de développement (en général, la programmation "en large" est réalisée par une équipe, par opposition à la programmation "en petit" qui est réalisée par une personne seule).
Une fois la donnée complexe identifiée, nous nous intéressons à sa spécification. Le but de cette spécification est de définir l'interface d'utilisation (**représentation externe**) de cette donnée et de lui donner une sémantique abstraite indépendante de l'implantation (**représentation interne**).
Les langages de programmation impératifs typés offrent des types dits prédéfinis. Ceci conduit à une utilisation naturelle et simple des variables définies à partir de ces types. Ces types sont souvent définis d'une façon abstraite puis implémentés dans le langage de programmation.

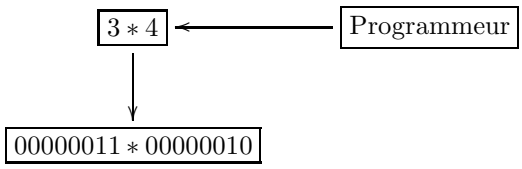


Exemple des entiers
- Représentation externe des entiers vue par le programmeur (en C) :

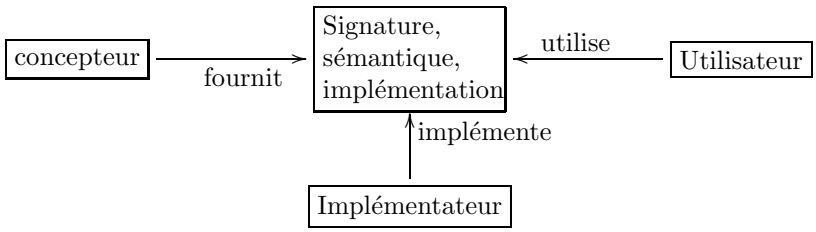
```
int                 /* le nom du type */
5, -6, 21           /* des nombres */
+, -, *, /          /* les opérations permises */
```

  Ceci représente l'interface des entiers. Cette représentation externe donne la possibilité d'une utilisation naturelle des entiers .
- Représentation interne des entiers : il s'agit d'une représentation binaire ; par exemple, l'entier 3 est codé par 00000011 sur un octet.

La représentation externe est incomparablement plus facile à utiliser qu'une représentation interne.
Exemple de la multiplication de deux entiers : il est plus facile d'écrire $3 * 4$ que $00000011 * 00000010$. On aura donc :



On a donc le schéma suivant :

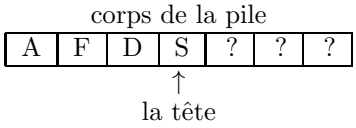


Dans ce cours, nous nous intéressons aux types abstraits de données les plus souvent utilisés. Nous proposons la classification suivante :
- Les types à structure linéaire : Liste, Pile, File
- Les types à structure arborescente : Arbre Binaire, Forêt
- Les Graphes

2 La spécification d'un TAD

Elle se fait en deux phases :
- on donne tout d'abord la *spécification fonctionnelle* appelée aussi *signature*, c'est-à-dire on décrit l'interface qui permettra d'utiliser ce TAD : donner un nom significatif au type, donner les profils des opérations du type, sachant qu'une opération est considérée comme une fonction.
- puis on donne la *sémantique abstraite* de ce TAD : pour cela nous utilisons le langage de la logique équationnelle. Ce travail se décompose en trois phases :
  - Nous commençons par partager les opérations du type en deux catégories : les **constructeurs** et les **opérateurs**. Un constructeur est indispensable à la représentation des valeurs du type et l'ensemble des constructeurs choisi doit être nécessaire et suffisant pour "construire" toutes les valeurs possibles du type.
  - Puis nous identifions les restrictions des opérations (constructeurs ou opérateurs) ; en effet, les opérations sont toutes considérées comme des fonctions totales ou partielles dont la restriction s'exprime par des **préconditions**.
  - Et enfin, nous caractérisons chaque opération par des **axiomes**. En général, ces axiomes sont construits par l'application d'un opérateur sur un constructeur si la précondition est satisfaite.

Exemple des piles Une pile est un type de données permettant de rassembler des données d'un type donné et auquel on accède par un seul point d'entrée (la tête) ; les piles sont donc gérées sur le principe du "dernier entré, premier sorti" (LIFO).



Les idées que l'on veut exploiter pour définir ce type Pile sont :
- La pile doit être vide à la création
- Une pile vide est construite par *Créer*
- Une pile non vide est construite par *Créer* suivie par une suite d'*Empiler*

Créer et *Empiler* seront les constructeurs ; elles sont indispensables pour représenter n'importe quel terme du type Pile (pile vide ou non).
Exemple : *Empiler (Empiler (Empiler (Empiler (créer,'A'),'F'),'D'),'S')*

Spécification du TAD pile d'entiers

Sorte : Pile
Utilise : Entier, Booléen
Opérateurs_Constructeurs :
Créer : $\rightarrow$ Pile
Empiler : Pile $\times$ Entier $\rightarrow$ Pile
Opérateurs_Projecteurs :
Appartient : Pile $\times$ Entier $\rightarrow$ Booléen
Est_Vide : Pile $\rightarrow$ Booléen
Dépiler : Pile $\rightarrow$ Pile
Sommet : Pile $\rightarrow$ Entier

Préconditions :
Dépiler(p), Sommet(p) ssi $\neg$ *Est_Vide(p)*

Axiomes :
Appartient(Créer, x) = faux
Appartient(Empiler(p, x),y) = (x = y) $\vee$ *Appartient(p,y)*

Est_Vide(Créer) = vrai
Est_Vide(Empiler(p,x)) = faux

Dépiler(Empiler(p,x)) = p

Sommet(Empiler(p,x)) = x

3 Implémentation d'un TAD

L'implémentation d'un type abstrait consiste à :
- Implémenter une représentation interne du type
- Implémenter les fonctions
- Assurer la séparation entre l'implémentation de l'interface (signature) et le corps. Deux grands avantages :
  1. Permet de modifier le corps sans toucher à l'interface donc sans modifier les programmes utilisateurs
  2. manipuler par abstraction
- Assurer la protection de la représentation interne

3.1 Implanter une représentation interne du type et implémenter les fonctions

Nous sommes souvent devant un choix entre une implémentation statique ou une implémentation dynamique :

Statique : signifie que la réservation des variables est effectuée au chargement du programme, et que lors de l'exécution ces variables ne changent pas de place par rapport à l'espace mémoire du programme. (cf. cours sur les tableaux – semestre précédent)
- Avantages : accès rapide
- Inconvénients : occupation inutile de l'espace mémoire pendant l'exécution

Dynamique : les réservations et libérations des variables s'effectuent en cours d'exécution. (cf. cours sur les pointeurs)
- Avantages : optimisation de l'espace mémoire occupé
- Inconvénients : l'efficacité est diminuée par le temps passé à la gestion de la mémoire (qui se fait au cours de l'exécution).

3.1.1 Implémentation Statique du TAD Pile

Pour représenter la pile, nous avons besoin de représenter 2 informations : le corps de la pile (un tableau d'entier de taille N) et l'indice de tête (un entier). Nous pouvons regrouper ces deux informations dans la même structure en C :

```
typedef struct {
    int indice;
    int t[N];} pile;
pile p;
```

L'accès au champ `indice` se fait par `p.indice` qui est utilisé comme une variable de type `int`. Par exemple, `p.indice = 5;` L'accès au corps se fait par `p.t` qui est utilisé comme un tableau de taille N. Par exemple, `p.t[p.indice] = 0;`
Attention : l'utilisation de l'instruction `assert` sera expliquée lorsqu'on parlera de la séparation corps/interface (section 3.2).

```
#include <stdio.h>
#define N 10;

typedef struct  {
    int indice;               /* répresente la dernière case remplie : la tête */
    int t[N];} pile;

void creer_pile (pile *p) {
    p->indice = -1; }  /* la pile est créée vide */
```

```
void empiler (pile *p, int e) {
    assert(p->indice != N-1) ;  /* précondition  en liaison avec */
           /* l'implémentation statique correspondant à la pile est pleine */
    p->indice ++;
    p->t[p->indice] = e; }

void depiler (pile *p) {
    assert (!est_vide(*p));          /* précondition*/
    p->indice--; }

int est_vide (pile p) {
    return (p.indice == -1);}

int sommet (pile p)  {
    assert (!est_vide(p)) ;           /* précondition*/
    return (p.t[p.indice]);}
```

`assert(condition)` est une operation permettant de sortir du programme quand la condition n'est pas vérifiée et si elle est vérifiée le sous-programme poursuit son exécution.
`p->indice` est une notation simplifiée correspondant à `(*p).indice`.

Exemple d'utilisation de ce TAD :

```
pile P, Q;
int x;
créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);
```

3.1.2 Cours sur les pointeurs

Voir Annexe A.

3.1.3 Rappel sur les structures

Voir Annexe B.

3.1.4 Les pointeurs + les structures

Voir Annexe C.

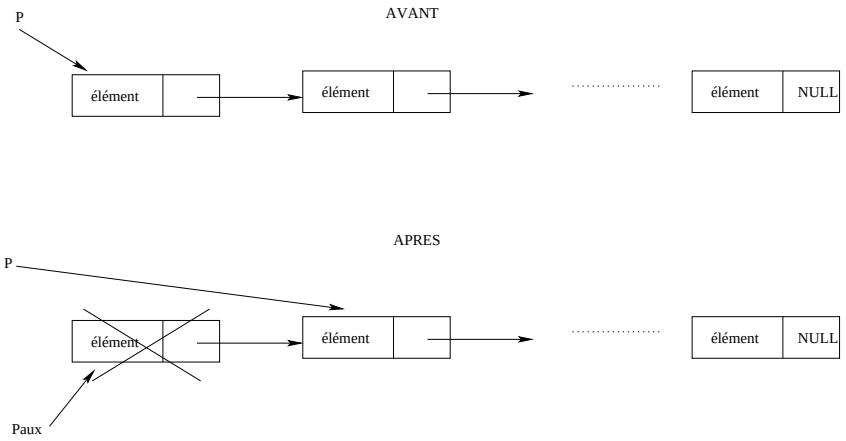
3.1.5 Implémentation Dynamique du TAD Pile

```
typedef  struct cel {
        int info;
        struct cel *  suiv;
        } cel ;   /* cel = structure à 2 champs (entier, ptr sur cel) */

typedef struct  cel *  pile ;     /* pile = pointeur sur une cel */
               /* cel et struct cel sont des synonymes */
               /* cel *, struct cel * et pile sont des synonymes */

void creer ( pile * p) {
        /* création d'une pile à vide */
        *p = NULL ; }

int est_vide (pile p) {
         /* vérifier si une pile est vide
            (renvoie 0 si non et une valeur différente de 0 si oui)*/
        return (p == NULL); }
```

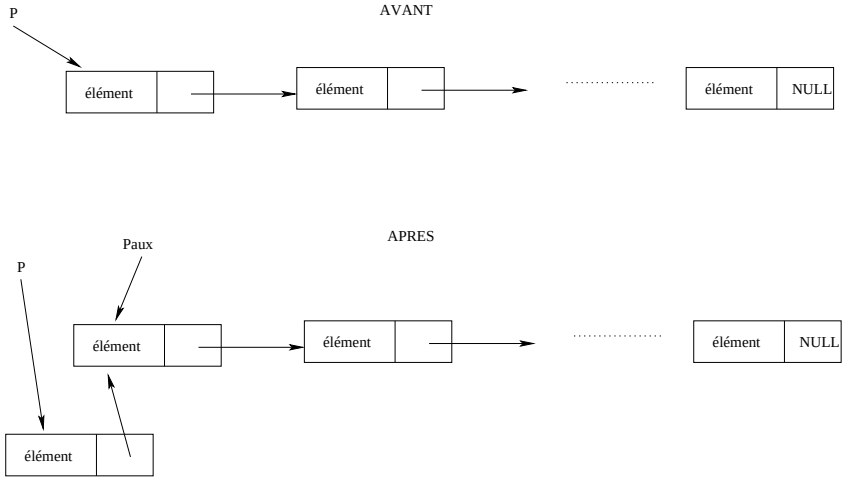
```
int sommet (pile p) {
         /* récupérer la valeur en sommet d'une pile */
        assert (!est_vide(p)) ;
        return( p->info) ; }

void depiler (pile * p) {
         /* supprime le sommet de pile */
        pile  paux;
        assert (!est_vide(*p)) ;
        paux = *p ;
        *p = (*(*p)).suiv ;
        free(paux);
        }

void empiler (pile * p, int x) {
        /* rajoute une valeur à la pile */
        pile paux ;
        paux = *p ;
        *p = (pile)malloc(sizeof(struct cel));
        assert (!est_vide(*p)) ; /* plus de place mémoire => allocation échoue */
        (*(*p)).info = x ;
        (*(*p)).suiv = paux ;
        }
```

Le fonctionnement de `depiler` et de `empiler` est décrit par les figures suivantes

Fonctionnement de "depiler"



Fonctionnement de "empiler"



3.2 Séparer interface et corps

Un premier pas dans l'implémentation des données de type abstrait est de séparer l'interface de l'implémentation des fonctions. L'interface est constituée ici par la représentation du type et les profils des fonctions.
Dans un fichier qui représente l'interface `piles.h` (le "s" indiquant statique), nous implémentons l'interface ; ce fichier est appelé un fichier "header" son nom contient l'extension `.h`.

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

// les deux lignes suivantes permettent de se protéger contre
// les inclusions multiples
#ifndef __SYMBOLEUNIQUE__
#define __SYMBOLEUNIQUE__

#define N 10
typedef struct pile {
     int indice;     /* représente la dernière case remplie : la tête */
     int t[N];} pile;

void creer (pile *);
void empiler (pile *, int );
void depiler (pile *);
int est_vide (pile );
int sommet (pile);

#endif // ferme le ifndef __SYMBOLEUNIQUE__
```

Dans un fichier qui contient l'implémentation du type `piles.c`, on implémente les corps des fonctions en incluant le fichier `piles.h` pour avoir accès au type PILE.

```
#include "piles.h"

void creer (pile *p) {
     p->indice = -1; } /*la pile est créée vide */
void empiler (pile *p, int e){
     assert (p->indice != N-1) ;
     p->indice ++;
     p->t[p->indice] = e; }
void depiler (pile *p) {
     assert (!est_vide(*p)) ;
     p->indice--;}
int est_vide (pile p) {
     return (p.indice == -1);}
int sommet (pile p) {
     assert (!est_vide(p)) ;
     return (p.t[p.indice]);}
```

L'utilisateur (programmeur), pour pouvoir manipuler des piles, n'a qu'à inclure `piles.h` dans son fichier source et lier son fichier objet (avec un suffixe `.o`) au `piles.o`. Par exemple, dans le fichier `testPile.c`, on trouvera :

```
#include "piles.h"

int main() {
pile P, Q;
int x;
créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);}
```

3.3 Protection du type

L'implémentation précédente permet au programmeur utilisateur de manipuler explicitement la structure du type. Par contre rien interdit à un programmeur d'écrire dans son code :

```
p.indice = 5;
```

alors que la valeur d'indice précédente était, par exemple, 1. Ceci casse la définition de la pile.
Pour protéger cette structure interne nous avons intérêt à la cacher dans le fichier `piles.c`. Le problème est de pouvoir donner la possibilité au programmeur de faire respecter la définition des piles. La structure interne sera dans `piles.c`. L'accès à cette structure protégée se fera à travers un type pointeur implémenté dans `piles.h`
Dans notre cas, cela donnera :

```
#include <stdio.h>
typedef struct pile *pile;

void creer (pile *);
void empiler (pile *, int );
void depiler (pile *);
int est_vide (pile);
int sommet (pile);
```

Dans le fichier `piles.c`, on inclut le fichier `piles.h` puis on implémente la structure et le corps des fonctions :

```
#include "pile.h"
#define N 10
struct pile {
     int indice;   /* représente la dernière case remplie : la tête */
     int t[N];};

void creer (pile *p) {
      p->indice = -1; }
void empiler (pile *p, int e){
      assert (p->indice != N-1) ;
      p->indice ++;
      p->t[p->indice] = e; }
void depiler (pile *p) {
      assert (!est_vide(*p)) ;
      p->indice--;}
int est_vide (pile p) {
      return (p->indice == -1);}
int sommet (pile p) {
      assert (!est_vide(p)) ;
      return (p.t[p.indice]);}
```

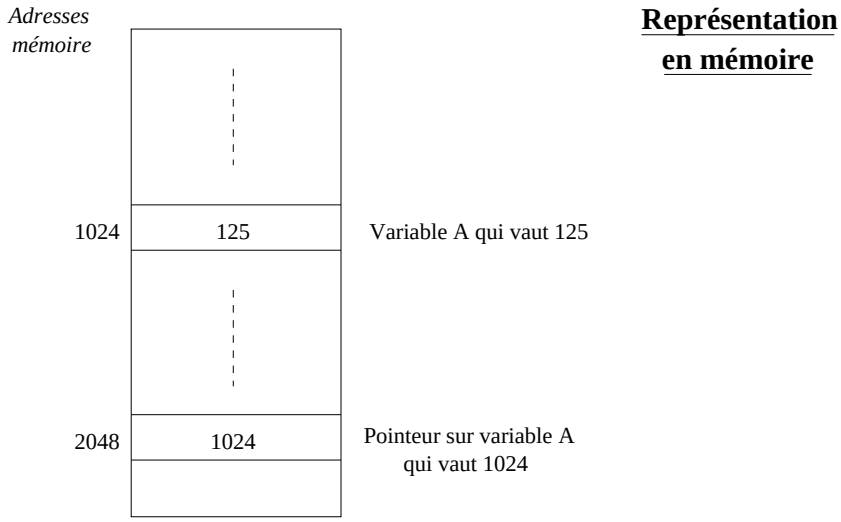
En fournissant seulement le fichier `piles.h` aux utilisateurs, on leur interdit de toucher à la structure interne du type et on la protège. Et cela ne doit rien changer au fichier `testPile.c` qui utilise la pile :

```
#include "piles.h"

int main() {
pile P, Q;
int x;
créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);}
```

A Cours sur les pointeurs

A.1 Définition

Un *pointeur* = variable dont la valeur correspond à une adresse.



A.2 Syntaxe en langage C

Les pointeurs correspondent à un type spécial : le type *pointeur*.

- en zone déclarative, on écrit : *type_que_l_on_veut * p ;*
  par exemple, si on écrit dans le programme :

  $$int\ *\ p$$

  il y a alors réservation mémoire dans le programme d'une case qui peut contenir l'adresse d'un entier, cette case est répérée par l'identificateur p et sa valeur au moment de la déclaration est non significative (exactement comme lorsqu'on déclare *int n*, il y a réservation mémoire dans le programme d'une case qui peut contenir un entier, cette case étant répérée par l'identificateur n et sa valeur au moment de la déclaration étant non significative) ;
- dans le corps du programme, on utilise
  - soit p pour manipuler l'adresse,
  - soit $*p$ pour manipuler la valeur pointée (le symbole * est appelé un constructeur et il sert aussi d'opérateur d'indirection).

Remarque : on a donc utilisé (sans le dire !) des pointeurs pour "simuler" le passage de paramètres par référence en langage C.

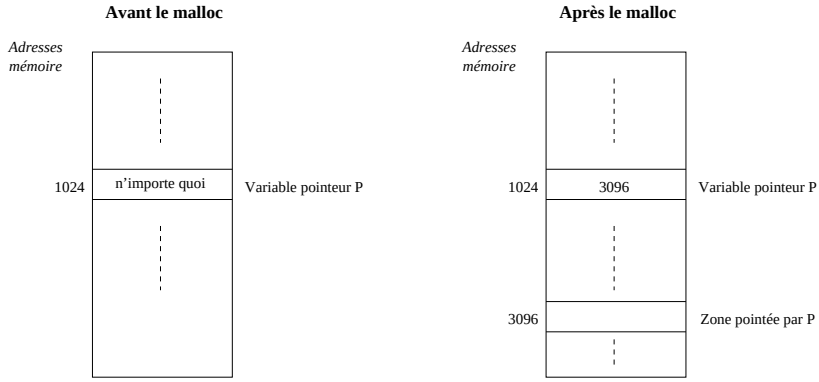
Opérations autorisées En langage C, il existe 6 opérations possibles sur les pointeurs :

- Les 4 premières ont été étudiées au semestre 1 :
  - L'affectation :
    $p = q$; (p et q doivent être des pointeurs sur un même type),
    $p = NULL$; ($NULL$ est l'élément nul pour le type pointeur),
    $p = \&x$; (x est une variable de type quelconque noté T et p est une variable de type pointeur sur le type T) (le symbole & sert à obtenir l'adresse d'une variable, c'est l'opérateur d'adressage).
  - L'addition et la soustraction d'un pointeur avec un entier (on parle aussi de *décalage*) :
    $p = q + 1$; (p et q doivent être des pointeurs sur un même type et sur une même entité mémoire[1]),
    $p = q - 10$; (p et q doivent être des pointeurs sur un même type et sur une même entité mémoire).
  - La soustraction de 2 pointeurs de même type :
    $i = p - q$; (p et q doivent être des pointeurs sur un même type et sur une même entité mémoire et i est un entier).

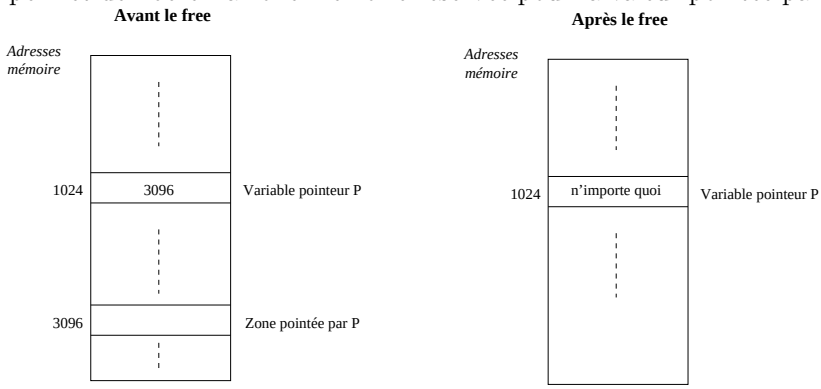
1. C'est-à-dire sur un même tableau.

– La comparaison de 2 pointeurs de même type :
$p > q$ (p et q doivent être des pointeurs sur un même type et sur une même entité mémoire).

– Les 2 dernières sont liées à la gestion dynamique de la mémoire :
  – La création dynamique (réservation et mise à jour) : p = *(type_de_p) malloc(taille_zone_pointée)* avec p une variable de type pointeur.
  Cette fonction C permet de réserver une zone mémoire pour la valeur pointée par p et met à jour le pointeur p avec l'adresse de cette zone.



  Remarque : si, après l'exécution de cette instruction, le pointeur résultat est égal à NULL, cela signifie que la mémoire est saturée et qu'on ne peut plus faire d'allocation dynamique.
  – La suppression dynamique : *free(p)* avec p une variable de type pointeur.
  Cette fonction C permet de libérer la zone mémoire réservée pour la valeur pointée par p.



  Attention : il arrive qu'après un *free(p)*, on puisse encore accéder à l'ancienne zone pointée par p (cela dépend de la machine et de son gestionnaire mémoire), c'est donc une source importante d'erreur ; un conseil : après *free(p)*, faire $p = NULL$.

Avantages Gestion dynamique de la mémoire et souplesse d'implémentation.

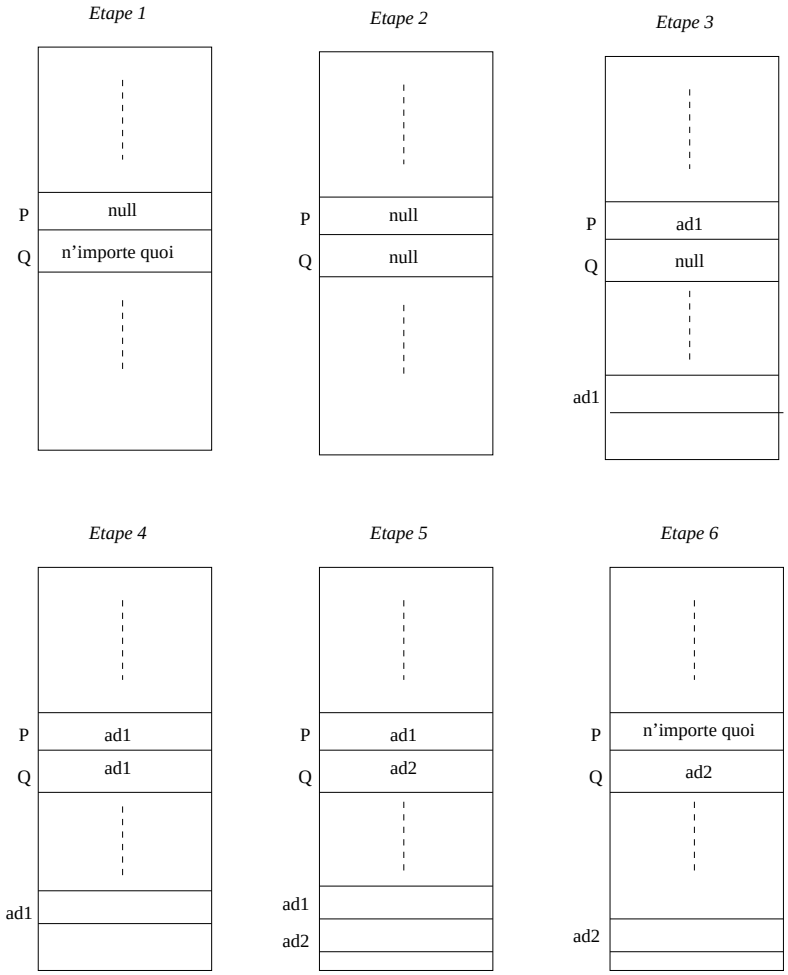
A.3 Exemples sur les pointeurs

1. Soit le programme suivant :

```
...
#include <stdio.h>
...
int * P, * Q ;
...
P = NULL ;                          /* étape 1 */
Q = P ;                             /* étape 2 */
P = (int *) malloc(sizeof(int)) ;   /* étape 3 */
Q = P ;                             /* étape 4 */
Q = (int *) malloc(sizeof(int)) ;   /* étape 5 */
free (P) ;                          /* étape 6 */
...
```

  Chaque étape est représentée par un schéma de la mémoire, et le déroulement du programme donne la

chose suivante :



2. Soit le programme suivant :

```
int * p, * q ;
p = (int *) malloc(sizeof(int)) ;
printf("Après malloc : %d \n", *p) ;
q = p ;
printf("Après q = p : %d et %d \n", *p, *q) ;
*p = 15 ;
printf("Après maj : %d et %d \n", *p, *q) ;
free(p) ;
printf("Après free : %d \n",*q) ;
```

   Ici, le résultat obtenu à l'impression est :

```
Après malloc : Valeur1_quelconque
Après q = p : Valeur1_quelconque et Valeur1_quelconque
Après maj : 15 et 15
Après free : 15        /* ATTENTION : erreur possible (dépend de la machine !) */
                  /* En effet, q pointe sur une zone mémoire qui a été libérée ! */
```

3. Soit le programme suivant :

```
/* définition de nouveaux types */
typedef int zone ;                                          /* zone sera le type int */
typedef zone * ptrzone ;                                /* ptrzone sera le type pointeur sur zone */
void maj_zone_pointee(ptrzone p, zone val) {
      *p = val ;
      }
```

main() {
ptrzone p ;
*p = (int *) malloc(sizeof(int)) ;*
*printf("Avant maj : %d\n",*p) ;*
maj_zone_pointee(p,10) ;
*printf("Après maj : %d\n",*p) ;*
}

Ici, le résultat obtenu à l'impression est :

```
Avant maj : Valeur_quelconque
Après maj : 10
```

Remarque : cette fonction permet de modifier une zone par l'intermédiaire de son adresse. C'est ainsi que l'on procède dans les langages où seul le passage par valeur est autorisé pour modifier des valeurs lors de l'appel d'un sous-programme (ex : le langage C).

A.4 Conclusion sur les pointeurs

Les pointeurs sont des variables dont la valeur est une adresse. Ils servent à faire de l'allocation dynamique de mémoire et à "simuler" un passage de paramètre par référence.

B Rappels sur les structures

B.1 Définition

Il s'agit d'un type de données composées permettant donc de "regrouper" des valeurs. Par contre, à la différence des **tableaux** qui regroupent des données de **même type**, une **structure** permet de regrouper des données de types **différents**.

B.2 Syntaxe en langage C

Il s'agit de définir un modèle de données que l'on pourra utiliser ensuite comme type pour définir des variables.

```
struct nom_modele {
         type1 champ1;
         type2 champ2;
         ...
         typeN champN;
       } ;
```

Chaque élément ou *champ de la structure* est défini par son type et son nom.
Exemple :

```
/* définition d'un modèle de donnée regroupant 2 */
/* entiers et une chaîne de caractères           */

struct Date  {
         int jour;
         char mois[10];
         int annee;
     };
```

La combinaison du mot-clé `struct` et du nom donné au modèle est utilisée ensuite pour définir des variables, à moins qu'un type spécifique ait été défini au moyen de l'opérateur `typedef`.

Utilisation de l'opérateur `typedef` Il s'agit de la possibilité de renommer un type puisque celui-ci doit être décrit au moment de la définition.

```
typedef description_type nom_type;
```

Exemple :

```
/* définition d'un type nommé type_date et */
/* correspondant au modèle de donnée       */
/* struct Date                             */

typedef struct Date  {
       int jour;
       char mois[10];
       int annee;
       }      type_date;
```

Le nom du type défini peut alors être utilisé pour la définition de variables.
Remarque : L'opérateur `typedef` fréquemment utilisé avec les structures peut également s'utiliser pour définir d'autres types de données.

Définition d'une variable correspondant à une structure Ceci n'est possible qu'une fois que le modèle ou le type de la structure a été défini. La définition d'une variable entraîne l'allocation mémoire de la zone nécessaire pour stocker les données correspondantes.

```
/* définition d'une variable de nom Jour_J et contenant*/
/* 3 champs : jour, mois et annee                      */
struct Date Jour_J;
```

ou

```
type_date Jour_J;
/* dans la mesure où le type type_date a été défini */
/* comme ci-dessus                                  */
```

Initialisation d'une variable correspondant à une structure Comme pour les tableaux, il faut spécifier la liste des valeurs de chaque élément.

```
struct nom_modele nom_var = {val_chp1, ..., val_chpN};
```

Exemple :

```
struct Date Noel = {25, "décembre", 2000 };
type_date Nouvel_an = {1, "janvier",2001};
```

Utilisation d'une structure dans le corps d'un programme En dehors de l'initialisation faite au moment de la définition de la donnée, on peut :

- accéder à chaque champ individuellement en utilisant la notation pointée (on pourra alors utiliser ce champ comme n'importe quelle variable du même type) :

  ```
  nom_variable.nom_champ
  ```

  Exemple :

  ```
  Jour_J.jour= 14;
  strcpy(Jour_J.mois, "juillet");
  Jour_J.annee = 2000;

  /* pour attribuer une valeur à une chaîne de caractères  */
  /*  (=tableau de caractères) il faut utiliser la         */
  /* fonction strcpy définie dans la bibliothèque string.h */
  ```

- affecter une valeur à une structure :
  Contrairement au tableau qu'il faut traiter élément par élément, il est possible d'affecter la valeur d'une structure à une autre structure.
  Exemple :

  ```
  type_date Autre_jour;
  ...
  Autre_jour = Jour_J;
  ```

Remarque : imbrication de structures Une structure peut admettre un champ qui soit lui même une structure. Seul cas impossible, lorsque le champ doit être du type de la structure en cours de définition. Dans ce cas particulier, celui des structures récursives, il faut passer par un champ qui soit un pointeur sur une donnée du type en cours de définition.
Exemple :

```
/* définition d'un autre type de structure */
struct Personne
   {
     char nom[20];
     char prenom[20];
     struct Date ne_le;
   };

/* définition et initialisation d'une variable */
/* comportant une structure imbriquée             */
struct Personne P1 = {"Durand", "Paul", {12, "mai", 1980}};

/* exemple d'acces au champ d'une structure imbriquée */
if(P1.ne_le.annee >1970)
        printf("%s %s a moins de 30 ans", P1.prenom, P1.nom);
```

Les structures auto-référentielles Le seul moyen d'avoir une structure récursive c'est-à-dire qui admet au moins un champ correspondant à une donnée du même type que la structure en cours de définition est d'utiliser un pointeur sur le type en question.
Exemple :

```
struct membre_famille                /* erreur */
 {         char nom[20];
           char prenom[20];
           int age;
           struct membre_famille pere, mere; } ;

   /* n'est pas possible car un type tel que          */
   /* struct membre_famille ne peut être utilisé     */
   /* qu'une fois complètement défini                */
   /* il faut donc utiliser un pointeur              */

struct membre_famille                /* correct */
 {         char nom[20];
           char prenom[20];
           int age;
           struct membre_famille *ptr_pere, *ptr_mere; } ;
```

Ce type de structure est utilisé comme élément de base d'une structure plus complexe où des données de même type sont liées les unes aux autres.

C Les pointeurs + les structures

On peut définir un pointeur sur des données de tout type y compris sur des structures et mêmes sur des pointeurs.
Exemple, en utilisant les structures `Date` et `Personne` :

```
struct Date *ptr_date ;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le    */
/* pointeur ptr_date ne pointe sur aucune   */
/* donnée valide                            */
```

```
struct Personne * ptr_pers;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le    */
/* pointeur ptr_pers ne pointe sur aucune   */
/* donnée valide                            */

struct Date une_date = {14, "Juillet", 2000};
struct Personne une_personne= {"Pierre", "Dubois", {12, "Avril", 1980}};
/* définition et allocation des  octets nécessaires */
/* pour stocker une première donnée correspondant à */
/* une structure date et une autre correspondant à  */
/* une structure personne                           */
```

Accès au champ d'une structure via un pointeur Si le type de donnée associé à un pointeur (= type de la donnée pointée) est une structure, et que bien sûr le pointeur a reçu l'adresse d'une variable du type de données en question alors le pointeur permet d'accéder indirectement aux champs de la structure pointée. Deux notations sont possibles et totalement équivalentes :

`nom_ptr->nom_champ` La flèche représentée par la combinaison des symboles - (moins) et > (supérieur) indique que l'on accède indirectement au champ spécifié de la donnée pointée par le pointeur.

`(*ptr_nom).nom_champ` La notation pointée ne peut être utilisée que si la partie gauche correspond à une variable de type structure. Pour obtenir cela à partir d'un pointeur, il faut d'abord appliquer l'opérateur d'indirection * au pointeur ; le parenthésage est obligatoire. On obtient ainsi la donnée pointée. On peut donc ensuite utiliser la notation pointée pour accéder au champ recherché.

Exemple :

```
ptr_date = &une_date ;
/* le pointeur re¸coit l'adresse d'une donnée */
/* ptr_date pointe sur la donnée une_date    */

/* affichage de la valeur de la donnée une_date de 3 fa¸cons */
/* différentes                                               */
printf(" une date = %d %s %d", une_date.jour, une_date.mois, une_date.annee);
printf(" même chose = %d %s %d",ptr_date->jour, ptr_date->mois, ptr_date->annee);
printf(" encore la même chose = %d %s %d", (*ptr_date).jour, (*ptr_date).mois,
          (*ptr_date).annee);

ptr_pers = &une_personne;
/* le pointeur re¸coit l'adresse d'une donnée  */
/* ptr_pers pointe sur la donnée une_personne */

/* affichage de la date de naissance de la donnée */
/* une_personne de 3 fa¸cons différentes          */
printf(" une date = %d %s %d", une_personne.ne_le.jour, une_personne.ne_le.mois,
          une_personne.ne_le.annee);
printf(" même chose = %d %s %d", ptr_pers->ne_le.jour, ptr_pers->ne_le.mois,
          ptr_pers->ne_le.annee);
printf(" encore la même chose = %d %s %d", (*ptr_pers).ne_le.jour, (*ptr_pers).ne_le.mois,
          (*ptr_pers).ne_le.annee);
```

Exemples sur les pointeurs + les structures

1. Soit le programme suivant :
   typedef struct et_cellule {
   int valeur ;
   *int * suivant ;*
   } *cellule ;* */* *cellule est une structure à 2 champs (entier, pointeur sur entier)* */*

```
                                        /* cellule et struct et_cellule sont des synonymes */
main()
        { cellule c ;
        c.valeur = 10 ;
        c.suivant = (int *) malloc(sizeof(int)) ;
        *(c.suivant) = 11 ;
        printf( "%d %x %d ", c.valeur, c.suivant, *(c.suivant)) ;
        }
```

Ici, le résultat obtenu à l'impression est :

```
10 Adresse 11
```

2. Soit le programme suivant :

```
typedef struct et_cellule {
        int valeur ;
        struct et_cellule * suivant ;
        } cellule ;                 /* cellule est une structure à 2 champs (entier, pointeur sur cellule) */
typedef cellule * ptrcellule ;                          /* ptrcellule est un pointeur sur une cellule */
                                                    /* cellule et struct et_cellule sont des synonymes */
                                        /* cellule *, struct et_cellule * et ptrcellule sont des synonymes */
main()
        { ptrcellule p ;
        p = (ptrcellule) malloc(sizeof(cellule)) ;
        p->valeur = 10 ;
        p->suivant = (ptrcellule) malloc(sizeof(cellule)) ;
        p->suivant->valeur = 11 ;
        p->suivant->suivant = (ptrcellule) malloc(sizeof(cellule) ;
        p->suivant->suivant->valeur = 12 ;
        printf( "%x, %d, %x, %d, %x, %d, %x, %d ",
                        p,
                        p->valeur,
                        p->suivant,
                        p->suivant->valeur,
                        p->suivant->suivant,
                        p->suivant->suivant->valeur,
                        p->suivant->suivant->suivant,
                        p->suivant->suivant->suivant->valeur) ;
        }
```

On n'a pas fait de malloc pour *p->suivant->suivant->suivant*. On a donc deux cas possibles :

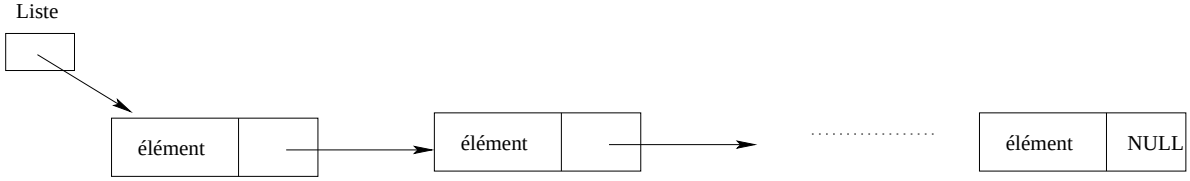
- soit le programme "plante", car *p->suivant->suivant->suivant* contient une adresse inderdite au programme et alors l'accès à *p->suivant->suivant->suivant->valeur* provoquera une erreur ;
- soit le programme s'exécutera sans erreur mais les deux dernières valeurs affichées sont non significatives. Le résultat obtenu à l'impression sera donc :

```
Adresse0 10 Adresse1 11 Adresse2 12 ? ?
```

3. Représentation de listes (structures de données linéaires) en dynamique.

On a au moins 4 cas possibles de listes :

- la liste simplement chaînée avec un seul point d'accès :

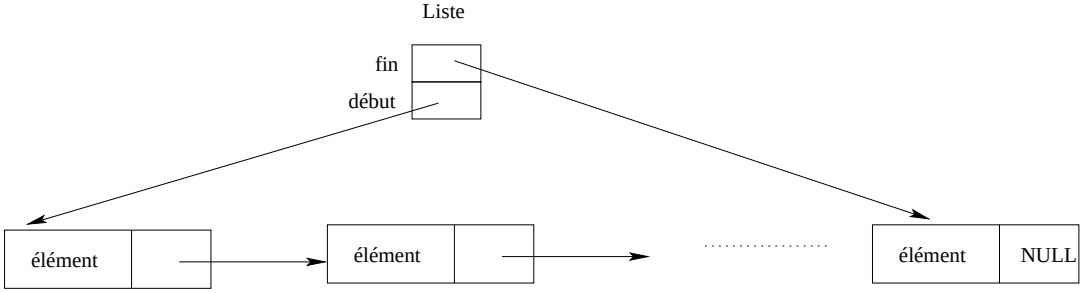


Cela correspondant au type suivant en langage C :

```
typedef struct et_cel_liste_simple {
        int info ;
        struct et_cel_liste_simple * suiv ;
        } cel_liste_simple ;
                        /* cel_liste_simple = structure à 2 champs (entier, pointeur sur cel_liste_simple) */
```

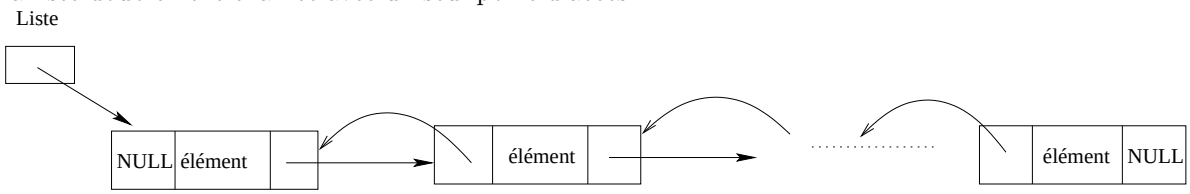
*typedef struct cel_liste_simple * liste_simple_1_pt ;*
/ liste_simple_1_pt = pointeur sur une cel_liste_simple */*
/ cel_liste_simple et struct et_cel_liste_simple sont des synonymes */*
/ cel_liste_simple *, struct cel_liste_simple * et liste_simple_1_pt sont des synonymes */*

– la liste simplement chaînée avec deux points d'accès :



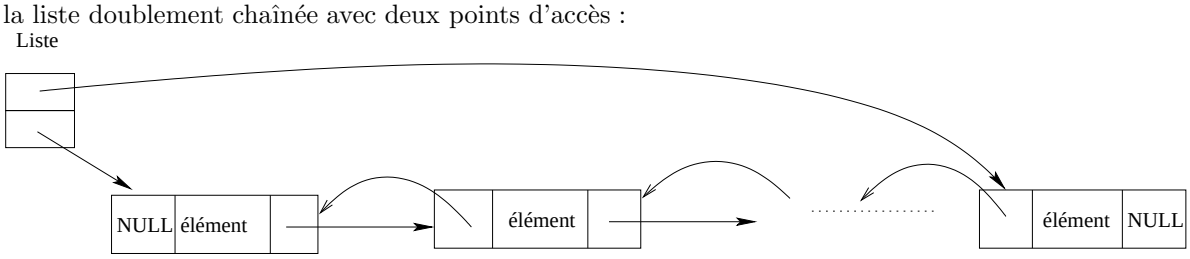
Cela correspondant au type suivant en langage C qui réutilise le type de la cellule simplement chaînée :
typedef struct et_liste_simple_2_pt {
*cel_liste_simple * premier ;*
*cel_liste_simple * dernier ;*
} *liste_simple_2_pt ;*
/ liste_simple_2_pt = structure à 2 champs : 2 pointeurs sur cel_liste_simple */*
/ liste_simple_2_pt et struct et_liste_simple_2_pt sont des synonymes */*

– la liste doublement chaînée avec un seul point d'accès :



Cela correspondant au type suivant en langage C :
typedef struct et_cel_liste_double {
int info ;
*struct et_cel_liste_double * suiv ;*
*struct et_cel_liste_double * prec ;*
} *cel_liste_double ;*
/ cel_liste_double = structure à 3 champs (entier, 2 pointeurs sur cel_liste_double) */*
*typedef struct cel_liste_double * liste_double_1_pt ;*
/ liste_double_1_pt = pointeur sur une cel_liste_double */*
/ cel_liste_double et struct et_cel_liste_double sont des synonymes */*
/ cel_liste_double *, struct cel_liste_double * et liste_double_1_pt sont des synonymes */*

– la liste doublement chaînée avec deux points d'accès :



Cela correspondant au type suivant en langage C qui réutilise le type de la cellule doublement chaînée :
typedef struct et_liste_double_2_pt {
*cel_liste_double * premier ;*
*cel_liste_double * dernier ;*
} *liste_double_2_pt ;*
/ liste_double_2_pt = structure à 2 champs : 2 pointeurs sur cel_liste_double */*
/ liste_double_2_pt et struct et_liste_double_2_pt sont des synonymes */*