Plan

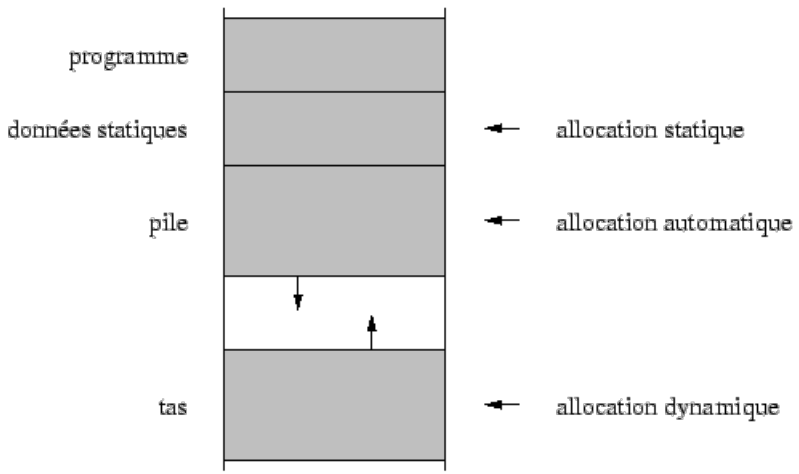
Plan

Allocation de mémoire

On appelle **allocation/libération de mémoire**, les mécanismes permettant à un programme de réserver/rendre au système une zone mémoire pour stocker des données.

On distingue 3 modes :

- l'allocation statique
- l'allocation automatique
- l'allocation dynamique



Modes d'allocation statique et automatique

- **Allocation statique** :
  - la zone mémoire est allouée lors du début de l'exécution du programme et libérée à la fin de l'exécution
  - les données ainsi placées sont dites permanentes
  - la zone est réservée dans le segment de données
  - c'est le mode des variables globales et variables locales `static`
- **Allocation automatique** (ou allocation dynamique dans la pile) :
  - la zone mémoire est allouée lors de la déclaration de la variable concernée et libérée lors du retour de fonction contenant cette variable (la fin de portée)
  - les données ainsi placées sont dites temporaires
  - la zone est réservée dans la pile
  - c'est le mode des variables locales et paramètres

Mode d'allocation dynamique

- **Allocation dynamique** (ou allocation dynamique dans le tas) :
  - la zone mémoire est allouée/libérée par l'appel à des fonctions spécifiques du langage : **malloc** et **free** fournies par la bibliothèque `stdlib.h`
  - les données sont dites (par abus de langage) "dynamiques"
  - la zone est réservée dans le tas
  - c'est un mode à utiliser quand la taille et/ou la durée d'utilisation des données à stocker dépend de paramètres externes au programme (de données lues par le programme)
  - la zone mémoire ainsi réservée n'est accessible que par pointeur

Fonction de réservation de mémoire

Memory allocation

```
void *malloc(size_t taille)
```

- `taille` est le nombre d'octets de la zone à allouer (`size_t` est un type entier non signé)
- l'adresse de la zone allouée est retournée :
  - c'est un pointeur sur un type indéterminé `void *`
  - la zone n'est pas initialisée
  - on ne peut pas faire d'arithmétique de pointeurs sur ce type
  - pour l'utiliser il faut **changer son type en un pointeur typé**

Exemple

```
void *p = malloc(4);
```

Attention

En cas d'échec d'allocation (plus assez d'espace mémoire disponible), ou de taille nulle

↪ le pointeur vide `NULL` est retourné.

Fonction de libération de mémoire

Libération

```
void free(void *p)
```

- `p` est l'adresse de début d'une zone mémoire précédemment allouée par un malloc et non déjà libérée

Exemple

```
free(p);
```

Attention

Une erreur d'exécution se produit si :

1. l'adresse n'est pas l'adresse de début d'une zone actuellement allouée : adresse ne correspondant pas à un malloc précédent ou déjà libérée,
2. et n'est pas le pointeur vide `NULL`.

Exemple d'allocation, manipulation, libération

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    void *p = malloc(5);          // reservation d'une zone de 5 octets
    if(p==NULL) {
        printf("allocation impossible\n");
        return 1;
    }
    p[0]='o'; // Erreur compil.: zone pointee par p non accessible avec ce type !
    char *pc = p;
    pc[0] = 'o';
    *(pc+1) = 'k';
    pc[2] = '\0';
    printf("%s\n",pc);                // Affiche ok
    printf("%s\n",(char *)p);         // Affiche ok
    int *pi = p;
    *pi = 12;
    pi[1]=23; // Erreur d'acces (poss. err. d'exec.): on deborde de 3 octets !
    printf("%d\n",*pi);               // Affiche 12
    printf("%d\n",*((int *)p));       // Affiche 12
    free(p);                          // libere la zone de 5 octets
    *pi=13;  // Erreur d'acces (poss. err. d'exec.): la zone n'est plus allouee !
    free(pc); // Erreur de restitution (err. d'exec.): zone non actuel. allouee !
    p = NULL; pc = NULL; pi = NULL;
    return 0;
}
```

Allocation dynamique en pratique

Lorsque l'on réserve une zone mémoire c'est pour y stocker des données d'un certain type. Un appel à **malloc** prend donc généralement la forme :

Syntaxe

```
type_d *pt = (type_d *)malloc(nb_d*sizeof(type_d))
```

- `type_d` est le type des données à stocker
- `nb_d` est le nombre de données à stocker

↪ `nb_d*sizeof(type_d)` donne directement le nombre d'octets à allouer.
↪ `pt` est manipulable comme un **tableau** de `type_d` de taille `nb_d`.

```c
#include <stdlib.h>

int main() {
    int *tab=(int *)malloc(10*sizeof(int));   // alloc.dyn. d'un tableau de 10 entiers
    ...           // utilisation de tab
    free(tab);                                // plus besoin du contenu de tab
    ...           // suite du programme sans tab
    return 0;
}
```

Les difficultés de l'allocation dynamique

La gestion "manuelle" de l'allocation mémoire demande une attention particulière :

- On manipule des **pointeurs** : il faut donc être sûr que la zone pointée a bien été allouée et que l'on ne déborde pas de la zone
- L'allocation n'est pas garantie : il faut vérifier que l'allocation s'est passée correctement

```
type_d *pt = (type_d *)malloc(nb_d*sizeof(type_d))
if(pt==NULL) {
    // traitement de ce cas d'echec de l'allocation
} else {
    // allocation reussie : on peut acceder a la zone pointee par pt
}
```

- La zone allouée n'est pas initialisée : il faut donc penser à l'initialiser "manuellement"

```
for (int i=0; i<nb_d; i++) {
    pt[i] = ....
}
```

Les difficultés de l'allocation dynamique

La gestion "manuelle" de l'allocation mémoire demande une attention particulière :

- La mémoire doit être restituée après utilisation pour éviter la **fuite de mémoire** : il faut donc libérer toute zone non utilisée
  - La fuite de mémoire est à traquer (pas de "ramasse-miettes" en `C`, contrairement à `Python`, `Java`, `Ocaml`...) ; dans certains cas (programmes à longue durée de vie, serveur par exemple) cela peut engendrer un arrêt non souhaitable du programme.
  - Attention, si on "perd" l'adresse d'une zone mémoire allouée avant de l'avoir libéré, on ne peut plus appeler `free` : il y aura donc fuite de mémoire.
- La mémoire ne doit pas être rendue plusieurs fois : il est conseillé de remettre immédiatement à `NULL` les pointeurs des zones libérées.

```
free(pt);
pt = NULL;
```

Plan

Création de tableaux dynamiques

L'allocation dynamique permet de créer des tableaux dynamiques. Leur création se fait en trois étapes :

1. déclaration d'un pointeur vers le type de données du tableau
2. allocation du nombre de cases désirées pour le tableau
3. vérification de la réussite de l'allocation

Attention

Pas de déclaration avec initialisation sur ces tableaux.

Exemple : création d'un tableau contenant les 10 premiers carrés.

```
...
int taille = 10;
int *tab;
tab = malloc(taille*sizeof(int));
if(tab==NULL) {
    printf("tableau non cree\n");
    return 1;
}
for(int i=0;i<taille;i++)
    tab[i]=i*i;
...
```

Tableaux dynamiques vs. tableaux statiques

Les cases de ces deux types de tableaux sont accessibles et modifiables de la même façon :

```
t[i] = t[j] + ...
*(t+i) = *(t+j) + ...
```

Mais, contrairement aux tableaux statiques, `t` la variable repérant un **tableau dynamique** est vraiment un pointeur :

- `t` est modifiable : on peut l'**affecter** ;
- `&t` est différent de `t` :
  - `t` a pour valeur l'adresse de la première case du tableau (une adresse du tas) : `t` est toujours égal à `&(t[0])`
  - `&t` a pour valeur l'adresse de la variable `t` (une adresse de la pile)
- `sizeof(t)` est toujours égale à 8, indépendamment de la taille du tableau .

Affectation et tableaux dynamiques

Affecter à un pointeur un tableau dynamique ne recopie pas le tableau, cela permet juste d'avoir un deuxième accès au tableau
↪ t1 et t2 pointent tous les 2 vers la même zone mémoire.

```
...
int *t1 = (int *)malloc(10*sizeof(int));
int *t2;
for(int i=0; i<10; i++) t1[i] = i;
t2 = t1;
t2[5] = 3 * t2[5];
printf("t1[5]=%d\n",t1[5]);
printf("t2[5]=%d\n",t2[5]);
...
```

Exécution :

```
t1[5]=12;
t2[5]=12;
```

Même si une allocation préalable est faite pour t2 `t2=(int *)malloc(10*sizeof(int));`, l'affectation ne recopiera pas le tableau, les deux pointeurs repèreront la même zone ! Et en plus, l'affectation engendrerait une fuite de mémoire !

Copie de tableau dynamique

Mais du coup comment copier un tableau dynamique ?

↪ **Comme pour les tableaux statiques** :

- par copie des cases une à une,
- après avoir réalisé une allocation mémoire du tableau copie !

```
...
int *t1 = (int *)malloc(10*sizeof(int));
int *t2;
for(int i=0; i<10; i++) t1[i] = i;
t2 = (int *)malloc(10*sizeof(int));
for(int i=0; i<10; i++) t2[i] = t1[i];
t2[5] = 3 * t2[5];
printf("t1[5]=%d\n",t1[5]);
printf("t2[5]=%d\n",t2[5]);
...
```

Exécution :

```
t1[5]=4;
t2[5]=12;
```

Tableau dynamique en sortie de fonction

Comme l'allocation/libération est décorrélée de la portée de la variable repérant un tableau, les tableaux dynamiques peuvent être retournés en **résultat de fonction** ou **paramètre de sortie** :

- un pointeur local à la fonction est déclaré pour accueillir l'adresse d'un tableau dynamique
- le tableau est dynamiquement créé par un `malloc` et son adresse est stockée dans le pointeur ;
- le pointeur permet de retourner cette adresse comme paramètre de sortie ou résultat ;
- en sortie de fonction, seul le pointeur est détruit (mais une copie du pointeur est retournée) et la zone mémoire dynamiquement allouée reste accessible.

Remarques :

- en plus de l'adresse de la zone mémoire, il sera parfois nécessaire de retourner la taille du tableau
- il faudra ultérieurement faire un `free` pour rendre la zone mémoire quand le tableau ne sera plus utilisé.

Exemple : une fonction de copie de tableau

```c
#include <stdlib.h>
#include <stdio.h>
#define TAILLE 10

int *copieTableau (int *t, int taille) {
    int *copie = (int *)malloc(taille*sizeof(int));
    for(int i=0; i<10; i++)
            copie[i] = t[i];
    return copie;
}

int main() {
    int *t1 = (int *)malloc(TAILLE*sizeof(int));  //ou   int t1[TAILLE];
    int *t2;
    for(int i=0; i<TAILLE; i++) t1[i] = i;
    t2 = copieTableau(t1,TAILLE);
    t2[5] = 3 * t2[5];
    printf("t1[i]=%d\n",t1[5]);
    printf("t2[i]=%d\n",t2[5]);
    free(t1);                          // pas de liberation si t1 statique
    free(t2);
    return 0;
}
```

Exécution :

```
t1[5]=4;
t2[5]=12;
```

Plan

Champs pointeurs dans une structure

Les champs pointeurs dans une structure permettent de stocker l'adresse de n'importe quelle donnée (scalaire, tableau, structure)

Syntaxe

```
struct nomStruct {
   type1 *champ1;
   type2 champ2;
};
```

- `champ1` contient l'adresse d'une donnée de type `type1`
- `type1` est n'importe quel type connu (`nomStruct` compris)

Exemple

```c
#include <stdio.h>

struct date {
    int j,m,a;
};
struct etudiant {
  char *nom;
  struct date naissance;
  struct etudiant *binome;
};

void afficheInfoBinome (struct etudiant e) {
    printf("L'etudiant %s (ne en %d) a pour binome %s (ne en %d)\n",
           quelles expressions pour obtenir l'affichage attendu ?);
}

int main() {
    struct etudiant e1={"Landut",{1,3,2002}};
    struct etudiant e2={"Demir",{12,1,1988}};
    e1.binome=&e2;
    e2.binome=&e1;
    afficheInfoBinome(e1);
    return 0;
}
```

Résultat attendu :

L'étudiant Landut (né en 2002) a pour binôme Demir (né en 1988)

Exemple

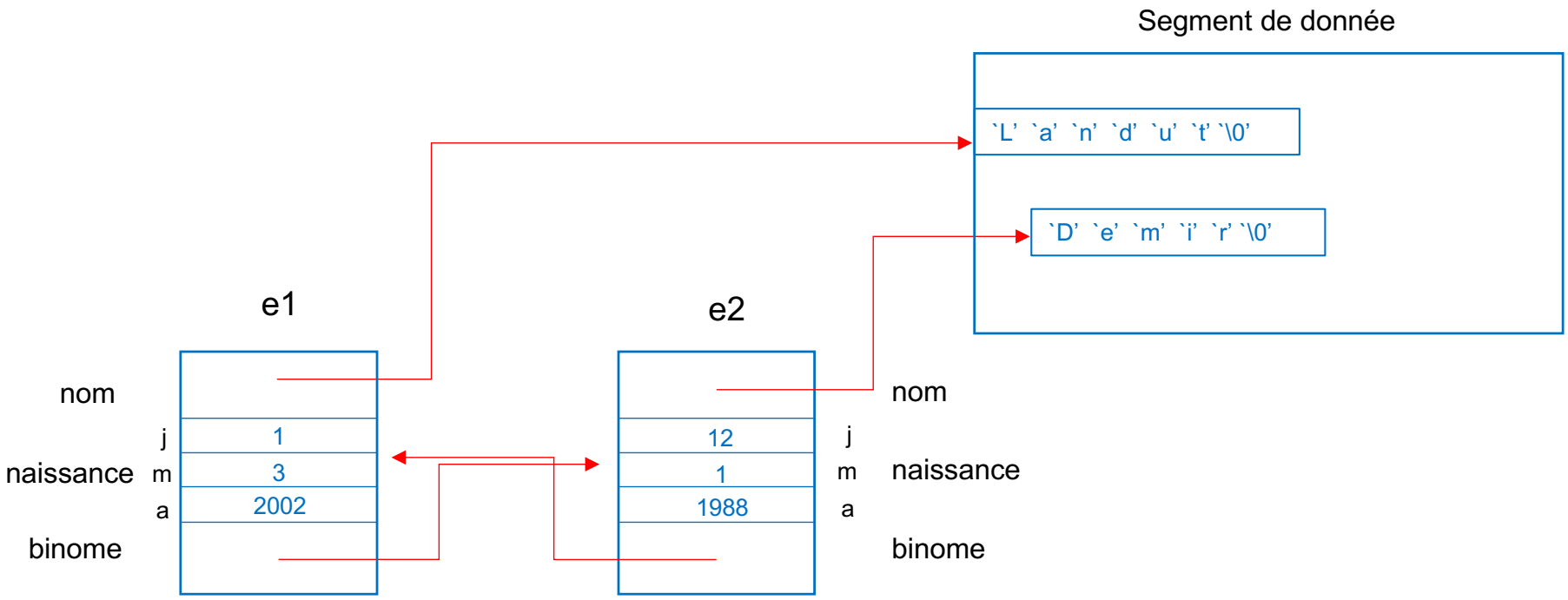
```c
#include <stdio.h>

struct date {
    int j,m,a;
};
struct etudiant {
  char *nom;
  struct date naissance;
  struct etudiant *binome;
};

void afficheInfoBinome (struct etudiant e) {
    printf("L'etudiant %s (ne en %d) a pour binome %s (ne en %d)\n",
           e.nom,e.naissance.a, e.binome->nom, e.binome->naissance.a);
}

int main() {
    struct etudiant e1={"Landut",{1,3,2002}};
    struct etudiant e2={"Demir",{12,1,1988}};
    e1.binome=&e2;
    e2.binome=&e1;
    afficheInfoBinome(e1);
    return 0;
}
```

Résultat attendu :
L'étudiant Landut (né en 2002) a pour binôme Demir (né en 1988)

Copie de structures avec champs pointeurs

Attention, seuls les membres sont recopiés (pas les valeurs pointées).

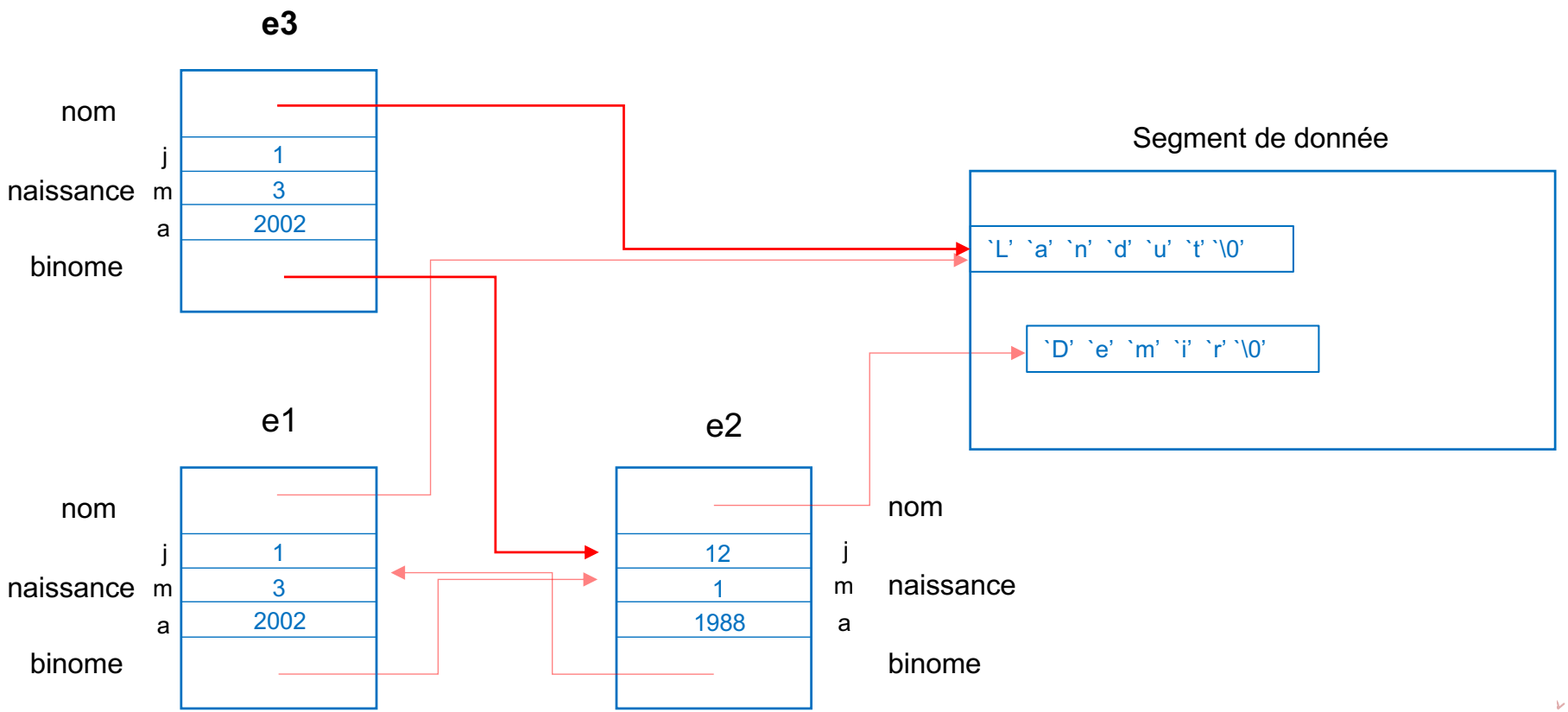
```
struct date { int j,m,a; };
struct etudiant { char *nom;  struct date naissance;  struct etudiant *binome; };
...
struct etudiant e3 = e1;
```



Copie de structures avec champs pointeurs

Attention, seuls les membres sont recopiés (pas les valeurs pointées).

```
struct date { int j,m,a; };
struct etudiant { char *nom;  struct date naissance;  struct etudiant *binome; };
...
struct etudiant e3 = e1;
```



Plan

Les structures chaînées

Les structures et pointeurs sont au cœur de l'implémentation de nombreuses **structures de données** : liste, file, pile, arbre, graphe...

Ces structures sont dites **chaînées** car elles sont représentées par de petits blocs de mémoire, des **cellules**, que l'on "chaîne" entre eux au moyen de **pointeurs** :

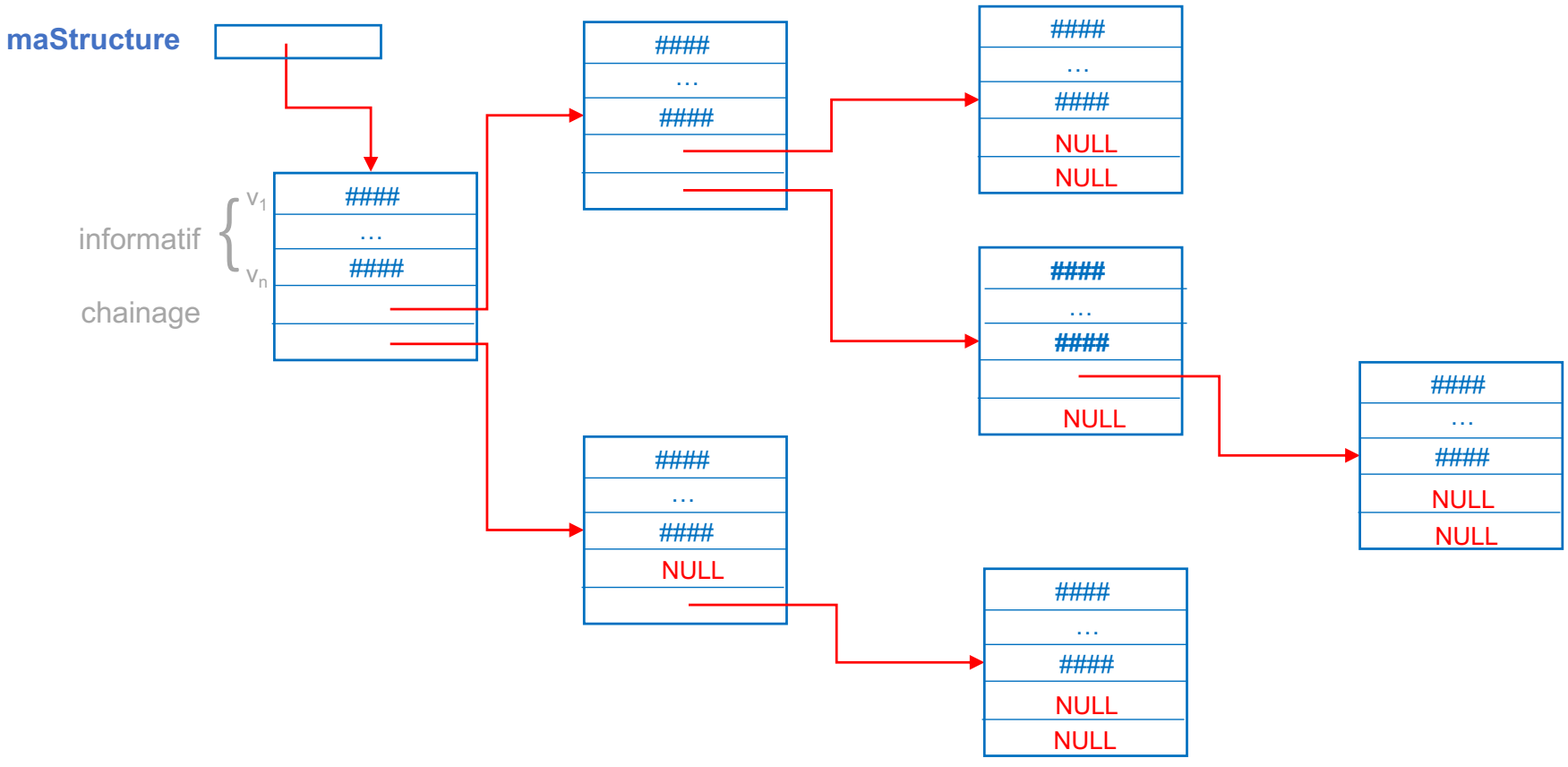
- Le pointeur d'une cellule repère l'adresse d'une autre cellule qui elle-même repère l'adresse d'autres cellules, et ainsi de suite...
- ces structures sont par nature à **allocation dynamique** : on alloue/libère les cellules à la demande.

Chaque cellule de cette structure est une **struct C** contenant :

- des membres précisant le contenu informatif de la cellule
- des membres permettant de chaîner cette cellule aux autres cellules

L'**accès à la structure** se fait en stockant dans un pointeur l'adresse d'une cellule.

Un exemple de structure d'arbre binaire



La liste chaînée

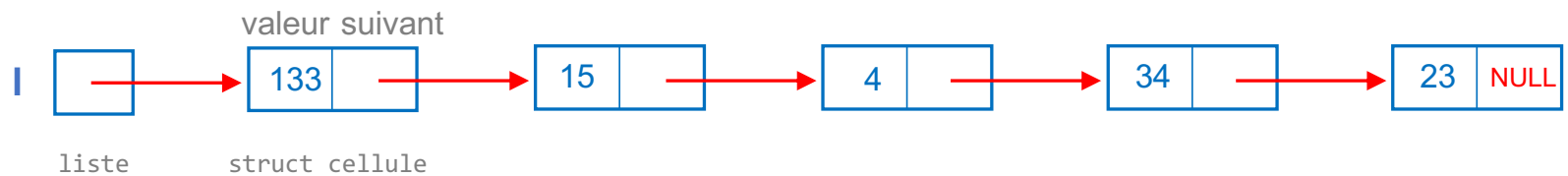
Comment représenter en C une séquence d'éléments de même type ?

- Un **élément** est un type quelconque : scalaire, structure, pointeur...
- Une première possibilité est le **type composé tableau** :
  - La taille est fixe ⇒ on ne peut ni ajouter, ni supprimer des elts
  - L'arithmétique des pointeurs permet d'indexer les éléments ⇒ on peut accéder en une opération unitaire à un élément
- Une deuxième possibilité est la **structure de données liste** :
  - La taille est variable ⇒ on peut ajouter et supprimer des elts par allocation dynamique
  - la structure est **chaînée** ⇒ on accède au i-ième élément en i opérations élémentaires
- Dans les 2 cas, on parcourt en $O(n)$ les n éléments de la séquence.

L'exemple de la liste chaînée d'entiers

Définition

```
struct cellule {
    int valeur;
    struct cellule *suivant;
};

typedef struct cellule *liste;
```



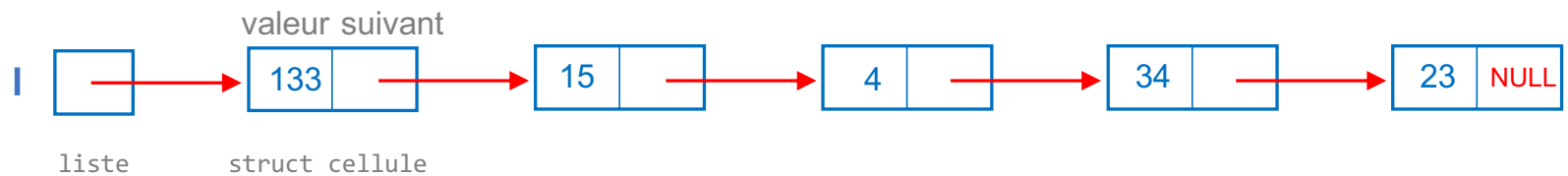
Soit la liste `l` de type `liste` ci-dessus, comment désigner :

- sa tête (son premier élément) ?

L'exemple de la liste chaînée d'entiers

Définition

```c
struct cellule {
    int valeur;
    struct cellule *suivant;
};

typedef struct cellule *liste;
```



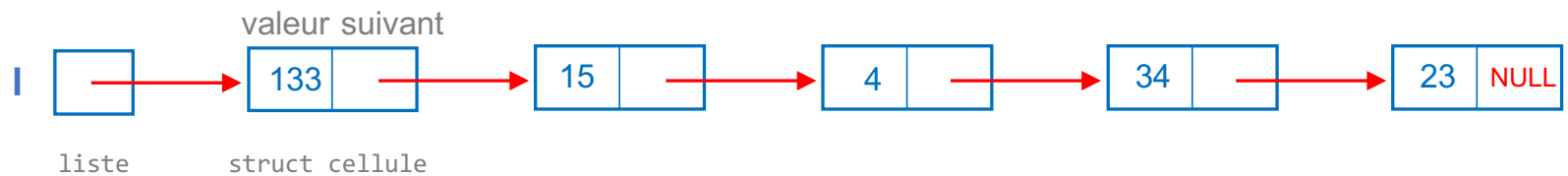
Soit la liste `l` de type `liste` ci-dessus, comment désigner :

- sa tête (son premier élément) ? `l->valeur`
- sa queue (la sous-liste sans son premier élément) ?

L'exemple de la liste chaînée d'entiers

Définition

```
struct cellule {
    int valeur;
    struct cellule *suivant;
};

typedef struct cellule *liste;
```



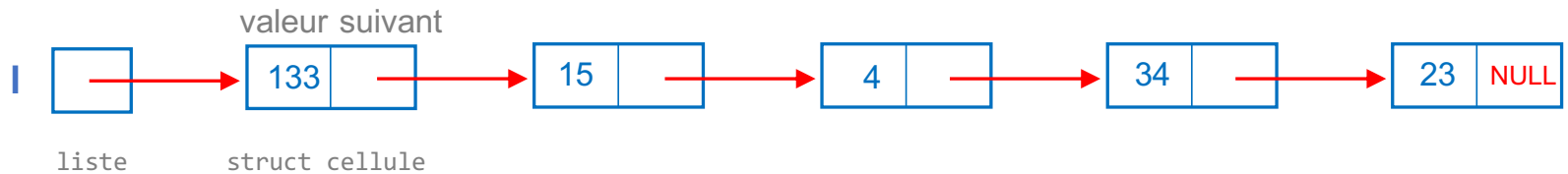
Soit la liste `l` de type `liste` ci-dessus, comment désigner :

- sa tête (son premier élément) ? `l->valeur`
- sa queue (la sous-liste sans son premier élément) ? `l->suivant`
- son deuxième élément ?

L'exemple de la liste chaînée d'entiers

Définition

```
struct cellule {
    int valeur;
    struct cellule *suivant;
};

typedef struct cellule *liste;
```



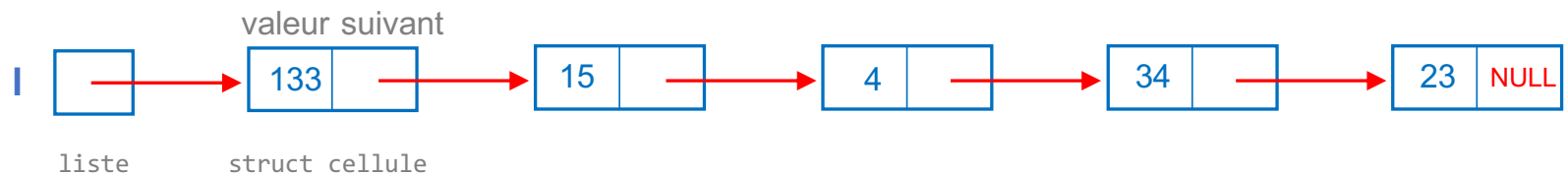
Soit la liste `l` de type `liste` ci-dessus, comment désigner :

- sa tête (son premier élément) ? `l->valeur`
- sa queue (la sous-liste sans son premier élément) ? `l->suivant`
- son deuxième élément ? `l->suivant->valeur`
- la liste vide ?

L'exemple de la liste chaînée d'entiers

Définition

```
struct cellule {
    int valeur;
    struct cellule *suivant;
};

typedef struct cellule *liste;
```



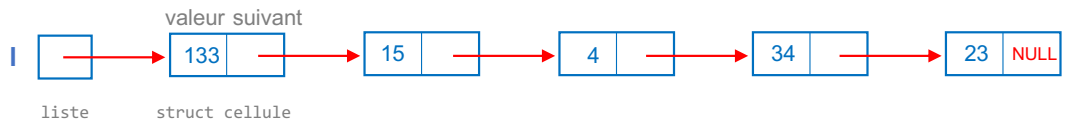
Soit la liste `l` de type `liste` ci-dessus, comment désigner :

- sa tête (son premier élément) ? `l->valeur`
- sa queue (la sous-liste sans son premier élément) ? `l->suivant`
- son deuxième élément ? `l->suivant->valeur`
- la liste vide ? `NULL`

Création de liste

Méthode

On démarre avec une liste vide et on ajoute progressivement les éléments à l'aide d'une fonction d'insertion en tête (l'équivalent du `::` `OCaml`).

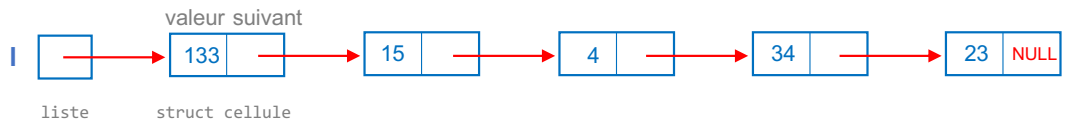



```
liste l=NULL; l=insereTete(23,l); l=insereTete(34,l); ...; l=insereTete(133,l);
```

Création de liste

Méthode

On démarre avec une liste vide et on ajoute progressivement les éléments à l'aide d'une fonction d'insertion en tête (l'équivalent du `::` `OCaml`).



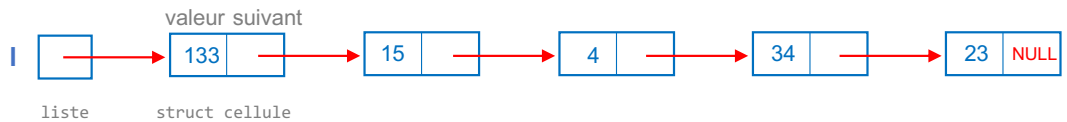

```
liste l=NULL; l=insereTete(23,l); l=insereTete(34,l); ...; l=insereTete(133,l);
```

```
liste insereTete(int element, liste lq) {
    liste li = (liste)malloc(sizeof(struct cellule));
    if(li==NULL) {
        printf("Erreur insertion : plus de memoire\n");
        exit(1);
    } else {
        li->valeur = element;
        li->suivant = lq;
        return li;
    }
}
```

Création de liste

Méthode

On démarre avec une liste vide et on ajoute progressivement les éléments à l'aide d'une fonction d'insertion en tête (l'équivalent du `::` `OCaml`).



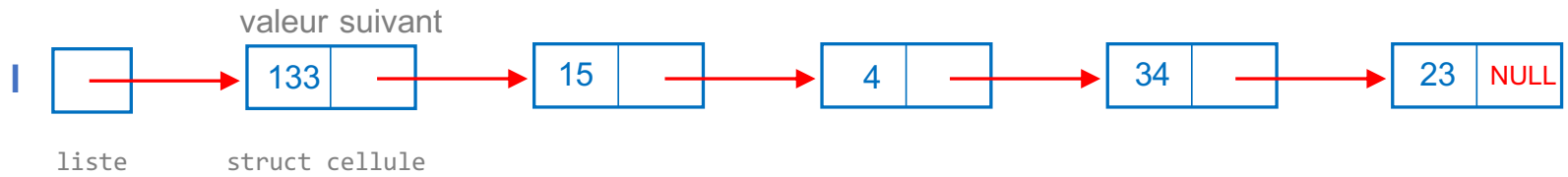

```
liste l=NULL; l=insereTete(23,l); l=insereTete(34,l); ...; l=insereTete(133,l);
```

```
liste insereTete(int element, liste lq) {
    liste li = (liste)malloc(sizeof(struct cellule));
    if(li==NULL) {
        printf("Erreur insertion : plus de memoire\n");
        exit(1);
    } else {
        li->valeur = element;
        li->suivant = lq;
        return li;
    }
}
```

Autre écriture :

```
liste l = insereTete(133,insereTete(15, ... ,insereTete(23,NULL)));
```

Longueur d'une liste




```
1 int longueur(liste li) {
2     int lg = 0;
3     while( condition ) {
4         lg++;
5
6     }
7     return lg;
8 }
```

Longueur d'une liste

Version itérative :

```
1 int longueur(liste li) {
2     int lg = 0;
3     while(li != NULL) {
4         lg++;
5         li = li->suivant;
6     }
7     return lg;
8 }
```

Longueur d'une liste

Version itérative :

```
1 int longueur(liste li) {
2     int lg = 0;
3     while(li != NULL) {
4         lg++;
5         li = li->suivant;
6     }
7     return lg;
8 }
```

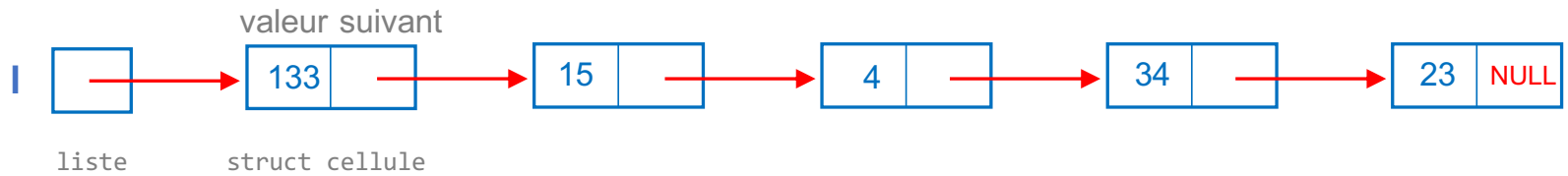
Version récursive :

```
int longRec(liste li) {
    if(li == NULL) return 0;
    else return 1 + longRec(li->suivant);
}
```

ou même

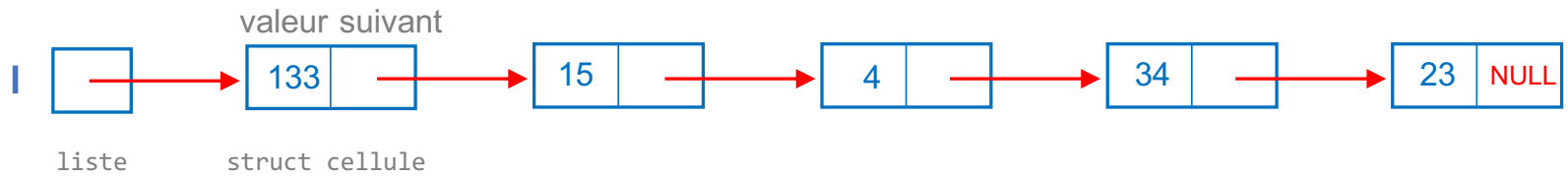
```
int longRec(liste li) { return li==NULL ? 0 : 1 + longRec(li->suivant); }
```

Nième élément de la liste



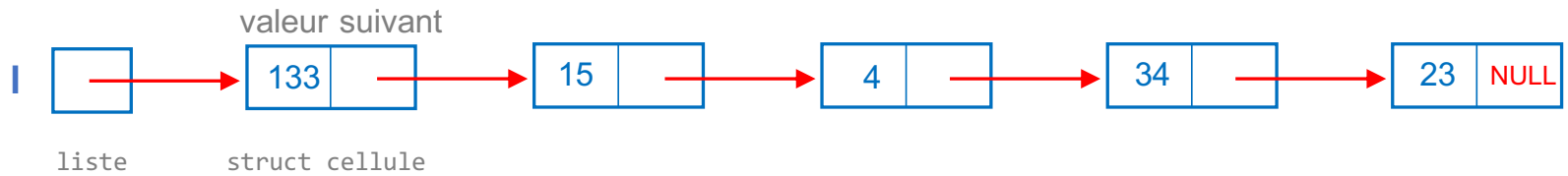

```
1  int nieme(int position, liste li) {
2      int i = 1;
3      while(i<position && condition) {
4
5
6      }
7      if( cas d'erreur ) {
8
9
10     }
11     else return ???;
12 }
```

Nième élément de la liste



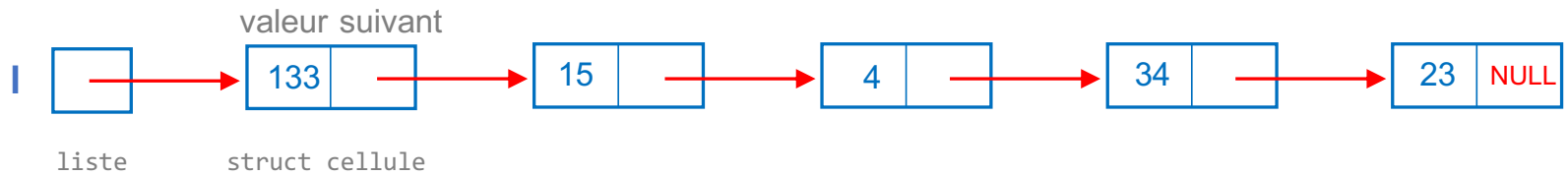

```
1  int nieme(int position, liste li) {
2      int i = 1;
3      while(i<position && li!=NULL) {
4          i++;
5          li = li->suivant;
6      }
7      if( cas d'erreur ) {
8
9
10     }
11     else return  ??? ;
12 }
```

Nième élément de la liste



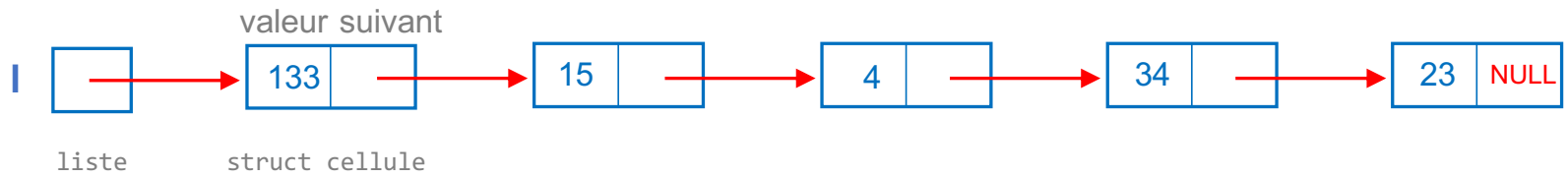

```
1  int nieme(int position, liste li) {
2      int i = 1;
3      while(i<position && li!=NULL) {
4          i++;
5          li = li->suivant;
6      }
7      if(position<1 || li==NULL) {
8          printf("Erreur nieme : position incorrecte\n");
9          exit(2);
10     }
11     else return ??? ;
12 }
```

Nième élément de la liste



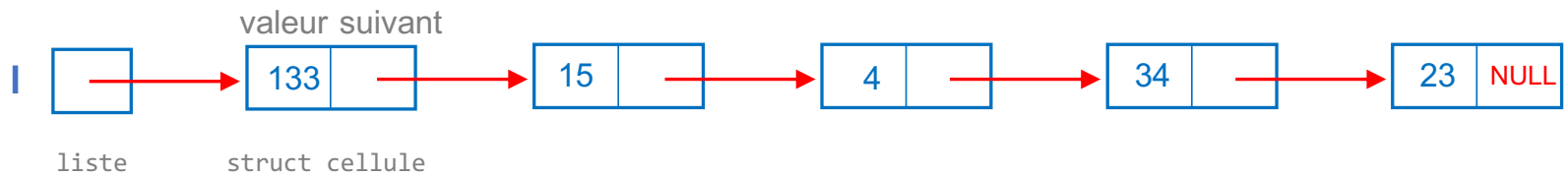

```
1  int nieme(int position, liste li) {
2      int i = 1;
3      while(i<position && li!=NULL) {
4          i++;
5          li = li->suivant;
6      }
7      if(position<1 || li==NULL) {
8          printf("Erreur nieme : position incorrecte\n");
9          exit(2);
10     }
11     else return li->valeur;
12 }
```

Affichage de liste




```
1 void afficher(liste li) {
2     printf("(");
3     while(condition) {
4         printf(" %d",???);
5         ???
6     }
7     printf(" )");
8 }
```

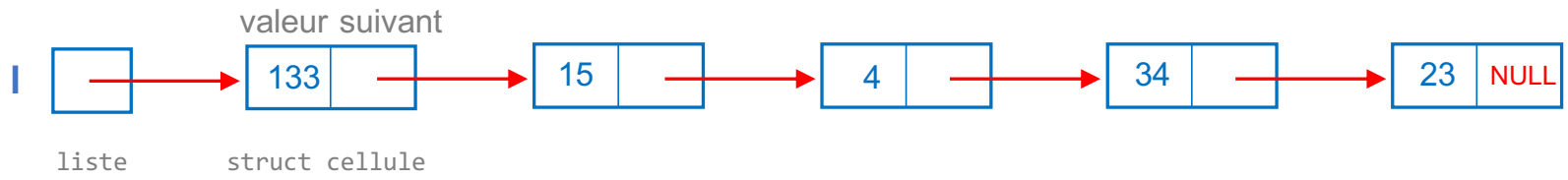
Affichage de liste




```
1 void afficher(liste li) {
2     printf("(");
3     while(li != NULL) {
4         printf(" %d",li->valeur);
5         li = li -> suivant;
6     }
7     printf(" )");
8 }
```

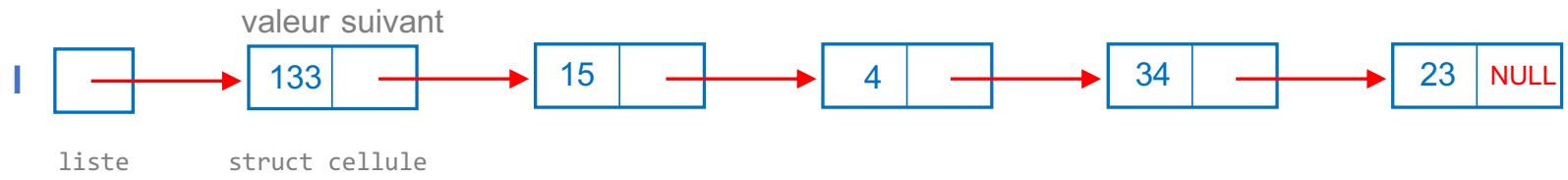
Résultat `afficher(l)` : (133 15 4 34 23)

Suppression en tête

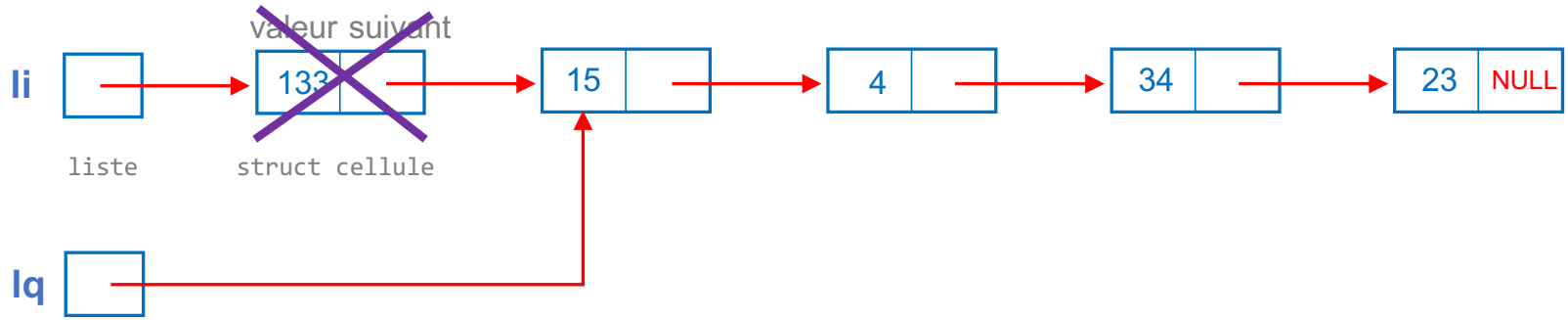



```
1 list supprimerTete(liste li) {
2     if(li == NULL) return NULL;
3
4     free(???);
5     return ???;
6 }
```

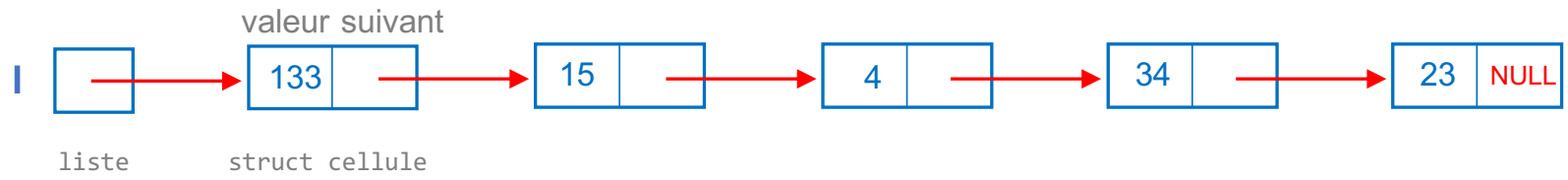
Suppression en tête



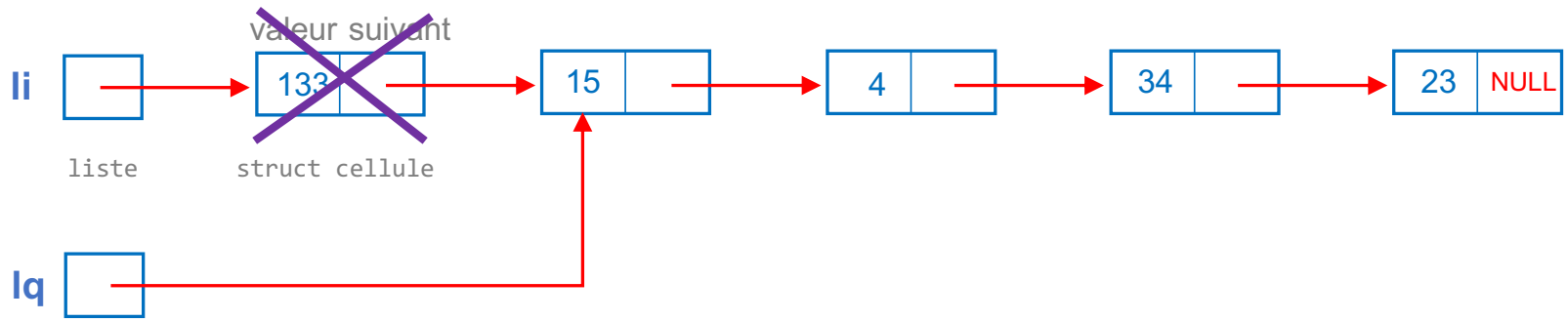

```
1  list supprimerTete(liste li) {
2      if(li == NULL) return NULL;
3
4      free(???);
5      return ???;
6  }
```



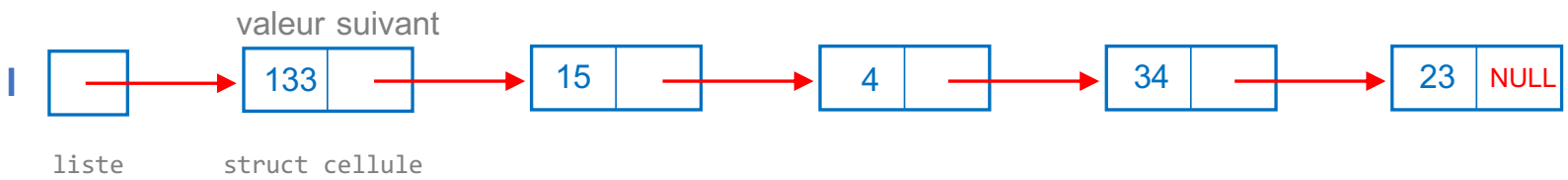
Suppression en tête




```
1 list supprimerTete(liste li) {
2     if(li == NULL) return NULL;
3     liste lq = li->suivant;
4     free(li);
5     return lq;
6 }
```

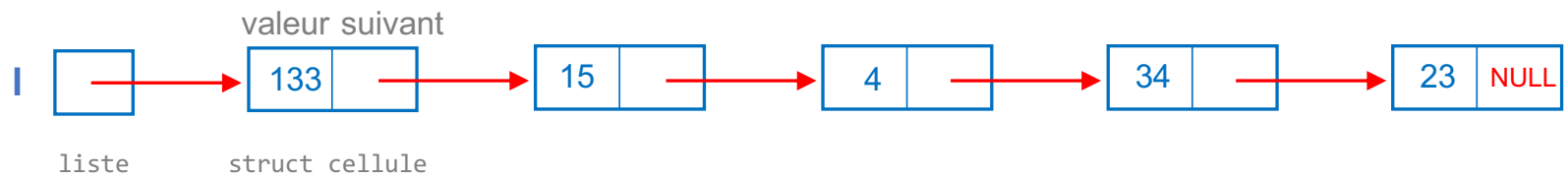


Insertion à une position donnée



Exemple : on veut insérer 27 à la position 3 de l

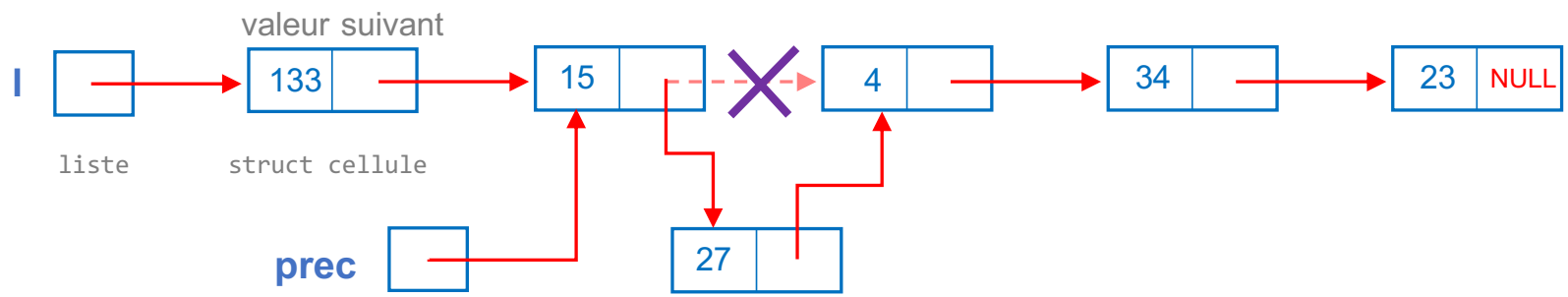
Insertion à une position donnée



Exemple : on veut insérer 27 à la position 3 de l

1. on recherche la cellule précédente
2. on crée une nouvelle cellule
3. on réalise le chaînage

↪ *l'insertion en position 1 est un cas particulier* ⇒ cf.`insererTete`



Insertion à une position donnée

```
list inserer(int element, int position, liste li) {
    if(position < 1) {
        printf("Erreur insertion : position incorrecte\n");
        exit(2);
    }
    if(position==1) return insererTete(element, li);
    int i = 1;
    liste prec = li;
    while(i<position && prec!=NULL) {
        prec = prec->suivant;
        i++;
    }
    if(prec==NULL) {
        printf("Erreur insertion : position incorrecte\n");
        exit(2);
    }
    liste nouvCellule = (liste)malloc(sizeof(struct cellule));
    if(nouvCellule==NULL) {
        printf("Erreur insertion : plus de memoire\n");
        exit(1);
    }
    else {
        nouvCellule->valeur = element;
        nouvCellule->suivant = prec->suivant;
        prec->suivant = nouvCellule;
        return li;
    }
}
```

Les opérations sur les listes sont chirurgicales

Attention

Les fonctions précédentes qui opèrent sur les listes sont trompeuses !

- Elles prennent une liste en paramètre et retourne une liste.
- Cependant il ne s'agit pas d'une nouvelle liste mais d'une **transformation** de la liste passée en paramètre dont la référence ne doit plus être utilisée !

↪ On parle d'opérations chirurgicales sur les structures chaînées !

On doit donc les utiliser correctement : `l = inserer(27,3,l) ;`

Alternativement, on aurait pu déclarer le paramètre `li` en Entrée-Sortie et renvoyer un booléen pour signaler la réussite ou l'échec de l'opération :

```
_Bool insererBis(int element, int position, liste *li);
```

Fonctions laissées en exercice

- Version Entrée-Sortie de insérer à une position donnée
- Supprimer le n-ième élément
- Modifier la valeur du n-ième élément
- Dupliquer une liste
- Concaténer deux listes
- Permuter deux cellules d'une liste
- Créer la liste des entiers pairs d'une liste donnée