POINTEURS

Le type pointeur

Pointeur = adresse sur un objet de type donné
Sert principalement pour gérer la mémoire dynamique

- Description d'un pointeur

  type $*$ *identificateur*

- Compatibilité avec l'opérateur d'adressage '&'
- Déclaration

```
char x, * p;
...
p = &x;  *p = 'w';
...
typedef char * type_p_char;
typedef enum{rouge,noir} * type_p_couleur;
personne * p_perso;
typedef personne * type_p_perso;
type_p_perso p_jean;
...
```

- Une définition de pointeur alloue en mémoire l'espace suffisant pour contenir une adresse
- C'est le compilateur qui contrôle le type dont la gestion est sous la responsabilité de l'utilisateur

- On peut pointer sur n'importe quel type
  - prédéfini
  - structure, union, enum
  - tableau, chaîne
  - pointeur
  - fonction
- Le type "`void *`" est un type universel de pointeur, il ne spécifie pas la nature de l'objet pointé : arithmétique impossible

```
int f(void *, void *);
```

- Taille d'un pointeur

  Elle est égale à `sizeof(void *)` qui est souvent la taille de `int`
- Compatibilité
  - Pointeurs entre eux, pointeurs et entiers sont compatibles entre eux par l'intermédiaire de l'opérateur de coercition
  - La coercition est obligatoire (en C ANSI), sauf pour le type `void *`
  - La valeur `NULL` (`(void *)0`, adresse nulle définie dans `<stdio.h>`), est commune à tous les types de pointeurs
  - La coercition en `void *` suivie de son inverse laisse la valeur d'un pointeur invariante : (p de type `TYPE *`) `(TYPE *)(void *)p` à la même valeur que `p`

Pointeurs et arguments

Les pointeurs permettent de réaliser une *transmission par adresse* des arguments aux fonctions

Les tableaux sont transmis de cette façon

Permet à une fonction de travailler directement sur les arguments de l'appel

```
...
void echange(int * px, int * py){
     int temp = *px;
     *px = *py;
     *py = temp;
}
...
int x, y;
...
/* échange des valeurs de x et y */
echange(&x, &y);
...
```

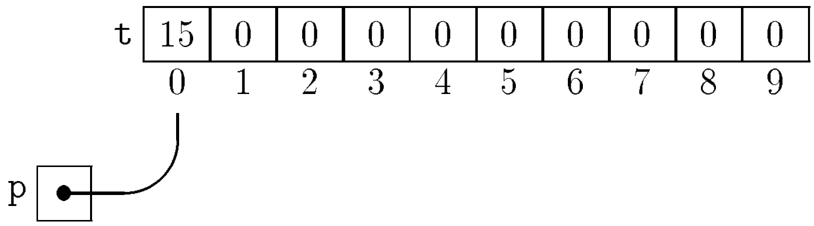
Pointeurs et tableaux

- Un tableau `t` est assimilé l'adresse `&t[0]` de son premier élément
  Une variable tableau est un *pointeur constant* sur le type de base

```
main(){
     short t[10] = 0,0,0,0,0,0,0,0,0,0;
     short *p;
     printf("t = %p, &t[0] = %p\n", t, &t[0]);
     p = t;
     *p = 15;
     printf("p = %p, t[0] = %d\n", p, t[0]);
}
```

Résultats

```
t = 001CCE68, &t[0] = 001CCE68
p = 001CCE68, t[0] = 15
```



Arithmétique des pointeurs

L'arithmétique des pointeurs est définie pour être compatible avec les tableaux : c'est une arithmétique relative à la taille du type de base

```
TRUC * p;  int i;  TRUC t[100];
```

- Pointeur `+/-` entier
  `p+i` est de type `TRUC *`
  pointe *i* objets de type `TRUC` après celui pointé par `p`
  (sa valeur entière est `(size_t)p+i*sizeof(TRUC)`)

- Compatibilité avec les tableaux :

`t[i]` est évalué comme `*(t+i)`

  (en particulier, on peut écrire `p[i]` pour `*(p+i)`)
  Si `p` pointe sur `t[k]`, `p+i` pointe sur `t[k+i]`

- Les opérateurs `++` et `--` s'appliquent aux pointeurs (pas aux tableaux qui sont des pointeurs constants)
  `++p` pointe sur l'objet suivant de type `TRUC`
  (`++p` a pour valeur entière `(size_t)p+sizeof(TRUC)`)

- Différence de pointeurs (de même type)
  nombre d'objets compris entre les deux adresses
  `(p+i)-p` vaut i, et est du type entier signé `ptrdiff_t` de `<stddef.h>`

- En particulier, `&t[i+1]-&t[i]` vaut `1`

Copie de tableau

```
...
double s[TAILLE], t[TAILLE];
...
void copie1(double s[], double t[]){
     int i;
     for(i = 0; i < TAILLE; i++)
         t[i] = s[i];
}
void copie2(double s[], double t[]){
     int i;
     for(i = 0; i < TAILLE; i++)
         *(t+i) = *(s+i);
}
void copie3(double * s, double * t){
     int i;
     for(i = 0; i < TAILLE; i++)
         *t++ = *s++;
}
...
```

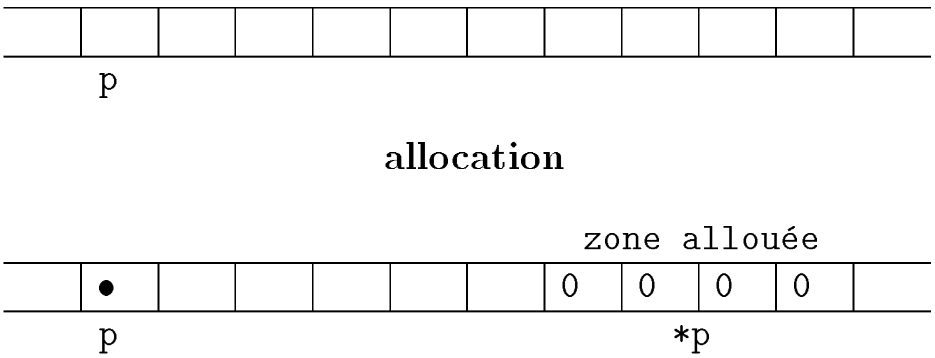
Remarque : dans les fonctions, `s` et `t` sont des variables locales de type pointeur; donc, `s++` et `t++` sont des expressions légales.
Elles ne sont pas légales sur les variables globales `s` et `t` qui sont des tableaux.

Remarques

- En paramètre de fonction
  `TYPE * t` est équivalent à `TYPE t[]`
- Comparaisons possibles (`==`, `!=`) entre pointeurs
  - `p != q`, `p == 0`
  - `(p)` à la valeur de vérité de `(p != NULL)`
  - `(!p)` à la valeur de vérité de `(p == NULL)`
- "`*p++ = i;`" a le même effet que "`*p = i, p = p+1;`"
- opérations interdites : `-` (moins unaire), `*`, `/`, `%`, opérations bit-à-bit

Allocation dynamique

Utilisation de la mémoire dynamique, le *tas*.
Il n'y a pas d'opérateur C. On utilise les fonctions de la bibliothèque `<stdlib.h>` :
— allocation : `calloc, malloc, realloc`
— libération : `free`



- `void * calloc(size_t nb, size_t t)`
  - retourne un pointeur sur une zone pour un tableau de `nb` éléments de `t` octets chacun
  - NULL en cas d'échec
  - zone initialisée à 0
- `void free(void * p)`
  - libère la zone d'adresse `p` (allouée par `Xalloc`)
  - aucun effet si `p = NULL`

```
...
double * p;
...
p = (double *)calloc(100, sizeof(double));
/* p pointe sur une table de 100 flottants "double" */
for(i=0; i<100; i++)
   p[i] = ...
...
```

Macro utile

```
#define alloue(nb,type) \
          (type *)calloc(nb,sizeof(type))
```

Autres fonctions

- `void * malloc(size_t n)`
  - retourne un pointeur sur une zone de `n` octets
  - NULL en cas d'échec
  - zone non initialisée
- `void * realloc(void *p, size_t n)`
  - modifie la taille de l'objet pointé par `p` (alloué par `Xalloc`)
    retourne un pointeur sur une zone de `n` octets
  - NULL en cas d'échec
  - recopie les $inf(t, n)$ premiers octets de `p` (`p` ayant été alloué pour t octets)

Chaînage

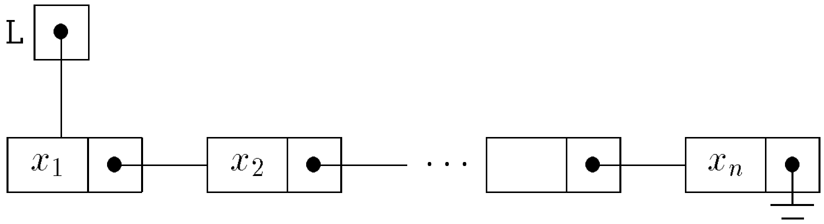
- **Structure récursive** = structure qui contient un pointeur sur une structure du même type
  Permet de représenter des suites de taille quelconque avec ajouts et suppressions efficaces d'éléments
- Déclaration

```
struct CELLULE {
     double element;
     struct CELLULE * suivant;
};
```

```
typedef struct CELLULE {
     double element;
     struct CELLULE * suivant;
} cellule;

typedef cellule * p_cellule;
```

Listes chaînées



- Liste identifiée par l'adresse de sa première cellule (comme pour les tableaux)

```
typedef p_cellule liste;
```

- Dernière cellule marquée : `NULL` dans le membre "`suivant`"

- Accès, opérateur "->"

```
p_cellule p;
...
(*p).element = 1.456;
if ((*p).suivant == NULL)
     ...
/* synonyme */
p->element = 1.456;
if (p->suivant == NULL)
     ...
...
```

- Parcours

```
p = L;
while (p != NULL) {
     ...p->element ...
     p = p->suivant;
}
```

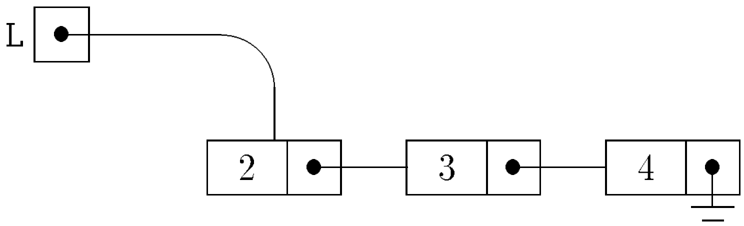
```
p = L;
while (p) {
     ...p->element ...
     p = p->suivant;
}
```

```
for (p = L; p; p = p->suivant) {
     ...p->element ...
}
```

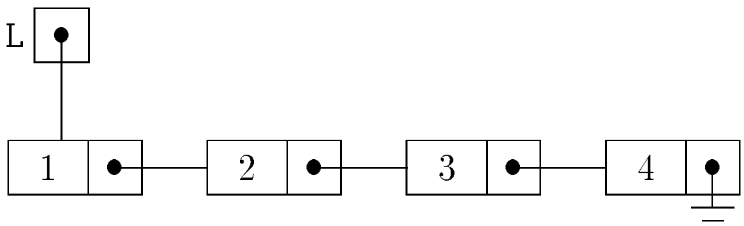
- Création

```
liste saisie(){
/* saisie d'une liste d'entiers non nuls */
     int i; liste L, temp;
     L = NULL; /* initialisation */
     scanf("%d", &i);
     while(i){
         temp = (liste) malloc(sizeof(cellule));
         temp->element = i;
         temp->suivant = L;
         L = temp;
         scanf("%d", &i);
     }
     return L;
}
```

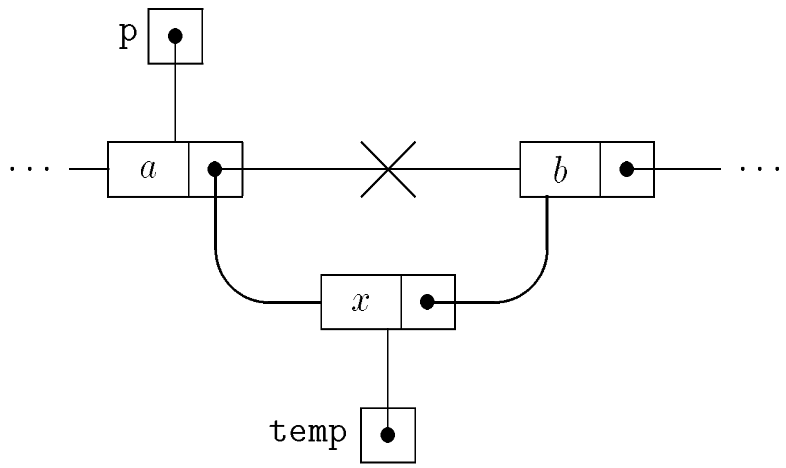
Liste après saisie de `4 3 2` :



... et après saisie supplémentaire de `1` :

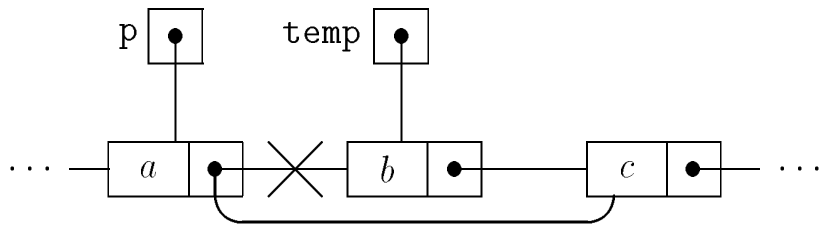


• Ajout d'un élément x (après la cellule d'adresse p)




```
p_cellule temp;
temp = (p_cellule) malloc(sizeof(cellule));
if(temp == NULL)
     erreur("tas saturé");
else{
     temp->element = x;
     temp->suivant = p->suivant;
     p->suivant = temp;
}
```

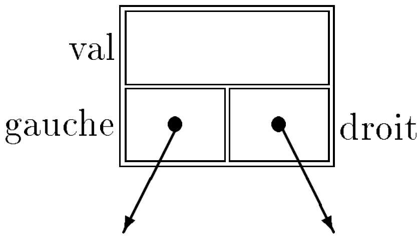
- Suppression d'un élément (après la cellule d'adresse `p`)




```
/* sans libération */
p->suivant = p->suivant->suivant;
...
/* avec libération */
p_cellule temp;
temp = p->suivant;
p->suivant = temp->suivant;
free(temp);
temp = NULL; /* pour la détection d'erreurs */
```

Arbres

- Plusieurs liens "`suivant`"
- Arbres binaires : nœuds avec au plus deux successeurs




```
typedef struct NOEUD {
     personne val;
     struct NOEUD * gauche;
     struct NOEUD * droit;
} noeud;

typedef noeud * arbre;
```

EXERCICES

1. **Position** : écrire une fonction qui donne la position d'un élément x donné (sa position) dans une liste chaînée L (et donne -1 si x n'apparaît pas dans la liste).
2. **Nombre** : écrire une fonction qui calcule le nombre d'occurrences d'un élément x dans une liste chaînée L.
3. **Sous-liste** : écrire une fonction qui détermine si $L2$ est une sous-liste de $L1$ ($L1$ et $L2$ sont des listes de même type).
4. **Suppression** : écrire une procédure qui supprime toutes les occurrences d'un élément x d'une liste chaînée.
5. **Libération** : écrire une procédure qui libère la mémoire occupée par une liste.
6. **Opérations sur listes** : écrire une unité pour la gestion de listes chaînées dont les cellules contiennent des flottants. On implémentera les opérations d'initialisation, de saisie, d'affichage, d'ajout en tête, de retrait en tête, de recherche d'élément, ...

Position, nombre

```
...
# include <float.h>
...
int position(double x, liste L){
   int p = 1;
   for( ; L; L = L->suivant, p++)
      if(abs(x - L->element) < DBL_EPSILON)
         return p;
   return -1;
}


int nombre(double x, liste L){
   int c = 0;
   for( ; L; L = L->suivant)
      if(abs(x - L->element) < DBL_EPSILON)
         c++;
   return c;
}
```

Sous-liste

```
...
# include <float.h>
...
boolean sous_liste(liste L2, liste L1){
   for( ; L2 && L1; L1 = L1->suivant)
      if(abs(L2->element - L1->element) < DBL_EPSILON)
         L2 = L2->suivant;
   if(L2 == NULL) return VRAI;
   else return FAUX;
}


boolean sous_lis(liste L2, liste L1) {
   if(L2 == NULL)
      return VRAI;
   else if(L1 == NULL)
      return FAUX;
   else if(abs(L2->element - L1->element) < DBL_EPSILON)
      return sous_lis(L2->suivant, L1->suivant);
   else
      return sous_lis(L2, L1->suivant);
}
```

Suppression

```
...
# include <float.h>
...
liste suppression(double x, liste L){
   if (L) {
      if(abs(x - L->element) < DBL_EPSILON) {
         liste temp = L->suivant;
         free(L);
         return suppression(x, temp);
      } else {
         L->suivant = suppression(x, L->suivant);
         return L;
      }
   } else return L;
}
```

Libération

```c
void liberation(liste L){
   if (L) {
      liste temp = L->suivant;
      free(L);
      liberation(temp);
   }
}
```

NOTE : on peut ajouter “`L = NULL`” après libération pour aider à la détection d'erreurs de programmation ou de logique. Sinon, la liste a de fortes chances de rester intacte tant qu'il n'y a pas d'autre allocation.

```c
...
liberation(L);  L = NULL;
...
```

Opérations sur listes

```
/************************* fichier "liste_entete.h" */

typedef struct CELLULE {
double element;
struct CELLULE * suivant;
} cellule;
typedef cellule * p_cellule;

typedef p_cellule liste;

typedef enum {FAUX, VRAI} boolean;

liste listevide();
liste saisie();
void  affichage(liste);
boolean vide(liste);
liste ajout_en_tete(double, liste);
liste suppr_en_tete(liste);
int position(double, liste);
int nombre(double, liste);
void liberation(liste);
liste suppression(double, liste);
boolean sous_liste(liste, liste);
```

```c
/******************************** unite "liste.c" */

# include <stdio.h>
# include <float.h>
# include "liste_entete.h"

liste listevide(){
   return NULL;
}

liste saisie(){
   int car; double x;
   liste L, temp;
   L = NULL;
   while(1){
      car = getchar();
      if(!(isdigit(car) || (car == '+') ||
           (car == '-') || (car == '\n'))) break;
      if(isdigit(car) || (car == '+') || (car == '-')) {
         ungetc(car, stdin);
         scanf("%lf", &x);
         temp = (liste) malloc(sizeof(cellule));
         temp->element = x;
         temp->suivant = L;
         L = temp;
      }
   }
   return L;
}
```

```
void  affichage(liste L){
   printf("LISTE : ");
   for( ; L; L = L->suivant)
      printf("%lf ", L->element);
   putchar('\n');
}

boolean vide(liste L){
   return L == NULL;
}

liste ajout_en_tete(double x, liste L){
   liste temp = (liste) malloc(sizeof(cellule));
   temp->element = x;
   temp->suivant = L;
   return temp;
}

liste suppr_en_tete(liste L){
   liste temp = L->suivant;
   free(L);
   return temp;
}
```