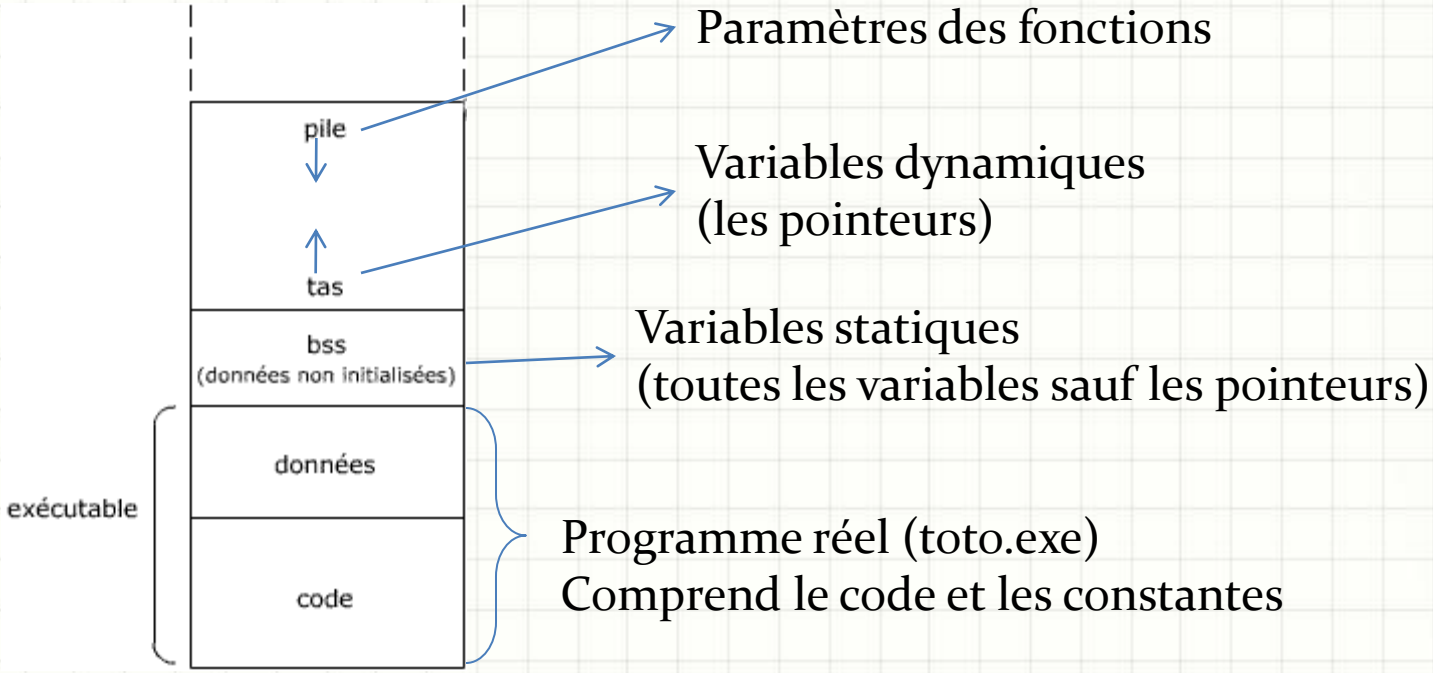
Le matériel

- Organisation de la mémoire

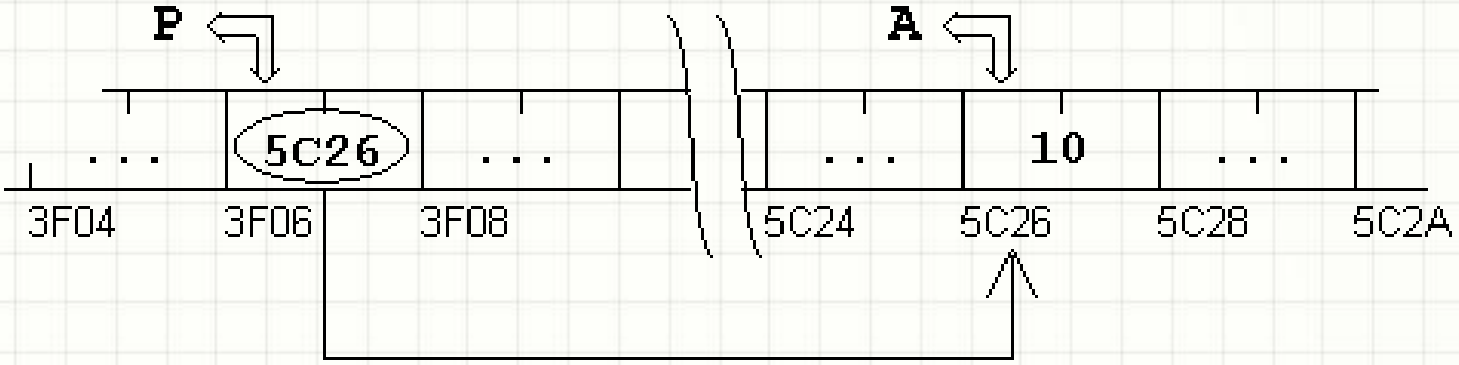


Pointeurs

- Un pointeur, c'est une variable dont la valeur est l'adresse d'une cellule de la mémoire
- Traduction :
  - Imaginons que la mémoire de l'ordinateur, c'est un grand tableau.
  - Un pointeur, c'est alors une variable de type **int** (un nombre) qui permet de se souvenir d'une case particulière.
- On comprend bien que le pointeur ne se soucis pas de savoir ce qu'il y a dans la case, mais bien de l'adresse de la case.

Pointeurs

- Outre l'utilisation d'une variable à travers son nom A, nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée **pointeur.** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.
- **Adressage indirect:** Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.
- **Exemple**
- Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



Allocation dynamique de la mémoire

Un tableau est alloué de manière statique : nombre d'éléments constant.

Alloué lors de la compilation (avant exécution)

problème pour déterminer la taille optimale, donnée à l'exécution

- surestimation et perte de place
- de plus, le tableau est un pointeur constant.

Il faudrait un système permettant d'allouer un nombre d'éléments connu seulement à l'exécution : c'est l'**allocation dynamique.**

Allocation dynamique

Faire le lien entre le pointeur non initialisé et une zone de mémoire de la taille que l'on veut.

On peut obtenir cette zone de mémoire par l'emploi de **malloc**, qui est une fonction prévue à cet effet.

Il suffit de donner à **malloc** le nombre d'octets désirés (attention, utilisation probable de sizeof), et **malloc renvoie un pointeur de type *void** sur la zone de mémoire allouée**.

Si malloc n'a pas pu trouver une telle zone mémoire, il renvoie NULL.

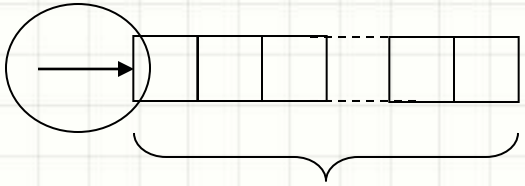
Appel par : malloc(nombre_d_octets_voulus);

Allocation dynamique

Symbolisation de l'effet de malloc:

si on utilise par exemple malloc(n), on a :

malloc() renvoie le pointeur



Zone de n octets

Pour accéder à cette zone, il faut impérativement l'affecter à un pointeur existant. On trouvera donc **toujours** malloc à droite d'un opérateur d'affectation.

Il ne faut pas oublier de transtyper le résultat de malloc() qui est de type void* en le type du pointeur auquel on affecte le résultat.

Allocation dynamique

```
nbElem = une valeur entière saisie au clavier
pt_int = (int *)malloc(nbElem*sizeof(int));
```

Détail de cette ligne : analyse de l'expression à droite de l'opérateur d'affectation :

(int *) : transtypage : car malloc() donne un pointeur void *, et pt_int est de type int *
malloc : appel à la fonction

nbElem*sizeof(int) : n'oublions pas que malloc reçoit un nombre d'octets a allouer ! Ici, on veut allouer nbElem élements, qui sont chacun de type int ! Or un int occupe plus d'un octet. Il occupe sizeof(int) octets !

Donc le nombre total d'octets à demander est : **nombre d'éléments * taille de chaque élément en octets**.

Allocation dynamique

Exemples d'utilisation :

Tableau de la taille requise, pas de perte de mémoire !

```
#include <stdio.h>
void main()
{
        int *pt_int;
        int nbElem;

        printf("combien d'elements dans le tableau ?:");
        scanf("%d",&nbElem);
        pt_int = (int *)malloc(nbElem*sizeof(int));
        if (pt_int == NULL)
            printf("L'allocation n'a pas fonctionné\n");
        else{
            /* saisie puis par exemple affichage du contenu du tableau */
          for (i=0;i<nbElem; i++)
              printf("%d ",pt_int[i]);
          free(pt_int);
    }
}
```

Libération de l'espace alloué

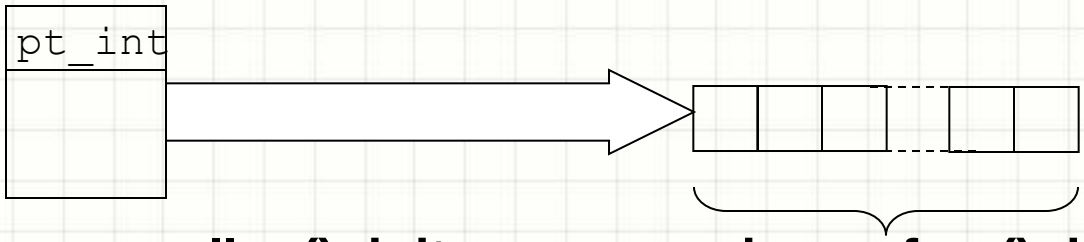
Lorsque la mémoire allouée dynamiquement n'est plus utile (le plus souvent, à la fin d'un programme, il est nécessaire de la libérer : la rendre disponible pour le système d'exploitation.

Fonction free qui réalise le contraire de malloc().

free(pointeur_vers_la_zone_allouée);

On ne peut libérer que des zones allouées dynamiquement : pas de free avec un tableau statique, même si le compilateur l'accepte.

```
free(pt_int);
```



À chaque malloc() doit correspondre un free() dans un programme !

Tableau de pointeurs

Puisqu'un pointeur est une variable comme une autre, on peut aussi les ranger dans un tableau: restriction, il faut que les pointeurs soient tous des pointeurs vers le même type.

Pour la déclaration :

type *tab[NB_ELEM] signifie : le contenu de chaque case du tableau est une valeur ayant le type donné, ou encore ; chaque case du tableau est un pointeur.

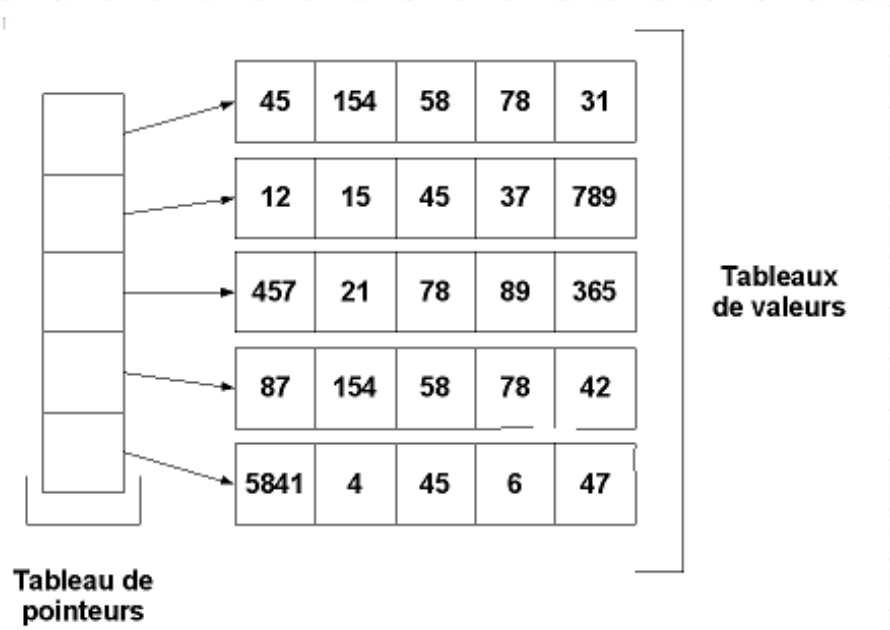
On peut s'en servir, par exemple, pour gérer un petit dictionnaire, c'est l'exemple que nous allons développer dans la suite du cours.

char *tab[100];

Tableau de pointeurs

Exemple :

int *tab1[100]; tab1 = tableau de 100 pointeurs, chacun pointant sur un entier.



char *tab2[50]; tab2= tableau de 50 pointeurs, chacun pointant sur un caractère,

Exemple : saisie d'un tableau à deux dimensions

```
#include <stdio.h>
const int nbElem=5;
void main()
{       int *pt_int[nbElem];
        int i, j, nbligne;
        printf(" Nombre de lignes ?:");
        scanf("%d",&nbligne);
       for (i=0; i<nbElem; i++)  {
         pt_int[i]= (int *)malloc(nbligne*sizeof(int));
            if (pt_int [i]== NULL)
                exit;
       } for (i=0;i<nbligne;i++)
                        for (j=0;j<nbElem;j++)
                            scanf("%d",&pt_int[i][j]);    }
        for (i=0;i<nbligne;i++) for (j=0;j<nbElem;j++)
                            printf("%d ",pt_int[i][j]);
}
```

Tableau de pointeurs

Libération de la mémoire

La libération de mémoire allouée doit se faire en parcourant le tableau, et en libérant pointeur par pointeur dans ce tableau :

```
……..
{  int *pt_int[nbElem];
   int i, j, nbligne;
   ……..
   for (i=0; i<nbElem; i++)  {
      …….
      free (pt_int[i])
  }
}
```

L'erreur à ne pas faire ! free(pt_int)

Pt_int est un type automatique, donc sera libéré automatiquement à la fin de la fonction dans laquelle il est déclaré.

Exercice

Ecrire un programme C qui permet de convertir un entier N quelconque en son équivalent binaire ?

44	2
0	22

22	2
0	11

11	2
1	5

5	2
1	2

2	2
0	1

1	2
1	0

Exemple :
44 (base 10) = 1 0 1 1 0 0 (base 2)

```
#include <stdio.h>
void main()
{
   int N, N1, *restes , i, taille=0;
   printf("donner un entier");
   scanf("%d",&N);
   N1=N;
   while (N1!=0)
  {N1 = N1/2; taille++;}
   restes = (int *)malloc(taille*sizeof(int));
   N1=N;    if (restes == NULL)
      printf("erreur de taille de tableau\n");
   else {       for (i=0;i<taille;i++) {
         restes[i] = N1 % 2;
         N1 = N1/2; }
      for (i=taille-1;i>=0;i--) printf("%d : ",restes[i]);
  free (restes);
} }
```

Exercice :

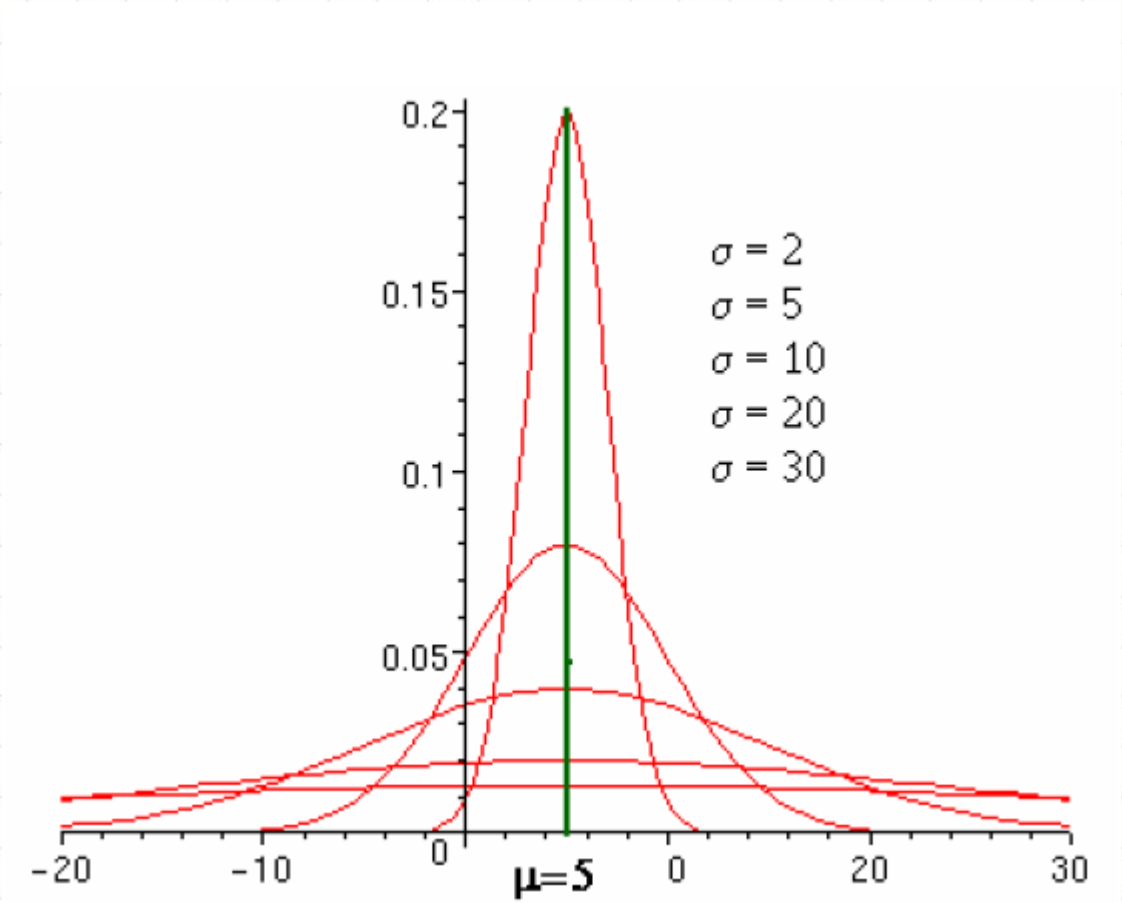
1- écrire une fonction sans type nommée saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ?

2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la moyenne ?

$$écart\ type = \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - moy)^2}$$

3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de

$$x \mapsto f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad \text{avec } \mu \in \mathbb{R} \text{ et } \sigma \in \mathbb{R}^+$$



Exercice :

1- écrire une fonction sans type saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ? **void sasie (int N, float *T);**

2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la moyenne ? **void calcul (int N, float *T, float *m, float *E);**

$$écart\ type = \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - moy)^2}$$

3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de

Exercice :

1- écrire une fonction sans type saisie qui permet de saisir un tableau de N réels, chaque valeur doit être comprise dans l'intervalle 0..20 ?

```
void saisie (int N, float *T)
{
  int i;
  for (i=0; i<N; i++)
  {
     do {
        printf("donner une valeur réelle : ");
        scanf("%f",&T[i]);   // scanf("%f",T+i); }
    while ((T[i]<0) || (T[i]>20));
  }
}
```

Exercice :

2- écrire une fonction calcul sans type permettant de calculer pour un tableau de réel de dimension N l'écart type et la moyenne ? $écart\ type = \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - moy)^2}$

```
void calcul (int N, float *T, float *m, float *E)
{
   int i; float s=0;
   for (i=0; i<N; i++) s = s+ T[i];
   *m = s/N;
    s = 0;
   for (i=0; i<N; i++)
     s = s+ (T[i] - *m)* (T[i] - *m);  //pow(T[i] - *m,2)
  *E = sqrt(s/N);
}
```

Exercice :

void sasie (int N, float *T);

void calcul (int N, float *T, float *m, float *E);

3- écrire la fonction main qui permet de 1) déclarer un pointeur sur réel 2) allouer dynamiquement un tableau Notes de réels de dimension N à saisir 3) appeler la fonction saisie pour remplir le tableau Notes 4) appeler la fonction calcul pour le calcul de l'écart type et la moyenne 5) afficher les résultats ?

```
#include <stdio.h>
#include <math.h>
main() {
  float *notes; int N; float M, E;
  printf("donner le nombre de notes ");
  scanf("%d", &N);
  notes = (float *)malloc(N*sizeof(float));
  if (notes == NULL)
    printf("erreur d'allocation de mémoire");
  else  { saisie(N, notes);
          calcul (N, notes, &M, &E);
          printf("moyenne = %f  ecart type = %f\n", M, E); } }
```

```
#include <stdio.h>
#include <stdlib.h>
int **matrice;
void alloue_matrice (int lignes, int colonnes);
int main() { int i,j;
    alloue_matrice(3,3);
    for (i=0;i<3;i++) {
      for(j=0;j<3;j++)
          printf("%d\n",matrice[i][j]);
       printf(("\n" );}
    return 0; }
void  alloue_matrice (int lignes, int colonnes)
{    int i,j;
      matrice = malloc(lignes*sizeof(int *));
      for (i=0;i<lignes;i++)
      matrice[i]= malloc(colonnes*sizeof(int));
      for (i=0;i<lignes;i++)
         for(j=0;j<colonnes;j++)
          matrice[i][j]=i*j;   }
```

Exercice void CODE (int N, int *T, int *C, int *D)

Soit un tableau **T** contenant **N** entiers, on cherche à compresser son contenu en appliquant le principe suivant : Chaque séquence d'éléments identiques est représentée par un couple (V, L) où V est la valeur qui se répète le long de la séquence et L est sa longueur.

Par exemple si nous appliquons cette méthode sur le tableau T suivant :

T

12	12	8	8	8	8	4	9	9	9	9	9	4	4

Alors en sortie nous obtiendrons un autre tableau :

C

12	2	8	4	4	1	9	5	4	2

Ainsi pour chaque indice **i** de **C**, nous obtiendrons :

- Dans C[i] la valeur d'une séquence d'entiers identiques dans T.
- Dans C[i + 1] la longueur de cette séquence.

1) Ecrire une fonction en langage C nommée **CODE** permettant de compresser un tableau **T** contenant **N** entiers en appliquant la méthode décrite en haut. Les paramètres de sortie de cette fonction sont le tableau compressé **C** et sa taille effective **D**.

2) Ecrire une fonction en langage C nommée **DECODE** permettant de décompresser un tableau **C** de taille **D**. Le paramètre de sortie de cette fonction est le tableau original **T**.

3) Ecrire la fonction main() incluant la saisie dynamique de T ?

Exercice :

1)Ecrire une fonction en langage C nommée **CODE** permettant de compresser un tableau **T** contenant **N** entiers en appliquant la méthode décrite en haut. Les paramètres de sortie de cette fonction sont le tableau compresse **C** et sa taille effective **D**.

```
void CODE (int N, int *T, int *C, int *D)
{ int i=0, j=0, L=1;
  while (i<N-1)
   {
        while ((T[i] == T[i+1]) && (i<N-1))
           { L++; i++;}
        C[j] = T[i];
        C[j+1] = L;
        i++;
        j = j +2;
        L=1;
    }
   *D = j;
}
```

Exercice :

2) Ecrire une fonction en langage C nommée **DECODE** permettant de décompresser un tableau **C** de taille **D**. Le paramètre de sortie de cette fonction est le tableau original **T**.

```
void DECODE (int T[], int C[], int D, int *N)  //(int *T, int *C, int D,N)
{
  int i=0, j, k=0;

  for (i=0; i<D; i+=2)
  {
      for (j=0;j<C[i+1]; j++)
        {  T[k]=C[i]; k++;  }
    }
   *N = k;
}
```

Exercice :

```
#include <stdio.h>
main() {
   int *T, *C; int N;
   printf(‘’donner le nombre de valeurs ‘’);
   scanf(‘’%d”, &N);
   T = (int *)malloc(N*sizeof(int));
   C = (int *)malloc(2*N*sizeof(int));
   if ((T == NULL) || (C==NULL))
      printf(“Erreur d’allocation de mémoire");
   else  { for (i=0;i<N; i++)
                {
                  printf("donner un entier");
                   scanf("%d",&T[i]);}
            CODE(N, T, C, &D);
            for (i=0;i<D; i++) printf("%d ",C[i]);
            DECODE(T, C, D, &N);
            for (i=0;i<N; i++) printf("%d ",T[i]);
            free(T); free( C );
         }
}
```

Exercice :

Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur **char** en réservant dynamiquement l'emplacement en mémoire pour les chaînes. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{ /* Déclarations */
char phrase[201]; /* chaîne pour l'introduction des données */
char *TEXTE[10]; /* Tableau des pointeurs sur les 10 chaînes */
char *PAIDE; /* pointeur d'aide pour l'échange des pointeurs */
int I,J; /* indices courants */
```

```
int main()
{ .....
 /* Saisie des données et allocation dynamique de mémoire */
 printf("Introduire 10 phrases :");
 for (I=0; I<10; I++) { printf("Phrase %d : ",I);
        scanf(" %s" , phrase);
        TEXTE[I] = malloc(strlen(phrase)+1);
        if (TEXTE[I]!=NULL)   strcpy(TEXTE[I], phrase);
                else { printf("Pas assez de mémoire \n"); exit(-1); } }
 printf("Contenu du tableau donné :");
 for (I=0; I<10; I++) printf(TEXTE[I]);
 /* Inverser l'ordre des phrases avec le pointeur PAIDE */
 for (I=0,J=9 ; I<J ; I++,J--)
  { PAIDE = TEXTE[I]; TEXTE[I] = TEXTE[J]; TEXTE[J] = PAIDE; }
 /* Afficher le tableau résultat */
 printf("Contenu du tableau résultat :");
 for (I=0; I<10; I++) printf(TEXTE[I]);
 return 0; }
```

Les principales fonctions d'allocation dynamiques sont (stdlib.h):

– **malloc** pour allouer un bloc de mémoire

– **calloc** pour allouer un bloc de mémoire et l'initialiser à 0

– **realloc** pour agrandir la taille d'un bloc de mémoire

– **free** pour liberer un bloc de mémoire.

Les prototypes de ces quatres fonctions sont les suivant :

« void * **malloc** (size_t size) » : alloue size octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

« void * **calloc** (size_t nmemb, size_t size) » : alloue la mémoire nécessaire pour un tableau de nmemb éléments, chacun d'eux représentant size octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.

« void * **realloc** (void * ptr, size_t size) » : modifie la taille du bloc de mémoire pointé par ptr pour l'amener à une taille de size octets. realloc()réalloue une nouvelle zone mémoire et recopie l'ancienne dans la nouvelle sans initialiser le reste.

« void **free** (void * ptr) » : libère l'espace mémoire pointé par ptr, qui a été obtenu lors d'un appel antérieur à malloc(), calloc() ou realloc().

Les Structures

- Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types long, char, int et double à la fois.
- Les structures sont généralement définies dans les *fichiers .h*, au même titre donc que les prototypes et les *define*. Voici un exemple de structure :

```
struct NomDeVotreStructure
{  int variable1;
   int variable2;
   int autreVariable;
   float nombreDecimal; };
```

- Une définition de structure commence par le mot-clé **struct**, suivi du nom de votre structure

Les Structures

- Exemples
- struct Coordonnées

```
{
    int x; // Abscisses
    int y; // Ordonnées
};
```

Structure utile pour créer un programme
De gestion de points dans le plan

- struct Personne

```
{
    char nom[100];
    char prenom[100];
    char adresse[1000];
     int age;
    int sexe; // Booléen : 1 = garçon, 0 = fille
};
```

Structure utile pour créer un programme
De gestion de carnet d'adresses

Les Structures – définition de types

- L'utilisation de **typedef** permet de faciliter les définitions de prototypes pour un code plus lisible. Il est ainsi possible de créer un type équivalent à une structure
- typedef struct {

```
    int x; // Abscisses
    int y; // Ordonnées
} Coordonnées ;
Coordonnées c1, c2, c3;
```

- typedef struct {

```
    char nom[100];
    char prenom[100];
    char adresse[1000];
     int age;
    int sexe;
} Personne ;
```

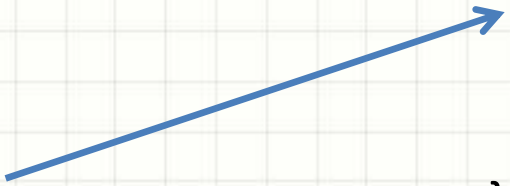
- typedef struct

```
{
   int jour;
  int mois;
   int annee;
} date;
```

Les Structures – définition de types

```
#include <stdio.h>
typedef struct {
        char nom[100];
        char prenom[100];
        char adresse[1000];
         int age;
        int sexe;
    } Personne ;
```

```
int main(int argc, char *argv[])
{
    Personne utilisateur, user;
    printf("Quel est votre nom ? ");
    scanf("%s", utilisateur.nom);
    printf("Votre prenom ? ");
    scanf("%s", utilisateur.prenom);
   scanf(‘’%d’’,&utilisateur.age);
    printf("Vous vous appelez %s %s",
utilisateur.prenom, utilisateur.nom);
   user = utilisateur;
  user.nom = utilisateur.nom NON
    return 0;
}
```



Affectation possible entre objets de même type

Les Structures – initialisation

- Pour les structures comme pour les variables, tableaux et pointeurs, il est vivement conseillé de les initialiser dès leur création pour éviter qu'elles ne contiennent « n'importe quoi »!
- Pour rappel :
- **une variable** : on met sa valeur à 0 (cas le plus simple) ;
- **un pointeur** : on met sa valeur à NULL (préféré à 0) ;
- **un tableau** : on met chacune de ses valeurs à 0.
- Pour les structures, l'initialisation va un peu ressembler à celle d'un tableau. En effet, on peut faire à la déclaration de la variable :
- Coordonnees point = {0, 0}; Cela définira, dans l'ordre, point.x = 0 et point.y = 0.
- Personne utilisateur = {"", "", "", 0, 0};

Les Structures – Exemple

Ecrire un programme C qui définit une structure *point* qui contiendra les deux coordonnées d'un point du plan. Puis lit deux points et affiche la distance entre ces deux derniers.

```
#include<stdio.h>

#include<math.h>
struct point{        typedef struct {
float x;  float y;        float x;  float y;
};                        } point;
int main()
{   struct point A,B;        point A, B;
    float dist;
    printf("Entrez les coordonnées du point A:\n");
    scanf("%f%f",&A.x,&A.y);
    printf("Entrez les coordonnées du point B:\n");
    scanf("%f%f",&B.x,&B.y);
    dist = sqrt( (A.x-B.x)*(A.x-B.x) + (A.y-B.y)*(A.y-B.y) );
    printf("La distance entre les points A et B est: %.2f\n",dist);
    return 0;
}
```

Les Structures – Exercice 1

Ecrire un programme C qui définit une structure *etudiant* où un étudiant est représenté par son nom, son prénom et une note. Lit ensuite une liste de 100 étudiants entrée par l'utilisateur et affiche les noms de tous les étudiants ayant une note supérieure ou égale à *10* sur *20*.

Les Structures – Exercice 1

Ecrire un programme C qui définit une structure *etudiant* où un étudiant est représenté par son nom, son prénom et une note. Lit ensuite une liste de 100 d'étudiants entrée par l'utilisateur et affiche les noms de tous les étudiants ayant une note supérieure ou égale à *10* sur *20*.

```
#include<stdio.h>
#include<stdlib.h>
struct etudiant{
char nom[20]; char prenom[20]; int note;
};
int main()
{ struct etudiant t[100];
int i;
for(i=0;i<100;i++) {
printf("donnez le nom, prenom et la note de l'etudiant %d:\n",i+1);
scanf("%s%s%d",t[i].nom,t[i].prenom,&t[i].note); }
for(i=0;i<100;i++) {
if(t[i].note>=10)
printf("%s %s\n",t[i].nom,t[i].prenom); }
return 0; }
```

```
typedef  struct {
char nom[20]; char prenom[20]; int note;
} etudiant;

etudiant t[100];
```

Pointeurs sur structures

- Intéressant pour deux raisons : 1) savoir comment envoyer un pointeur de structure à une fonction pour que celle-ci puisse modifier le contenu de la variable. 2) **Création de listes chaînées : les piles et les files.**

```
 1) int main(int argc, char *argv[])
{   Coordonnees monPoint;
    initialiserCoordonnees(&monPoint);
    return 0;
}
void initialiserCoordonnees(Coordonnees  *point)
{   (*point).x = 0;   (*point).y = 0; }
```

Ou bien { **point->x = 0**; **point->y = 0;** }

Les Structures – pointeurs

```
typedef struct {
        char nom[100];
        char prenom[100];
        char adresse[1000];
         int age;
        int sexe;
      } Personne ;
```

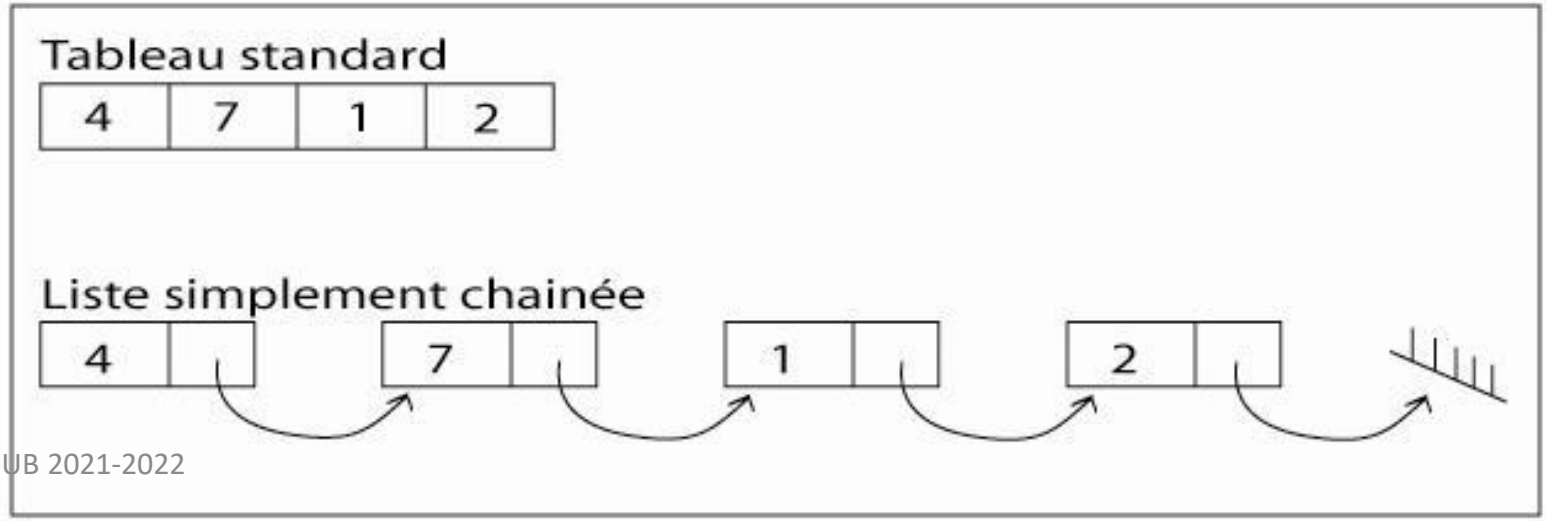
```
int main(int argc, char *argv[])
{
   Personne *user1, user2;
   user1=(Personne *)malloc(sizeof(personne));
   if (user1== NULL) exit(1);
   Else {
   printf("Quel est votre nom ? ");
   scanf("%s", (*user1).nom);   // ou user1->nom
   printf("Votre prenom ? ");
   scanf("%s", user1->prenom);
    scnaf(‘’%d’’,user1->age);
   printf("Vous vous appelez %s %s", user1->prenom, user1->nom);
  user2 = *user1;
  printf("Vous vous appelez %s %s", user2.prenom, user2.nom); }
   return 0;
}
```

Les listes chaînées

- Lorsque on crée un tableau, les éléments de celui-ci sont placés de façon contiguë en mémoire. Pour pouvoir le créer, il vous faut connaître sa taille.
- Si on veut par exemple supprimer un élément au milieu du tableau, il faut recopier les éléments temporairement, réallouer de la mémoire pour le tableau, puis le remplir à partir de l'élément supprimé.
- En bref, ce sont beaucoup de manipulations coûteuses en ressources.
- Une liste chaînée est différente dans le sens où les éléments de votre liste sont répartis dans la mémoire et reliés entre eux par des pointeurs.
- On peut ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière.

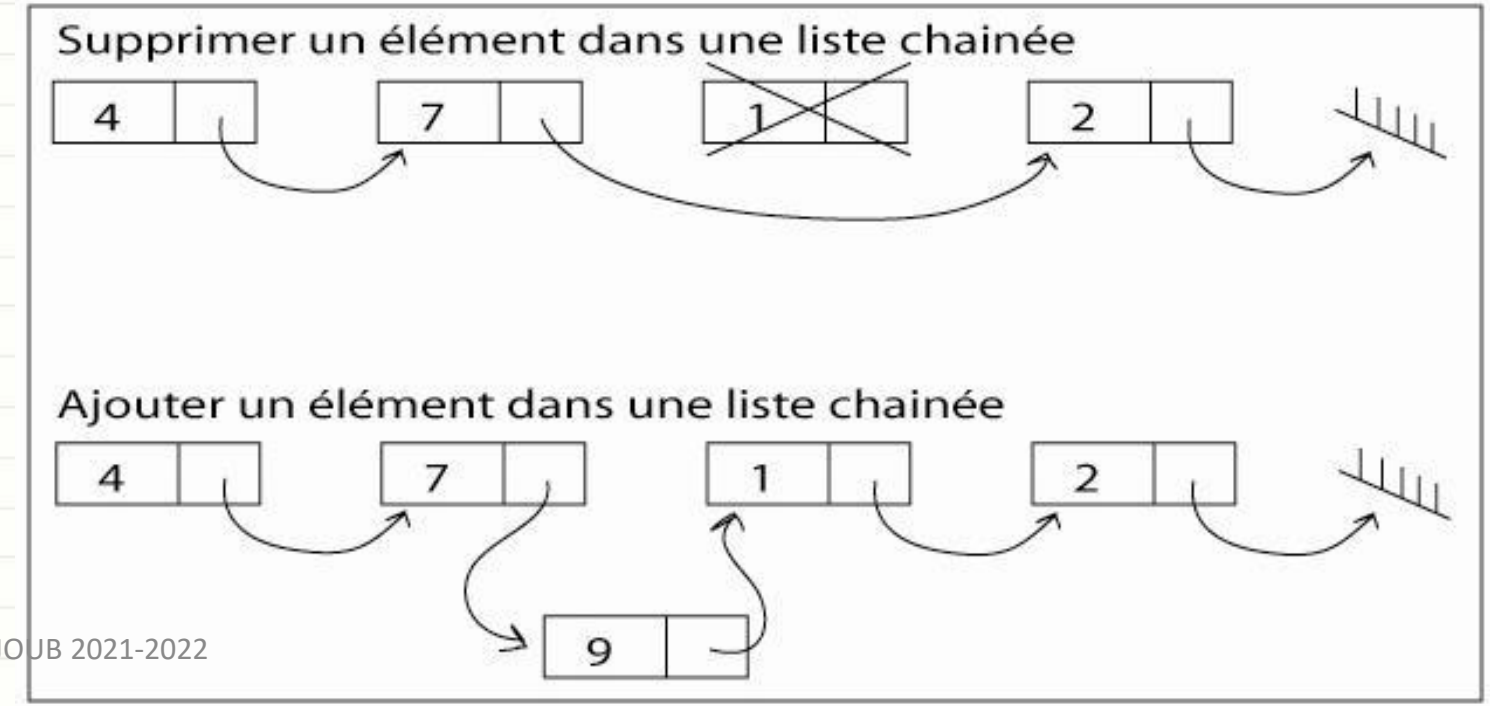
Les listes chaînées

- Dans une liste chaînée, la taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet. Il est en revanche impossible d'accéder directement à l'élément i de la liste chainée. Pour déclarer une liste chaînée, il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est donc à spécifier.
- Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînée sans avoir à recréer la liste en entier, mais en manipulant simplement leurs pointeurs.



Les listes chaînées

- Chaque élément d'une liste chaînée est composé de deux parties :
  - la valeur que vous voulez stocker,
  - l'adresse de l'élément suivant, s'il existe.
    S'il n'y a plus d'élément suivant, alors l'adresse sera NULL, et désignera le bout de la chaîne.
- Ajout et la suppression d'un élément d'une liste chaînée.



Les listes chaînées : déclaration

- Quel type sera l'élément de la liste chaînée ?
- On peut créer des listes chaînées de n'importe quel type d'éléments : entiers, caractères, structures, tableaux, voir même d'autres listes chaînées... Il est même possible de combiner plusieurs types dans une même liste.

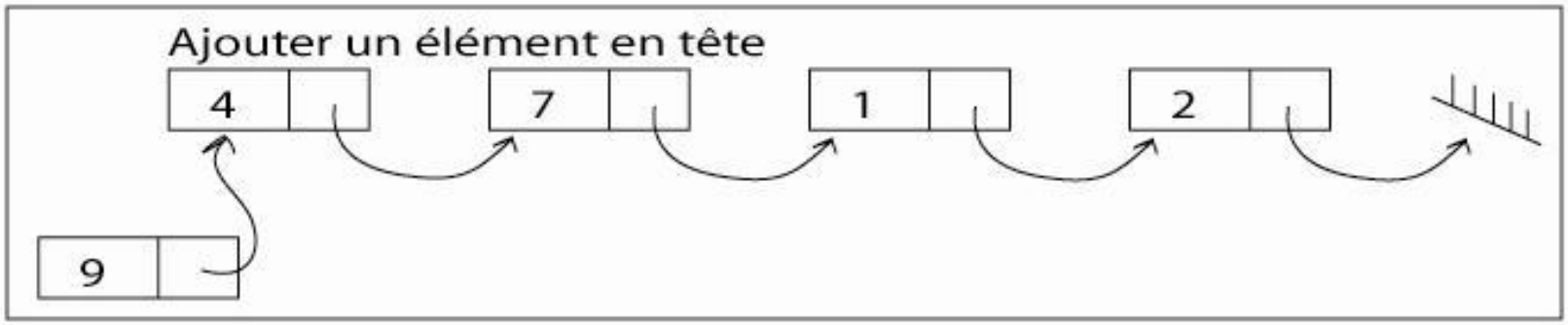
```
#include <stdlib.h>
typedef struct element
{
    int val;
    struct  element *next;
} element;
 element* llist;
```

Les listes chaînées : déclaration

```
#include <stdlib.h>
#include <stdlib.h>
        typedef struct element
        {   int val;
            struct element *next;
        } element;
         typedef element* llist;
 int main(int argc, char **argv)
{    /* Déclarons 3 listes chaînées de façons différentes mais équivalentes */
    llist ma_liste1 = NULL;
    element *ma_liste2 = NULL;
    struct element *ma_liste3 = NULL;
     return 0;    }
```

Les listes chaînées : Ajout en tête

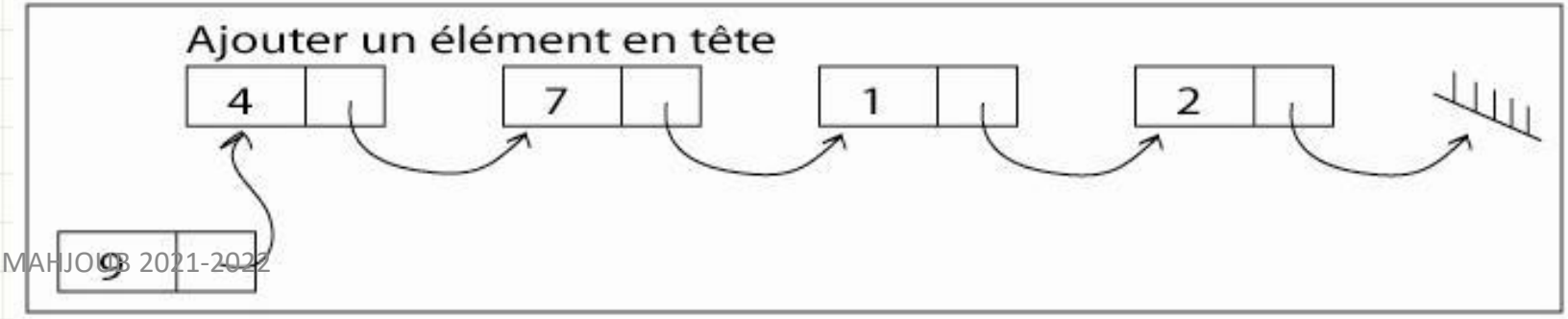
- Lors d'un ajout en tête, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis pour terminer, raccorder cet élément à la liste passée en paramètre.
- Lors d'un ajout en tête, on devra donc assigner à next l'adresse du premier élément de la liste passé en paramètre. Visualisons tout ceci sur un schéma :



- C'est l'ajout le plus simple des deux. Il suffit de créer un nouvel élément puis de le relier au début de la liste originale. Si l'original est , (vide) c'est NULL qui sera assigne au champ next du nouvel element. La liste contiendra dans ce cas-là un seul élément.

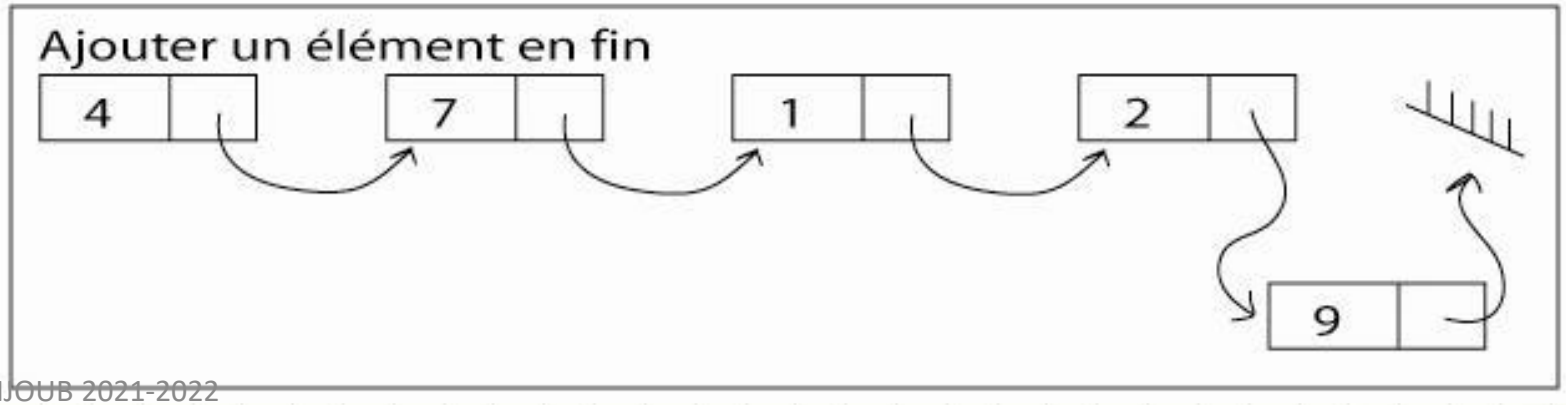
Les listes chaînées : Ajout en tête

```
llist ajouterEnTete(llist tete_liste, int valeur)
{   /* On crée un nouvel élément */
    element *   nouvelElement;
   nouvelElement = (element *)malloc(sizeof(element));
     /* On assigne la valeur au nouvel élément */
    nouvelElement->val = valeur; //(*nouvelElement).val = valeur;
     /* On assigne l'adresse de l'élément suivant au nouvel élément */
    nouvelElement->next = tete_liste;
     /* On retourne la nouvelle liste, le pointeur sur le premier élément */
    return nouvelElement;
}
```



Les listes chaînées : Ajout en fin de liste

Cette fois-ci, c'est un peu plus compliqué. Il nous faut tout d'abord créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à NULL. En effet, comme cet élément va terminer la liste nous devons signaler qu'il n'y a plus d'élément suivant. Ensuite, il faut faire pointer le dernier élément de liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur **element** qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si NULL est assigné à son champ next.



Les listes chaînées : Ajout en fin de liste

```
llist ajouterEnFin(llist liste, int valeur)
{   element* nouvelElement = malloc(sizeof(element));
    nouvelElement->val = valeur;
    nouvelElement->next = NULL;
     if(liste == NULL)  return nouvelElement;
    else  {
        /* Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on
      indique que le dernier élément de la liste est relié au nouvel élément */
        element* temp=liste;
        while(temp->next != NULL)   temp = temp->next;
        temp->next = nouvelElement;
        return liste;     }
}
```

Les listes chaînées : Affichage

```
void afficherListe(llist liste)
{   element *tmp = liste;
   /* Tant que l'on n'est pas au bout de la liste */
   while(tmp != NULL)
   {    /* On affiche */
      printf("%d ", tmp->val);
      /* On avance d'une case */
      tmp = tmp->next;}
 }
```

La seule chose à faire est de se déplacer le long de la liste chaînée grâce au pointeur tmp. Si ce pointeur tmp pointe sur NULL, c'est que l'on a atteint le bout de la chaîne, sinon c'est que nous sommes sur un élément dont il faut afficher la valeur.

Les listes chaînées : Application

- Utiliser les trois fonctions que nous avons vues jusqu'à présent :
  - ajouterEnTete (tete, val)
  - ajouterEnFin (tete, val)
  - afficherListe (tete)

- Vous devez écrire la fonction main permettant de remplir et afficher la liste chaînée ci-dessous. Vous ne devrez utiliser qu'une seule boucle for.

- 10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10

Les listes chaînées : Application

```
int main()
{
    llist ma_liste = NULL;
    int i;
     for(i=1;i<=10;i++)
    {
        ma_liste = ajouterEnTete(ma_liste, i);
        ma_liste = ajouterEnFin(ma_liste, i);
    }
    afficherListe(ma_liste);
    // il reste à libérer l'espace mémoire occupé par la liste
    return 0;
}
```

```
typedef struct element
{   int val;
    struct element *next;
} element;
typedef  element* llist;
```

Les listes chaînées : Application

- On veut écrire une fonction qui renvoie 1 si la liste est vide, et 0 si elle contient au moins un élément.

```
int estVide(lliste liste)
{   if(liste == NULL)
        return 1;
    else        return 0;
}
```

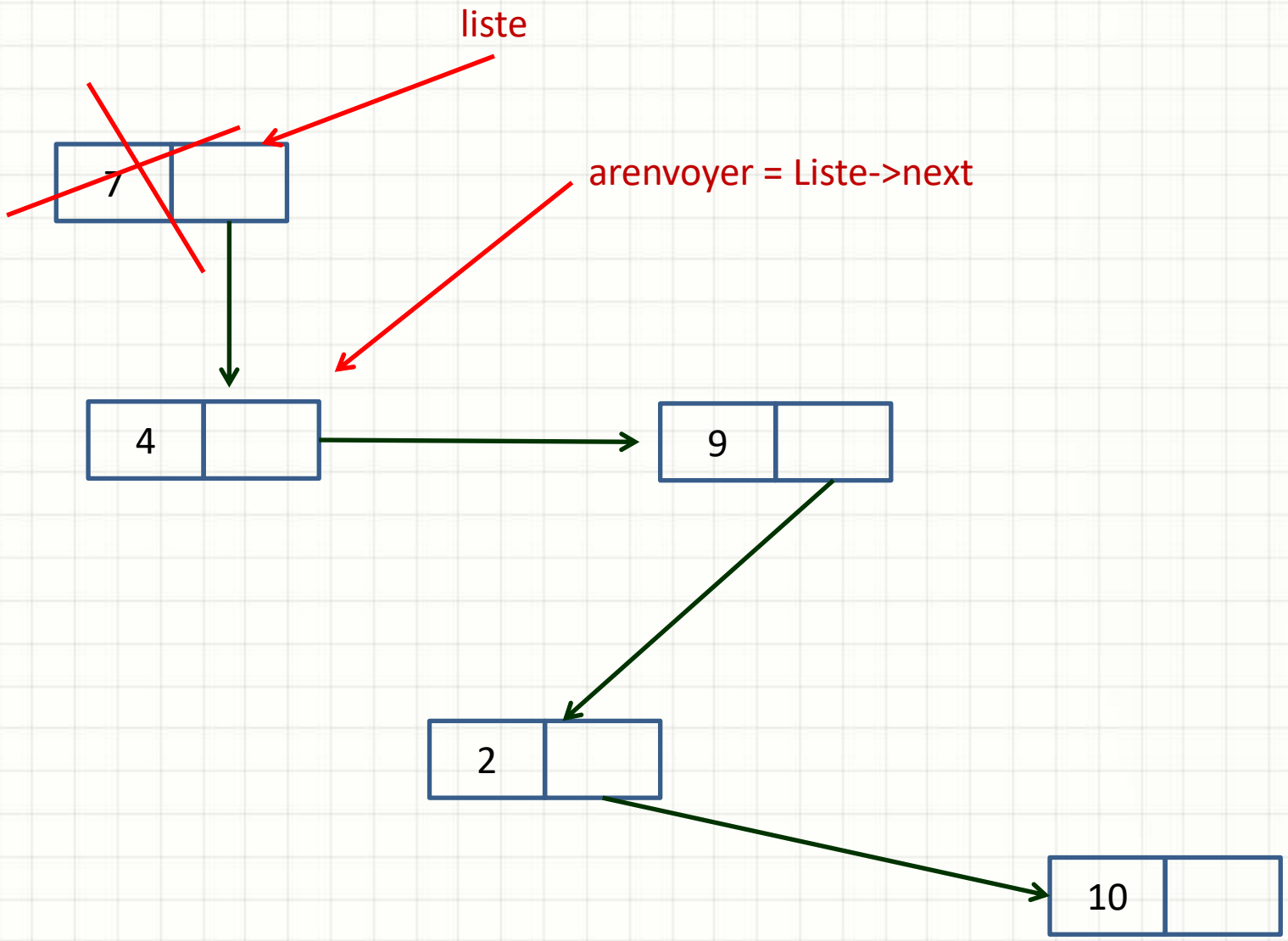
- Utilisation dans main():

```
.....
if(estVide(ma_liste))
        printf("La liste est vide");
    else
        afficherListe(ma_liste);
```

Ecriture condensée :

```
int estVide(llist liste)
{
    return (liste == NULL)? 1 : 0;
}
```

Effacer le premier élément

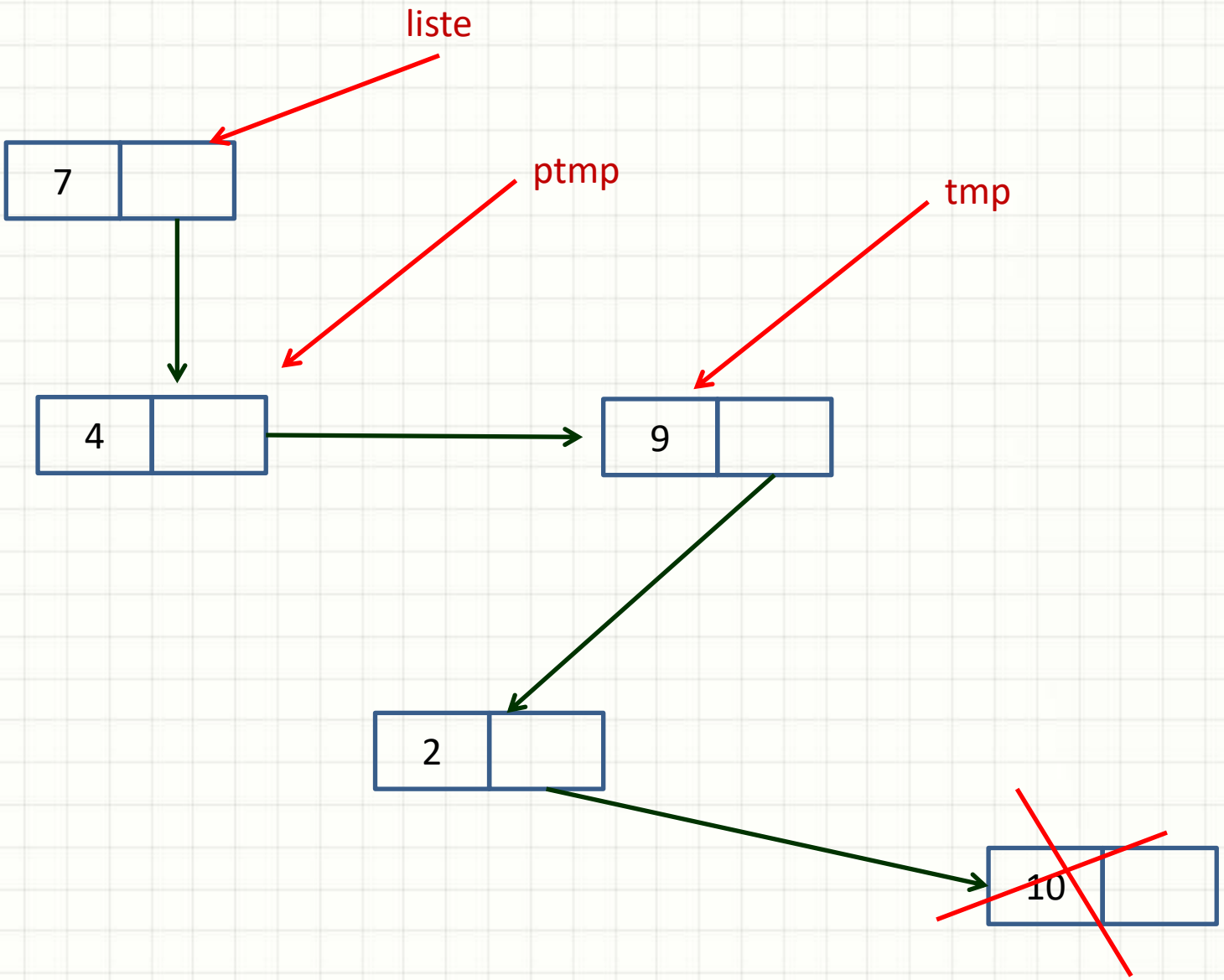


Les listes chaînées : Suppression en tête

```
llist supprimerElementEnTete(llist liste)
{   if(liste != NULL)
   { /* Si la liste est non vide, on se prépare à renvoyer l'adresse
        de  l'élément en 2ème position */
     llist aRenvoyer = liste->next;
     /* On libère le premier élément */
     free(liste);
     /* On retourne le nouveau début de la liste */
     return aRenvoyer;    }
   else    return NULL;
}
```

Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de la récupérer.

Effacer le dernier élément



Les listes chaînées : Suppression en fin de liste

```
llist supprimerElementEnFin(llist liste)
{   if(liste == NULL)       return NULL;
   if(liste->next == NULL)   {
         free(liste);  return NULL;   }
   element* tmp = liste;
   element* ptmp = liste;
   while(tmp->next != NULL)   {
      /* ptmp stock l'adresse de tmp */
      ptmp = tmp;       /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
      tmp = tmp->next;   }
    ptmp->next = NULL;
   free(tmp);
   return liste; }
```

Suppression en fin de liste (avec commentaires)

```
llist supprimerElementEnFin(llist liste)
{/* Si la liste est vide, on retourne NULL */
   if(liste == NULL)       return NULL;
   /* Si la liste contient un seul élément */
   if(liste->next == NULL)    {
         /* On le libère et on retourne NULL (la liste est maintenant vide) */
          free(liste);  return NULL;    }
   /* Si la liste contient au moins deux éléments */
   element* tmp = liste;
   element* ptmp = liste;
   while(tmp->next != NULL)    {
      /* ptmp stock l'adresse de tmp */
      ptmp = tmp;      /* On déplace tmp (mais ptmp garde l'ancienne valeur de tmp */
      tmp = tmp->next;    }
   /* A la sortie de la boucle, tmp pointe sur le dernier élément, et ptmp sur  l'avant-dernier.
    On indique que l'avant-dernier devient la fin de la liste  et on supprime le dernier élément */
    ptmp->next = NULL;
   free(tmp);
   return liste; }
```

Recherche d’un élément dans une liste

But : renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra NULL

```
llist rechercherElement(llist liste, int valeur)
{   element *tmp=liste;
   /* Tant que l'on n'est pas au bout de la liste */
   while(tmp != NULL)
   {   if(tmp->val == valeur)
      /* Si l'élément a la valeur recherchée, on renvoie son adresse */
         return tmp;
      tmp = tmp->next;
   }   return NULL;
}
```

Compter le nombre d'occurrences d'une valeur

On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.

```
int nombreOccurences(llist liste, int valeur)
{   int i = 0;
    if(liste == NULL)     return 0;
    /* Sinon, tant qu'il y a encore un élément ayant la val = valeur */
   while((liste = rechercherElement(liste, valeur)) != NULL)
   {   liste = liste->next;
       i++;
   }
   /* Et on retourne le nombre d'occurrences */
   return i;
}
```

Compter le nombre d'occurrences d'une valeur

On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée.

```
int nombreOccurences(llist liste, int valeur)
{   int i = 0;
    if(liste == NULL)     return 0;
     llist tmp=liste;
    while(tmp!= NULL)
    {   if (tmp->val==valeur)    i++;
        tmp = tmp->next;
    }
    /* Et on retourne le nombre d'occurrences */
    return i;
}
```

Recherche du i-ème élément

```
llist element_i(llist liste, int indice)
{   int i; element *tmp=liste
   /* On se déplace de i cases, tant que c'est possible */
   for(i=0; i<indice && liste != NULL; i++)
      tmp = tmp->next;
   /* Si l'élément est NULL, c'est que la liste contient moins de i éléments */
   if(tmp == NULL)
      return NULL;
   else
   {
      /* Sinon on renvoie l'adresse de l'élément i */
      return tmp;   }
}
```

Exercice :

Fonction pour effacer tous les éléments d'une liste ?

1- version séquentielle
2- version récursive

```
llist effacer (llist liste)
llisit p;    // p est un pointeur
p->val    <==>    (*p).val
p->next   <==>   (*p).next
```

```
typedef struct element
{   int val;
    struct element *next;
} element;
 typedef element* llist;
```

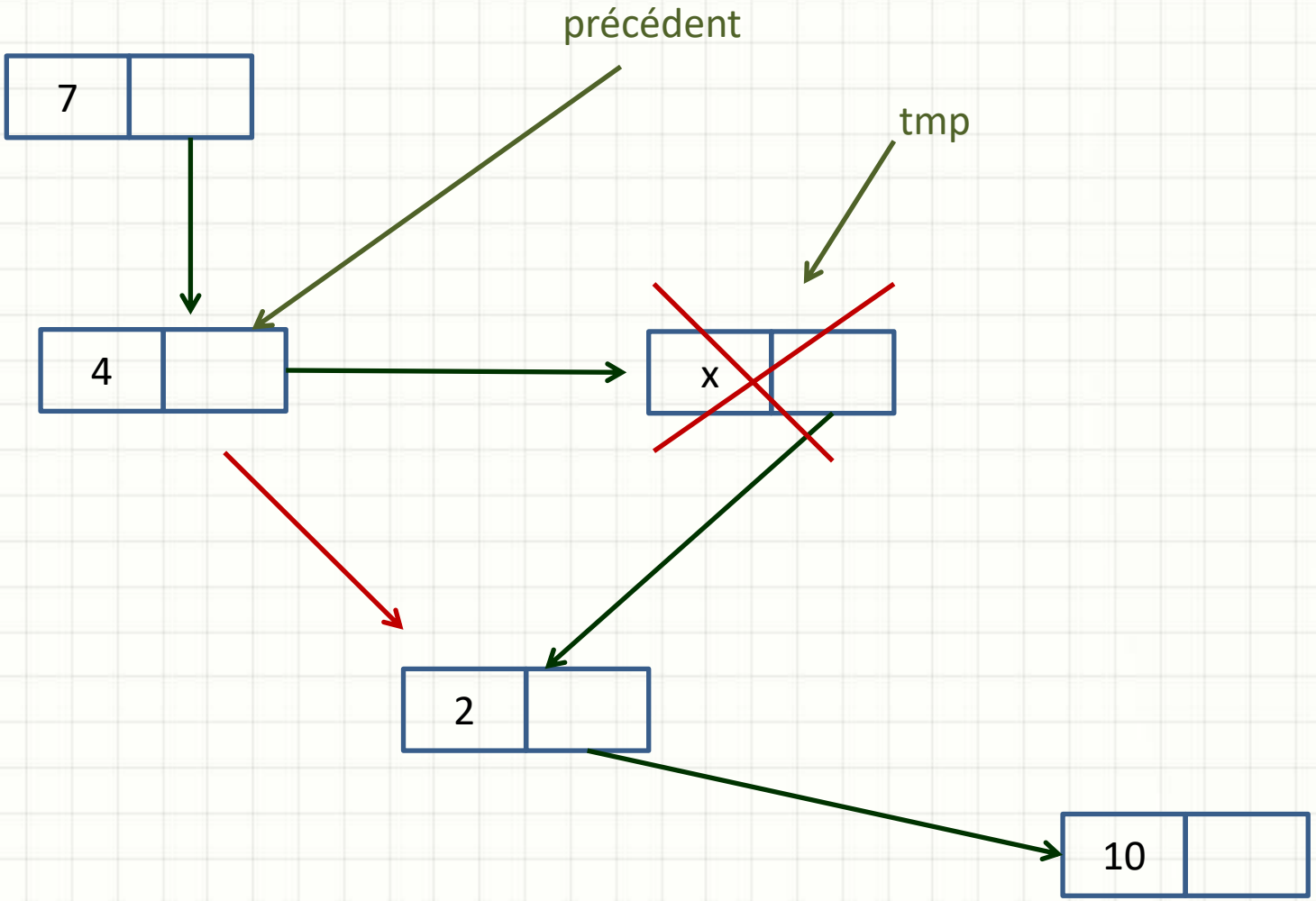
Effacer tous les éléments d'une liste (ver 1)

```
llist effacerListe(llist liste)
{  element* tmp = liste;
   element* tmpnext;
  while(tmp != NULL)
   {  /* On stocke l'élément suivant pour pouvoir ensuite avancer */
      tmpnext = tmp->next;
      /* On efface l'élément courant */
      free(tmp);
      /* On avance d'une case */
      tmp = tmpnext;
   }
   /* La liste est vide : on retourne NULL */
   return NULL;
}
```

Effacer tous les éléments d'une liste (ver 2)

```
llist effacerListe(llist liste)
{  if(liste == NULL)
   { /* Si la liste est vide, il n'y a rien à effacer, on retourne
        une liste vide i.e. NULL */
        return NULL;    }
   else
   { /* Sinon, on efface le premier élément  */
        element *tmp;
        tmp = liste->next;
        free(liste);
        return effacerListe(tmp);
   }
}
```

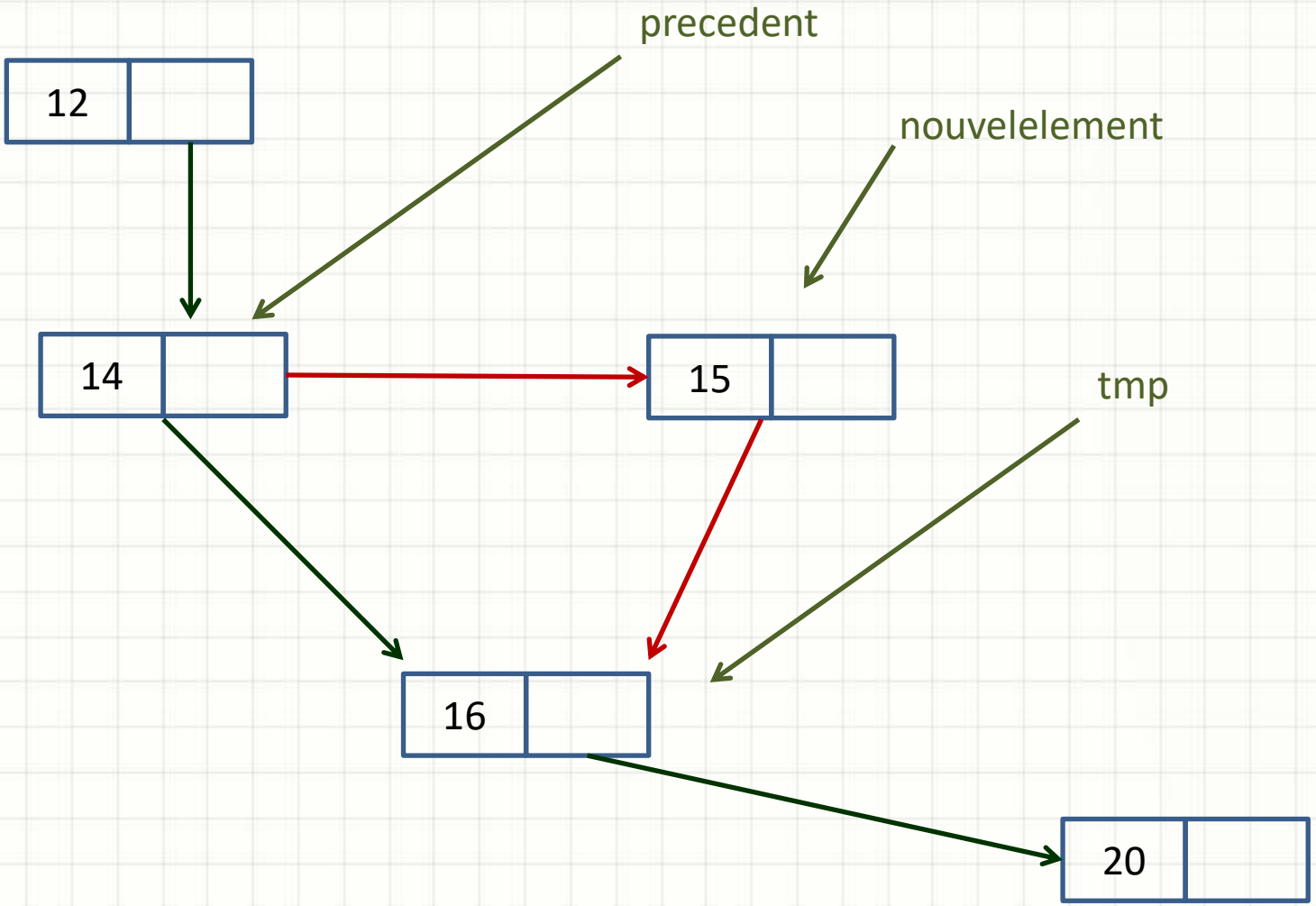
Effacer le premier élément de valeur = x



Effacer le premier élément de valeur = x

```
llist effacerElementListe(llist liste, int x)
{  if(liste == NULL)    return NULL;
   else
   { /* Sinon, on parcourt et on s'arrêt au premier élément trouvé */
     element *tmp=liste, *precedent=liste;
     while ((tmp->next!=NULL) && (tmp->val!=x))
                { precedent = tmp; tmp = tmp->next;}
    if (tmp->val!=x) return  NULL;
    else { if (tmp->next==NULL)
                 tmp = supprimerEnfinListe(liste,x);
           else  if (tmp == liste) tmp = supprimerEntêteListe(liste,x);
                    else {precedent->next = tmp->next;
                          free(tmp);retrurn (liste);}
          return tmp;
   }
```

Ajouter élément de valeur = x dans une liste ordonnée



Ajouter élément de valeur = x dans une liste ordonnée

```
llist AjouterElementListe(llist liste, int x)
{   if(liste == NULL)    liste=ajouterEntete(liste, x);
    else
    { element* nouvelElement = malloc(sizeof(element));
      nouvelElement->val = x;
      nouvelElement->next = NULL;
      /* Sinon, on parcourt et on s'arrêt à la bonne position*/
       element *tmp=liste, *precedent=liste;
       while ((x>tmp->val) && (tmp->next!=NULL))
            {precedent = tmp;    tmp = tmp->next;}
       if (tmp->next == NULL) ajouterEnFin(liste,x);
      else {precedent->next = nouvelelement;
         nouvelelement ->next = tmp; }
}
```