Tableaux, pointeurs, structures et listes chaînées

en langage C

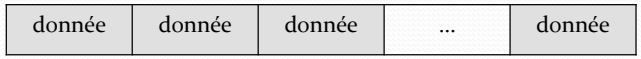
Types complexes

Le langage C propose deux types complexes permettant au programmeur de construire d'autres types à partir des types primitifs :

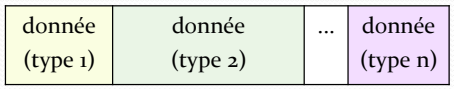
- le type tableau qui peut contenir un certain nombre de données d'un même type
- le type structure qui peut contenir des données de types différents

Ces deux types de données combinés aux pointeurs permettent de définir de nombreux types de données structurées.

- Un tableau est une variable composée de données de même type, stockées de manière contiguë en mémoire (les unes à la suite des autres).



- Une structure est une variable composée de données de types hétérogènes, stockées en mémoire les unes derrière les autres.



Tableau

```
int tab [4] ;
```

On déclare ici un tableau de 4 variables de type `int`. Sa taille est fixe et ne peut pas être modifiée.

Cette déclaration permet d'accéder à quatre cases mémoires comme s'il s'agissait de variables, à l'aide d'identificateurs composés de tab et d'un indice, comme les composantes d'un vecteur mathématique : `tab[0]`, `tab[1]`, `tab[2]` et `tab[3]`

Nombre d'éléments

Le nombre nb d'éléments d'un tableau est calculable à partir de la taille du tableau et de celle de son 1er élément (ou son type)

```
nb = sizeof tab /sizeof tab[0]
```

Les variables accessibles du tableau sont `tab[i]` avec `0<=i` et `i<=nb-1`.

L'opérateur `sizeof` renvoie la taille en bytes d'une variable ou d'un type. La taille renvoyée a le type `unsigned long`, donc le nombre d'éléments ainsi calculé aussi.

Nombre d'éléments

```
#define nbElem(t) sizeof t/sizeof t[0]
```

Cette définition s'adresse au préprocesseur.

Il remplace, dans le texte source du programme, toutes les apparitions (ou occurrences) de `nbElem(<qqc>)`

par

`sizeof <qqc>/sizeof <qqc> [0]`

Attention: à ne pas terminer une ligne commençant par `#define` avec un ";".

Déclaration + initialisation

- `int tab[4] = {6,4};`

Les deux premières valeurs seront définies par 6 et 4, et les suivantes par 0.

6	4	0	0

`tab[0]` `tab[1]` `tab[2]` `tab[3]`

Ainsi, pour initialiser un tableau de 4 éléments par des valeurs nulles, on pourra écrire

```
int tab[4] = {0};
```


Déclaration + initialisation

- `int tab[] = {6,4,9,1,0,8};`

Les valeurs des variables `tab[0]`, … , `tab[5]` sont initialisées par ces valeurs constantes, et le nombre d'éléments du tableau est égal au nombre de constantes figurant entre les accolades, soit ici six.

6	4	9	1	0	8

Déclaration + initialisation

- `int tab[] = {0};`

Ce tableau n'a qu'une case, initialisée à zéro. Par contre

- `int tab[15] = {0};`

Ce tableau a 15 cases, initialisées à zéro. Mais cela ne fonctionne que pour `0`.

- En `C99` (mais pas en `C90` ou `ANSI C`).

```
int tab[8]={[1]=5,[5]=6,[3]=9};
```

équivaut à

```
int tab[8] = {0,5,0,9,0,6,0,0};
```


Pointeurs

Les pointeurs contiennent une adresse, et ils permettent d'accéder à n'importe quelle zone de la mémoire.

Ils permettent de faire des opérations sur des variables de type complexe, comme les tableaux, les chaînes de caractères, les structures ou de manipuler de nouveaux types comme des tableaux de tableaux, des listes d'éléments, des arbres, etc.

Type pointeur

- Un pointeur est une variable dont la valeur est une adresse (celle d'une autre variable).

=> mais s'il n'est pas initialisé, il peut contenir n'importe quoi, et pas nécessairement une adresse valide.

- quand p a pour valeur l'adresse d'une variable x, on dit que p *pointe sur* x ou que p est *un pointeur sur x.*

Dans ce cas **p est la valeur de x*

Déclaration des pointeurs

Les pointeurs occupent la même place en mémoire, mais on leur attribue un type de pointeur sur un autre type: celui des variables sur lesquelles ils vont pointer.

```
int *pt;  /* pointeur sur un int */
char *p;  /* pointeur sur un char */

void *p_g; /* pointeur générique */
```

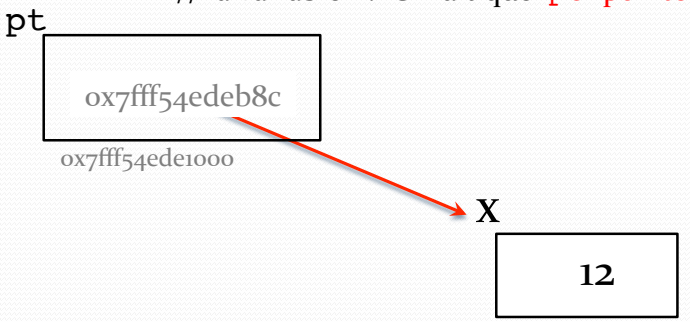
Initialisation des pointeurs

Quelque soit le type d'un pointeur, on peut l'initialiser à la valeur NULL.

```
int *pt=NULL;
int x = 12;

pt = &x;
printf("*pt =%d", *pt);
/* imprime "*pt =12" */
```



```
int * pt = NULL ;  // déclaration d'un pointeur sur un int
                   // on peut aussi écrire  int  *p;
int x = 12 ;       //  x  initialisé à la valeur 12

   pt = &x ;  // initialisation du pointeur p  par l'adresse de
              // la variable x.  On dit que  pt  pointe sur x
```



Incrémentation de pointeur

Si p est un pointeur sur t, et que la valeur de p est une adresse a (celle d'une case de la taille de t), alors p+1 ne vaudra pas a+1, mais l'adresse de la case de taille t suivante.

On peut ainsi parcourir les cases d'un tableau, en mettant dans un pointeur p l'adresse de début du tableau, et en incrémentant ensuite ce pointeur pour aller de case en case.

Relation tableau/pointeur

Un tableau tab est un pointeur constant dont la valeur est l'adresse de la 1ere variable à laquelle il donne accès.

On a donc l'égalité `tab == &tab[0]` et si on cherche à afficher `tab` au format pointeur

```
printf("tab = %p", tab);
```

on obtiendra une adresse, par ex:

```
tab = 0x7fff58140c20
```

Relation tableau/pointeur

On a les égalités suivantes

```
tab       ==  &tab[0]
tab+i     ==  &tab[i]
*tab      ==  tab[0]
*(tab+i)  ==  tab[i]
```

Ces équivalents sont ceux utilisés par le compilateur pour calculer les références à tab[i].

Tableau

!!!! W A R N I N G !!!!

Si on tente d'accéder à `t[j]` avec un indice `j` négatif ou supérieur ou égal au nombre d'éléments du tableau, un avertissement (*warning*) sera émis par le compilateur si on compile le programme avec l'option -Wall.

L'exécution du programme risque d'être interrompue ou le programme pourrait terminer en retournant un résultat faux, car la variable `t[j]` n'est en effet pas définie.

Exécution en mémoire vive

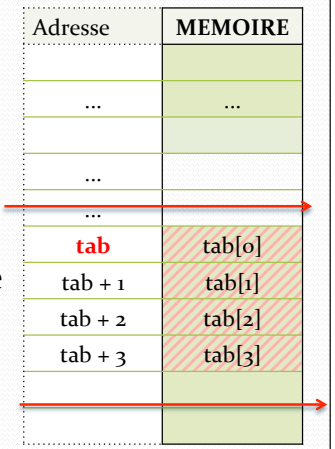
Pile d'exécution des appels de fonctions	Variables locales, paramètres et infos sur les fonctions
Tas	Allocation dynamique
Zone de données	Variables globales et statiques
Zone texte du code du programme	Contenu des fonctions

Pourquoi ?

```
int tab[4];
```

Cette déclaration conduit le compilateur à allouer de la mémoire pour la variable `tab`.

La place allouée est égale à 4 fois la taille d'un `int` et permet de ranger les 4 valeurs des variables du tableau.



Que se passe-t-il si le programme cherche à accéder à `tab[4]`*?*
L'adresse de tab[4] est tab + 4 et on distingue deux cas, selon que cette adresse appartient à un fragment accessible de la mémoire, ou non.

Adresse	MEMOIRE
...	...
	return
...	
...	
tab	tab[0]
tab + 1	tab[1]
tab + 2	tab[2]
tab + 3	tab[3]

Si tab + 4 n'est pas valide, l'évaluation de tab[4] provoquera une erreur d'exécution. Dans ce cas, on verra s'afficher le message *segmentation fault* ou *memory fault* qui signifie que l'on a tenté d'accéder à une adresse mémoire non autorisée.

Le message supplémentaire *core dumped* indique que la mémoire vive a été recopiée dans un fichier qui pourra être exploité par un debugger.

Si tab + 4 est accessible, il n'y aura pas d'interruption, mais le programme sera incorrect car la valeur stockée à tab + 4 est a priori indéterminée (cela peut aussi être l'adresse d'une autre variable).

Une erreur plus tardive est alors possible, et particulièrement difficile à déceler. Et même si le programme termine, son résultat risque d'être faux sans que l'on en soit averti.

- `int tab[4];`

L'espace mémoire est réservé (4 `int`) mais les valeurs ne sont pas encore initialisées.

On pourra initialiser ces valeurs dans la fonction `main` en faisant des affectations sur les variables `tab[i]` du tableau.

```
int main() {
      int tab[4];

      tab[0] = 6;  tab[1] = 4;
      tab[2] = 9;  tab[3] = 2;
      return EXIT_SUCCESS;
}
```

Variante avec une boucle et la macro qui renvoie le nombre d'éléments du tableau. On initialise ici toutes les variables du tableau avec l'entier 9.

```
#define nbElem(t) sizeof t/sizeof t[0]
…
int main() {
     int tab[4];

     for (int i=0; i<nbElem(tab) ;i++)
        tab[i]= 9;
}
```


Tableau et fonction

Quand on passe un argument de type tableau à une fonction, on ne voit pas l'opérateur d'adresse &, mais on lui passe une adresse, puisqu'un tableau est un pointeur (sa valeur est &tab[0]).

Conséquence:

Il s'agit d'un passage d'argument par référence (i.e. par adresse) et les valeurs des variables du tableau peuvent être modifiées par la fonction.

On peut ainsi initialiser le tableau en appelant une fonction prenant en paramètre ce tableau.

Par exemple on peut appeler la procédure `init9` qui affecte toutes les valeurs du tableau avec l'entier 9.

```
void init9(int tableau[],int nb);
```

Notez que premier argument a le type tableau de type `int`, sans précision de taille. C'est l'argument `nb` qui permet ici d'en préciser la taille.


```
/* déclaration de init9 */
void init9(int tableau[],int nb);

int main() {
     int tab[4];

     init9(tab,4);
     return EXIT_SUCCESS;
}
/* définition de init9 */
void init9(int tableau[],int nb){
     for (int i=0 ; i<nb; i++)
          tableau[i]= 9;
     return;
}
```

Exercices

Dans toutes les déclarations qui suivent, on suppose que `nb` *est le nombre d'éléments du tableau* `tab` *passé en argument.*

- `void imprime(int tab[], int nb)`

Cette procédure affiche les valeurs du tableau entre crochets en les séparant avec des virgules.

- `void init(int tab[],int nb, int val)`

cette procédure initialise un tableau d'entiers en affectant la valeur val à tous ses éléments.

Exercices

- `void imprimeChar(char tab[], int nb)`

cette procédure imprime un tableau de caractères en affichant successivement les valeurs de tous ses éléments.

- `void traduire(short tab[], int nb, char ascii[])`

cette procédure effectue la traduction d'un tableau d'entiers courts en un tableau de caractères (de même taille) où pour chaque i < nb, ascii[i] sera le caractère de code ASCII tab[i].


```
void imprime(int tab[], int nb)
{ /* on suppose que nb est le nb d'elements de tab */
  if (nb == 0) {
       - afficher: [ ]
       - quitter la fonction imprime
   }
   sinon: on sait ici que l'on a nb >=1
   – afficher d'abord [tab[0]
   – faire une boucle à partir de i= 1 qui
          - affiche   ,tab[i]
          - incrémente i
     tant que i<= nb-1 (condition d'entrée)
   Ici, on sait que i == nb  car la condition est fausse
   et on a déjà affiché [val0, ..., valn-1
   – afficher alors ]
   – retourner void
}
```

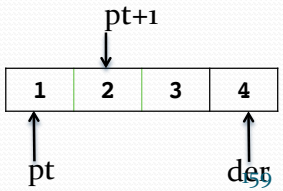


```
void imprime(int tab[], int nb)
{/* on suppose que nb est le nb d'elements de tab */
  int i;
  if (nb == 0) {
   printf("[]");/* affiche: [ ]   */
   return;  /* retourne void */
   }
  printf("[%d",tab[0]);/* affiche [tab[0] */
  i = 1;
  while (i < nb) {
   printf(",%d", tab[i]);
   i = i+1;
  }
  /* on a déjà affiché [val0, ..., valn-1 */
  printf("]");    /* affiche ] */
  return;
}
```

Parcours de tableau

Les pointeurs permettent de parcourir les valeurs d'un tableau. Exemple :

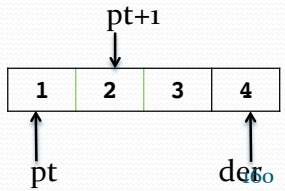
```
int tab[]= {1,2,3,4};
int *pt = &tab[0]; // pt pointe sur le début
int *der = &tab[3]; // der pointe sur la fin

for ( ; pt <= der; ) {
     printf( "%d ", *pt);
     pt = pt+1;
}
```



Parcours de tableau

Variante 1 :

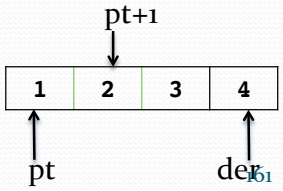
```
int tab[]= {1,2,3,4};
int *pt = tab; // pt pointe sur le début
int *der = &tab[3]; // pointe sur la fin
/* et est équivalent à:   int *der = tab + 3 */

for ( ; pt <= der; pt++) {
     printf( "%d ", *pt);
}
```



Parcours de tableau

Variante 2 :

```
#define nbElem(t) sizeof t/sizeof t[0]
int tab[]= {1,2,3,4};
int *pt = tab; // pt pointe sur le début
int *der = tab + nbElem(tab)-1;
for ( ; pt <= der; )      {
     printf( "%d ", *pt++);
}
```



Tri d'un tableau

Un algorithme possible est le suivant :

Si le nb d'éléments du tableau est `0` ou `1`
alors (ne rien faire) retourner `void`.
Sinon parcourir le tableau et pour chaque i
comparer `tab[i]` et `tab[j]` pour `j > i`
(il doit être plus petit pour que le tableau soit trié)
si on trouve un `j` avec `tab[j] < tab[i]`
échanger les valeurs de `tab[i]` et `tab[j]`.

162

```
void trieTab(int tab[], int nb){
  int i, j;
  if ((nb == 0) || (nb == 1))
      return;
  for (i=0; i < nb-1; i++)
    for (j=i+1; j < nb; j++)
      if (tab[i] > tab[j])
        echange(&tab[i],
                &tab[j]);
  return;
}
```


Version pointeur

```
void trieTab(int tab[], int nb){
 int *p , *q;
 int *der = tab + nb-1;
  if ((nb == 0) || (nb == 1))
      return;
 for (p = tab; p <= der; p++)
   for (q = p+1; q <= der; q++)
     if (*p > *q)
       echange(p,q);
 return;
}
```



```
void trieTab(int tab[], int nb){
  int *p, *q;
  int *out= tab + nb; // en dehors
  if ((nb == 0) || (nb == 1))
      return;
  for (p = tab; p < out; p++)
    for (q = p+1; q < out; q++)
      if (*p > *q)
        echange(p,q);
  return;
}
```


Chaînes de caractères

Chaines de caractères

Il n'y pas de type chaîne de caractères explicite en C. Mais le langage définit des chaînes de caractères constantes, notées par une suite de caractères entre guillemets.

Les chaînes de caractères constantes sont représentées par des tableaux de caractères (type char *) et la fin de la chaîne est indiquée par le caractère '\0' (caractère nul, de code 0).

définition de type: typedef

Le mot-clé `typedef` permet de définir un nouveau nom de type pour un type défini par ailleurs. Cette déclaration ressemble à celle d'une variable :

```
typedef unsigned int Longueur;
Longueur longueurMax;
Longueur  longueurs[];
```

Pour les chaînes de caractères, on trouve

```
typedef char * String;
String toto = "TOTO";
```


Chaînes constantes

Une chaîne constante est représentée par une suite de caractères entre guillemets. Elle est invariante en mémoire.

Ex: "Coucou"
est un tableau de 7 `char` dont le dernier est le caractère de code ASCII zéro.

Les chaînes constantes permettent de représenter un texte en utilisant les caractères usuels et des caractères spéciaux, appelés caractères d'échappement.

Chaînes constantes

Les caractères entre les guillemets sont a priori des caractères imprimables, mais on utilise aussi des caractères précédés d'un \ qui indique que ce qui suit a une autre signification. Ainsi: \", \\, \n, \t, etc. ont des significations bien précises.

On peut aussi indiquer un caractère par son code, comme dans `\012`.

Chaînes de caractères

Le caractère nul en bout de chaîne permet d'afficher des chaînes de caractères dans le format "string" avec la fonction printf("%s", str).

Dans ce format, les caractères sont affichés un à un jusqu'à rencontrer le caractère nul qui indique la fin de la chaîne.

Si aucun caractère nul ne survient, la mémoire étant limitée, on en viendra à chercher à accéder à une adresse invalide et il se produira une erreur.

Chaînes de caractères

On peut déclarer une chaîne de caractères `str` de deux manières :

1. comme tableau de `char`

```
char str[] =  "COUCOU";
```

ou

2. comme pointeur sur un `char`

```
char*  str = "COUCOU";
```

Les deux types sont équivalents et la chaîne de caractères finira par le caractère nul, du fait de son initialisation par la chaîne constante "COUCOU".

chaîne: tableau de char

- `char tab[4] = "ABC";`

Les valeurs des variables `tab[0]`, ..., `tab[3]`sont initialisés avec les caractères 'A', 'B', 'C' et '\0' de la chaine constante (de taille 3). Une chaine constante termine toujours par un caractère nul - ce qui permet de l'imprimer dans le format %s.

A	B	C	\0

La déclaration

`char tab[] = "ABC";` est équivalente

chaîne: tableau de char

- `char tab[] = "ABC";`

Ici, le tableau est de taille 4 et initialisé de la même manière. On préfèrera cette forme d'initialisation à la précédente, car une erreur sur la taille, comme avec

- `char tab[4] = "ABCD";`

ferait que le tableau contiendrait les caractères A, B, C et D, mais ne finirait pas par le caractère nul. Il ne pourrait alors pas être imprimé comme chaîne de caractères dans le format %s.

chaîne: tableau de char

- `char tab[20] = "ABC";`

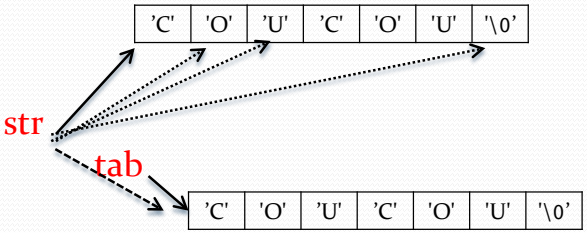
On pourra initialiser un tableau avec une chaine constante de taille inférieure à celle du tableau, en vue de faire ensuite d'autres opérations. Mais il est prudent de compléter cette initialisation en plaçant un caractère nul dans le dernier caractère du tableau :

```
tab[19] = '\0';
```

'A'	'B'	'C'	'\0'	...	...	...		...	...		'\0'

tableau ou pointeur ?

```
char tab[] = "COUCOU";
char *str  = "COUCOU";
```



On peut modifier str, faire str++, ou encore str = tab, mais tab lui-même ne peut pas être modifié.

Pointeur char * pt;

Les pointeurs sont un outil formidable pour parcourir les chaînes de caractères, et la bibliothèque contient de nombreuses fonctions sur les chaînes accessibles dans `<string.h>`.

A titre d'illustration, nous allons lister quelques unes de ces fonctions.

Le test de fin de chaîne est `*pt == '\0'`. Mais le compilateur convertissant le nombre zéro en le caractère char de code 0, on peut tout aussi bien écrire

`*pt == 0` ou `*pt == NULL`

`int strlen(const char* str);`

La fonction **`strlen`** prend en argument une chaîne **`str`** de caractères et retourne sa longueur, c'est-à-dire le nombre de caractères qu'elle contient (=ceux qui précèdent le caractère nul).

L'argument **`str`** étant déclaré **`const`**, la chaîne passée en argument ne sera pas être modifiée par l'exécution de la fonction.

L'argument **`str`** peut aussi être un tableau de caractères **`char str[]`** car les deux types sont équivalents.

char * strcpy (char *dest, const char *src) ;

La fonction **strcpy** prend une chaîne de caractères source **src** en second argument, et recopie tous ses caractères (y compris le caractère nul) dans la chaîne **dest** passée en premier argument.

C'est la chaîne de destination qui est retournée. Mais attention, la chaîne **dest** doit être dès l'origine suffisamment grande (allocation en mémoire) pour permettre la recopie de la chaîne source.

char * strcat (char *dest, const char *src) ;

Cette fonction concatène la chaine **dest** et la chaine **src**, et retourne un pointeur sur la chaine initiale **dest** - laquelle aura été modifiée et contiendra le résultat de cette mise bout à bout.

N.B. Il faut que la chaîne **dest** soit suffisamment grande (en mémoire allouée) pour que la concaténation soit possible. Mais comme pour la fonction précédente, aucun test n'est effectué pour vérifier que sa taille est assez grande.

int strcmp (const char *s1, const char *s2) ;

La fonction `strcmp` (*string compare*), compare lexicalement deux chaînes passées en argument et retourne `0` si les deux chaînes ont les mêmes caractères.
Sa valeur de retour est

- inférieure à `0` si `s1[]` est inférieure à `s2[]` selon l'ordre lexicographique,
- égale à `0` si `s1[]`et `s2[]` ont des caractères identiques,
- supérieure à `0` si `s1[]` est supérieure à `s2[]` selon l'ordre lexicographique.

Attention à ne pas confondre le fait que `s1` soit inférieure à `s2` selon l'ordre lexicographique, et le fait que la chaine `s1` soit plus courte que la chaîne `s2`.

ex: s1

'a'	'b'	'c'	'4'	'X'	'x'	'\0'

s2

'a'	'b'	'd'	'\0'

Lecture de chaînes

Un inconvénient de la fonction `scanf` est qu'elle est complexe à utiliser.

Une difficulté vient de ce que les entrées au clavier sont rangées dans une mémoire tampon (*buffer*) et que ce tampon garde les caractères encore non lus par le programme.

En particulier si vous entrez un nombre suivi d'un return, le caractère return reste dans le buffer et sera le caractère suivant lu par le programme.

Lecture de chaînes

Pour remédier à ce problème, on peut vider le buffer en appelant une fonction `clean()` après chaque appel à `scanf`. Une définition de cette fonction est donnée dans le tutoriel http://sdz.tdct.org/sdz/utiliser-les-bonnes-fonctions-d-entree.html

On pourra ainsi lire des chaînes de caractères représentant des nombres séparés par des return ou des blancs, et les affecter à des variables grâce aux fonctions qui suivent.

Conversion en nombres

Il existe plusieurs fonctions pour convertir des chaînes de caractères en nombre, quand ces chaines contiennent bien entendu les caractères d'affichage d'une constante numérique de C.

Dans la bibliothèque standard, on aura accès avec #include <string.h> à

```
int atoi(char * str)      ascii to int
long atol(char * str)     ascii to long
float atof(char * str)    ascii to float
```


Conversion en nombre

Dans <string.h> on trouvera également les conversions

string to long

```
long strtol(const char * str);
```

string to double

```
double strtod(const char * str);
```

string to unsigned long

```
unsigned long
     strtoul(const char * str);
```

Lecture de chaînes

Une seconde difficulté pour lire des chaînes de caractères avec `scanf` est que le format de lecture %s ne permet pas de lire une chaîne de caractères contenant des blancs ou des tabulations.

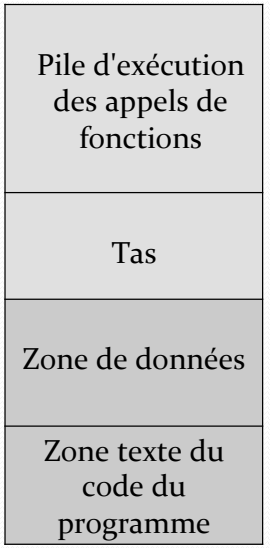
Une alternative à l'utilisation de `scanf` est `fgets`. Nous verrons dans un autre chapitre différentes fonctions d'entrées/sorties. En attendant, le code qui suit pourra être utile pour faire déjà quelques exercices en TP.

Lecture de chaînes

```
#include <stdio.h>
#include <stdlib.h>
int main ()  {
   char str[80] = "";
   char *p = NULL;

   printf("Entrez une chaine:\n");
   fgets(str, sizeof(str), stdin);
/* remplacer le return par un nul */
   for (p=chaine; *p !='\n'; p++)
       ;
   *p = '\0';

   printf("Chaine: %s\n", str);
   return EXIT_SUCCESS;
}
```


Allocation dynamique

Exécution en mémoire vive



Variables locales, paramètres et infos sur les fonctions

Allocation dynamique

Variables globales et statiques

Contenu des fonctions

Allocation dynamique

L'allocation dynamique (ou allocation dans le tas) consiste à s'attribuer une zone de mémoire dans le tas, et d'y accéder via un pointeur. On peut ainsi par exemple, allouer l'espace mémoire pour un tableau dont on ignorait la taille à la compilation.

Avantages

- On peut allouer ou libérer de la mémoire à tout moment.
- On peut accéder à cette zone depuis n'importe quelle fonction.
- On peut aussi décider d'en modifier la taille.

Allocation dynamique

Inconvénients

- L'allocation dynamique fait appel à une fonction système souvent coûteuse en temps.
- Contrairement à l'allocation des variables dans la pile d'exécution, la mémoire n'est pas libérée automatiquement quand elle n'est plus utilisée.

Si l'on ne prend pas garde à libérer cette mémoire quand on n'en a plus besoin, on risque de ne plus pouvoir faire d'autres allocations, car la taille de la mémoire du tas est limitée.

void * malloc(size_t size);

La fonction `malloc` alloue des bytes en mémoire et retourne un pointeur générique sur l'espace alloué. On force ensuite la conversion de ce pointeur par le type des données souhaitées.
Exemple:

```
typ *pt = (typ*) malloc(n*sizeof(typ))
...
free (pt);  /* libère l'espace alloué
               précédemment par malloc */
```


void * malloc(size_t size);

```
int main() {
   char* tab[]; // sa taille n'est pas fixée
   int n;
   scanf("%d", &n);
   tab = (char*)malloc(sizeof(char)*n);
/*  la taille de tab est maintenant n */
   ....
   free(tab);
   return EXIT_SUCCESS;
}
```

```
    /* alloue et initialise un tableau d'entier à o */
int* creer_tableau(int n) {
    int *table = (int*) malloc(sizeof(int)*n);
    for (int i=o; i<n ; i++)
            table[i] = o;
    return table;
}
int main() {
    int* tab[];
    tab = creer_tableau(6); /* initialisé à zéro */
    …
}
```


void * calloc (size_t n, size_t size);

`calloc` alloue n cases d'une taille size donnée, et retourne un pointeur générique sur l'espace alloué. Cette fois les bytes sont initialisés à zéro.

```
int* creer_tableau(int n) {
    int *tab =
            (int*) calloc(sizeof(int)*n);
    return tab;
}
```


Libérer la mémoire allouée

void free(void *ptr);

permet de libérer la mémoire qui a été allouée à un pointeur ptr.

- Il faut prendre soin de libérer la mémoire allouée quand on n'en pas plus besoin.
- Un programme doit faire autant de libérations qu'il a fait d'allocations.

void * realloc(void *ptr, size t n)

Lorsqu'une zone mémoire devient insuffisante, il faut lui en réallouer une autre. La fonction `realloc` recopie si nécessaire l'ancienne mémoire dans la nouvelle, et libère la place utilisée par l'ancienne.
Mais même en cas d'allocation initiale par calloc, il n'y a pas d'initialisation à zéro des nouveaux éléments.

void * realloc(void *ptr, size t n)

Attention aussi au pointeur de retour de la fonction.

En cas d'échec, realloc retourne le pointeur NULL. Il faut donc utiliser un pointeur intermédiaire pour recevoir le résultat et le tester ensuite afin de ne pas perdre l'ancienne mémoire allouée. On pourra alors libérer l'ancienne ou la garder.

Tableaux multidimensionnels

Tableaux multidimensionnels

- Ces tableaux s'utilisent de manière analogue aux tableaux à une dimension.
- Pour leur déclaration, il faut préciser la taille de chaque dimension.

Par exemple, un tableau d'entiers avec 3 dimensions :

```
int tab[size1][size2][size3];
```


Tableaux à 2 dimensions

```
int tab[4][3] ;
```

On déclare ici un tableau à deux dimensions qui contient `12` variables de type `int`.

Cette déclaration permet d'accéder aux variables du tableau avec deux indices, comme les composantes d'une matrice mathématique :

`tab[i][j]` avec `0<=i<=3` et `0<=j<=2`

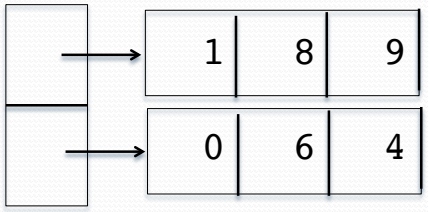
On a plus généralement la possibilité de déclarer des tableaux de n dimensions.

Déclaration + initialisation

- int tab[2][3] = {{1,8,9},{0,6,4}};

On a ici 6 cases initialisées et accessibles par tab[0][0], … , tab[1][2].

C'est un tableau de deux éléments eux-mêmes tableaux de 3 éléments.



Déclaration + initialisation

On pourra si on le souhaite utiliser un nom de type particulier pour déclarer des tableaux :

```
typedef int Tab2_3[2][3];
Tab2_3 tab;
```

ou encore

```
typedef int Matrice[2][2];
Matrice m1, m2;
```

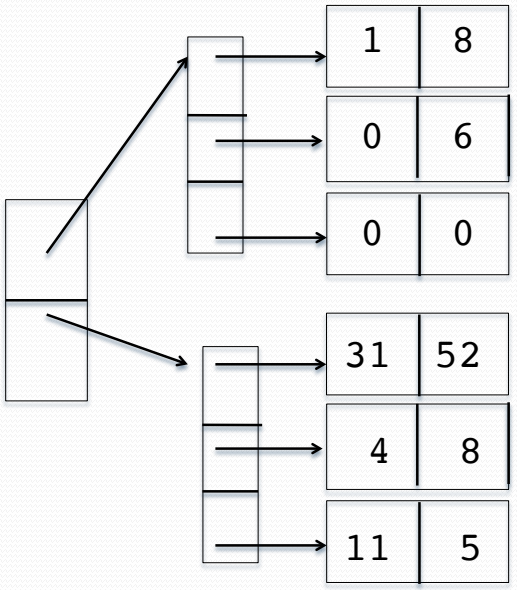

Déclaration + initialisation

- int tab[2][3][2] = {
  { {1,8} , {0,6} , {0,0} },
  { {31,52} , {4,8} , {11,5}} };

Ici, on déclare un tableau de 2 éléments qui sont des tableaux de 2 dimensions (de tailles 3 et 2).

Avec 3 dimensions, le schéma est moins parlant, mais on peut quand même élaborer la figure qui suit.

On a représenté les valeurs des cases des tableaux des premières dimensions par des pointeurs (comme dans le schéma précédent).



Parcours des valeurs

Ici, pour initialiser toutes les valeurs :

```
int tab[s1][s2][s3];
for(int i=0 ; i < s1; i++) {
  for(int j=0 ; j < s2; j++){
    for(int  k=0 ; k < s3; k++){
          tab[i][j][k] = 0;
    }
  }
}
```


Pointeurs sur ces tableaux

On ne va d'abord pas utiliser de tableau de pointeurs, mais un tableau à plusieurs dimensions d'un type donné - sur lequel on déclare un pointeur (donc pas de int ** pt);

La déclaration d'un pointeur sur un tableau de 4 entiers est

int (*pt)[4];

Les parenthèses sont nécessaires, car sans

int * pt[4];

déclare un tableau pt de 4 pointeurs sur des entiers.

Allocation avec des pointeurs sur des tableaux

Pour des tableaux à deux dimensions, sachant qu'une dimension est toujours pré-allouée lors d'une déclaration, on pourra déclarer par exemple un pointeur sur un tableau de 4 entiers avec

int (*tableau)[4];

et en compléter l'allocation par

tableau = malloc(5 * sizeof(*tableau));

ce sera équivalent à une déclaration de

int tableau[5][4];

Libération de la mémoire

```
int (*tableau)[4];
tableau = malloc(5 * sizeof(*tableau));
 if(tableau == NULL) {
        exit(EXIT_FAILURE);
}
...
free(tableau);
```

Avec ce type de déclaration, l'allocation et la libération de la mémoire est très rapide, car il n'y a pas de boucles.

avec des tableaux de pointeurs

```
int * tableauDePtr[5];
// pour un tableau  de dim2: 5x4
for(int i=0 ; i < 5 ; i++) {
        tableauDePtr[i] = malloc(4 *
                                 sizeof(tableau[0]));
        if(tableauDePtr[i] == NULL) {
                for(int i=i-1 ; i >= 0 ; i--)
                        free(tableauDePtr[i]);
                exit(EXIT_FAILURE);
        }
}
```


avec des tableaux de pointeurs

```
/*  int * tableauDePtr[5];  */
....
// libération

for(i=0 ; i < 5 ; i++) {
        free(tableauDePtr[i]);
}
```

On constate bien que dans ce cas le code est plus compliqué et moins rapide puisqu'il comporte des boucles.

Allocation dynamique de tableaux multidimensionnels

On alloue les dimensions une par une.

- Pour un tableau à 3 dimensions, on utilise un triple pointeur :

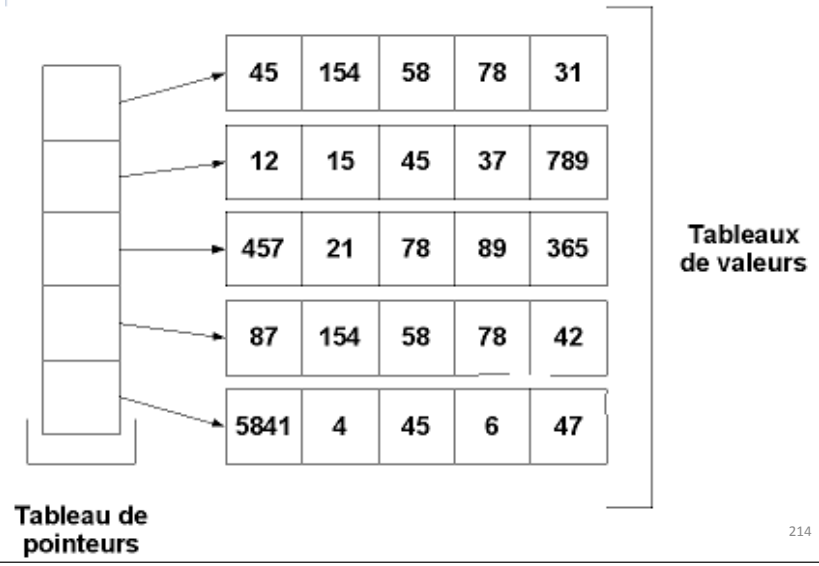
```
int ***ptr;  /* 3 étoiles si 3 dimensions */
```

- Pour un tableau à 2 dimensions un double pointeur :

```
int **ptr;  /* 2 étoiles pour 2 dimensions */
```

- On veillera à avoir le même nombre d'astérisques * que le nombre de dimensions du tableau.

Cas des tableaux bi-dimensionnels



Allocation de la 1ere dimension

Il s'agit d'allouer un tableau de pointeurs (de taille1) qui pointeront ensuite sur des espaces alloués sous forme de tableau (ici de taille2). C'est ptr qui est ainsi initialisé :

```
int **ptr;
/* on alloue d'abord taille1 pointeurs */
ptr = malloc(taille1 * sizeof(*ptr));
if(ptr == NULL)
        /* avertir et de quitter le programme.
...
```


Allocation de la 2eme

On va allouer ensuite (*ptr), c'est à dire les différents tableaux de valeurs de taille2 de l'image précédente.

```
for(int i=0 ; i < taille1 ; i++) {
  ptr[i] = malloc(taille2 * sizeof(*(ptr[i])));
/* ou malloc(taille2 * sizeof(**ptr))) */
  if(ptr[i] == NULL)
        /* libérer la mémoire déjà allouée,
        avertir et quitter le programme */
}
```


Libération de la mémoire

Elle s'effectue dans l'ordre inverse de l'allocation :

```
/* on libère d'abord la seconde dimension
c'est-à-dire les tableaux de valeurs */
for (int i=0 ; i < taille1 ; i++) {
        free(ptr[i]);
}
/* puis la 1ere: le tableau de pointeurs */
free(ptr);
```


tableaux à 3 dimensions

On procède de manière analogue, en allouant de l'espace pour des tableaux de pointeurs avec les 2 premières dimensions, puis on alloue de l'espace pour des tableaux de valeurs de taille3 concernant la dernière.

La méthode se généralise facilement au cas de tableaux de dimensions N.

Le code qui suit illustre le cas de tableaux de 3 dimensions.

```
int ***ptr;

/* allocation de la 1ere dimension */
ptr = malloc(taille1 * sizeof(*ptr));
if(ptr == NULL)
    return -1;

/* allocation de la seconde dimension */
for(int i=0 ; i < taille1 ; i++) {
    ptr[i] = malloc(taille2 * sizeof(**ptr));
    if( ptr[i] == NULL)
    return -1;
}
```



```
/* allocation de la 3eme dimension */
for(int i=0 ; i < taille1 ; i++){
    for(int j=0 ; j < taille2 ; j++) {
        ptr[i][j] = malloc(taille3 * sizeof(***ptr));
        if (ptr[i][j] == NULL)
            return -1;
    }
}
```

...

Pour la libération, en supposant que tout s'est bien passé à l'allocation, on procèdera cette fois encore en ordre inverse.


```
/* libération de la 3eme dimension: **ptr */
for(i=0 ; i < taille1 ; i++){
    for(j=0 ; j < taille2 ; j++){
        free(ptr[i][j]);
    }
}
/* Libération de l'espace *ptr */
for(i=0 ; i < 1 ; i++){
    free(ptr[i]);
}
/* Libération de l'espace ptr */
free(ptr);
```


Structures et listes chaînées

Le type struct

Déclaration d'un type nommé ici `struct toto`. Les variables auxquelles il donne accès s'appellent des champs (ou des membres) :

```
struct toto {
    int val;
    char f;
};
```

On peut ensuite déclarer des variables

```
struct toto x;
```


Le type struct

On peut aussi déclarer les variables d'un type struct sans le nommer

```
struct {
  char nom[20];
  int numero;
} x1, x2 ;
```

On accéde aux champs de ces variables avec l'opérateur '.'

```
x2.nom = "dupont";
x1.numero = 1;
```


Le type struct

Et pour un pointeur sur un struct :

```
struct {
  char nom[20];
  int numero;
} *pt, x;
```

On accédera aux membres de la structure pointée avec l'opérateur '->'

```
pt = &x;
pt->nom = "dupont";
pt->numero = 1;
```


Le type union

Une variable de type `union` est une variable dont la déclaration est semblable à celle d'un `struct`, et dont les champs peuvent aussi être de types différents. Cependant, il n'y a qu'un seul emplacement pour stocker tous les champs : sa taille est celle du plus grand type utilisé.

On peut affecter à cette variable une valeur de n'importe lequel de ces types via un membre, mais la valeur stockée et lue ensuite sera en mémoire du type de la dernière valeur affectée.

type union

```
union u_val {
  int ival;
  float fval;
  char * sval;
} valeur;
```

Ce sera la responsabilité du programmeur de garder trace du type des valeurs qu'il manipule.

type union

On a souvent une `union` dans une structure (ici son type de données pourra être gouverné par le membre `type`) :

```
struct val {
  int type;
  union {
    int ival;
    float fval;
  } u;
} *ptval, valeur;
```


type union

On accède aux champs d'une union comme à ceux d'une structure :

```
#define INT 1
if (valeur.type == INT)
    valeur.u.ival = 5;
else valeur.u.fval = 5;
ptval = &valeur;
...
if (ptval->type == INT)
    ptval->u.fval = 2.0;
```


Récursivité

Un algorithme, une fonction ou une structure de donnée sont dits **récursifs** lorqu'ils font référence à eux-mêmes dans leur définition.

Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.

Une fonction récursive est une fonction qui s'appelle elle-même directement (avec d'autres arguments), ou indirectement, par l'intermédiaire d'une autre fonction.

Fonctions récursives

On peut définir la multiplication de a par b, (a et b étant des entiers positifs ou nuls) de manière récurrente :

a x 0 = 0

a x b = (a x (b - 1)) + a

cela conduit à la définition de

```
int multiplier (int a, int b) {
  if (b==0)
    return 0;
  return (multiplier(a,b-1)+a);
}
```


Fonctions récursives

Pour qu'une fonction récursive termine son calcul, il faut qu'elle renvoie une valeur pour un cas de base, et que les autres appels à la fonction ramènent finalement aux cas de base.

Les appels correspondant aux appels successifs sont empilés dans la pile d'exécution avec leurs arguments d'appel, et dépilés au fur et à mesure du retour des résultats intermédiaires.

exemples à faire en TP

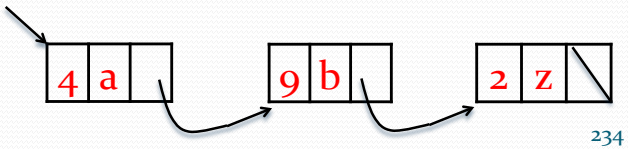
Ecrire une fonction qui prend en argument une liste et retourne la liste de ces éléments en sens inverse. On peut baser l'algorithme sur le fait qu'une liste d'un seul élément est sa propre inverse, et qu'une liste de n éléments s'inverse en mettant son premier élément en queue de la liste inversée de ces n-1 derniers éléments.

Le tri par fusion (d'un tableau ou d'une liste) est aussi un algorithme qui peut s'écrire de manière récursive.

Listes chaînées

Grâce à la **définition récursive** d'une structure (elle utilise un pointeur sur elle-même), on va pouvoir créer des listes "chainées" d'éléments.

Une liste chaînée sera elle-même un pointeur sur une structure de ce type.



Listes chaînées

Définition d'une structure appelée element. C'est une définition récursive :

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

Une liste chaînée d'entiers est un pointeur sur une structure de ce type.


```
struct element {
   void * data; /* plus générique */
   struct element *suivant;
} ;
typedef struct element Element;
typedef    Element*      Liste;

// trois déclarations équivalentes
struct element*    liste1 = NULL;
Element*           liste2 = NULL;
Liste              liste3 = NULL;
```

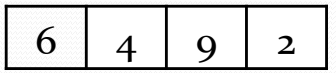


```
struct element {
   void *data;
   struct element *suivant;
} ;
typedef struct element element;
typedef    element*      liste;
```

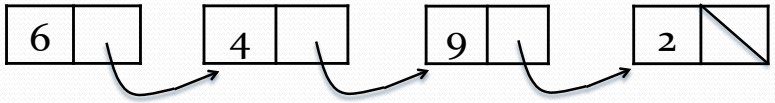
Ces définitions sont équivalentes aux précédentes (sans majuscules). Le type element est ici bien distinct d'un struct element.

Comparaison des types tableau et liste chaînée

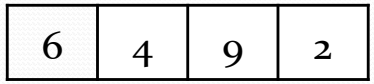
Un tableau d'entiers



Une liste chaînée d'entiers

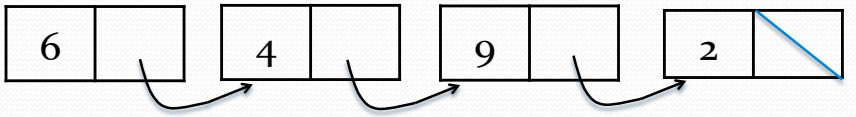


Un tableau, tab



- taille fixe (connue à la création)
- adresse du premier élément connue
- stockage continu: on peut accéder directement au i-ème élément tab[i]
- erreur d'exécution si on cherche à accéder à un élément inexistant (tab[j] avec j >= nb d'éléments du tableau ou j < 0)
- supprimer ou ajouter un élément est difficile (réallocation et/ou recopie)

Une liste chaînée d'éléments

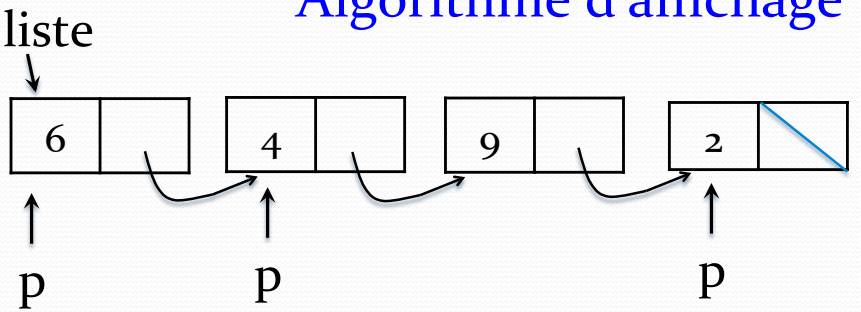


- taille inconnue et variable, limitée uniquement par la mémoire disponible
- mais impossible d'accéder directement à l'élément de rang i
- s'il n'y a pas d'élément suivant, l'adresse indiquée pour ce dernier sera NULL
- il est facile d'ajouter ou de supprimer un élément (sans avoir à recréer la liste)

Création (d'une liste d'entier) d'un seul élément

```
element*  newElement (int val) {
 element*  nouvElem
   = (element *) malloc
                    (sizeof(element));
   nouvElem->valeur = val;
   nouvElem->suivant = NULL;
   return nouvElem;
}
```

Algorithme d'affichage




```
printf("(%d", p->valeur);          // ici: (6
while ( (p = p->suivant) != NULL)
    printf(", %d",  p->valeur); // puis: , 4
printf(")\n");
```

```
void affiche (liste list)    {
 Element * p  = list;

 if (p == NULL) {          // on affiche "()"
              printf("()");
              return  ;
 }
 printf("(%d", p->valeur);          // "(" et 1er caract.
 while ( (p = p->suivant) != NULL)
          printf(", %d",  p->valeur); // les suivants...
 printf(")\n");                      // et pour terminer ")"
 return;     }
```

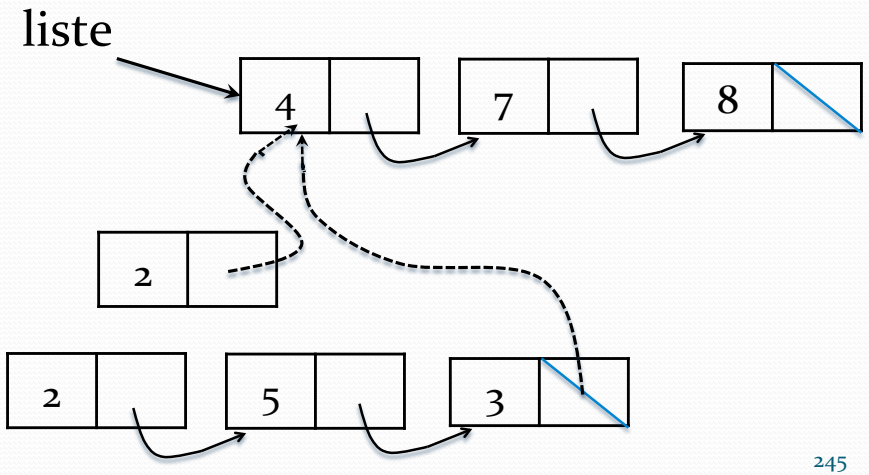

suite des fonctions en TP...

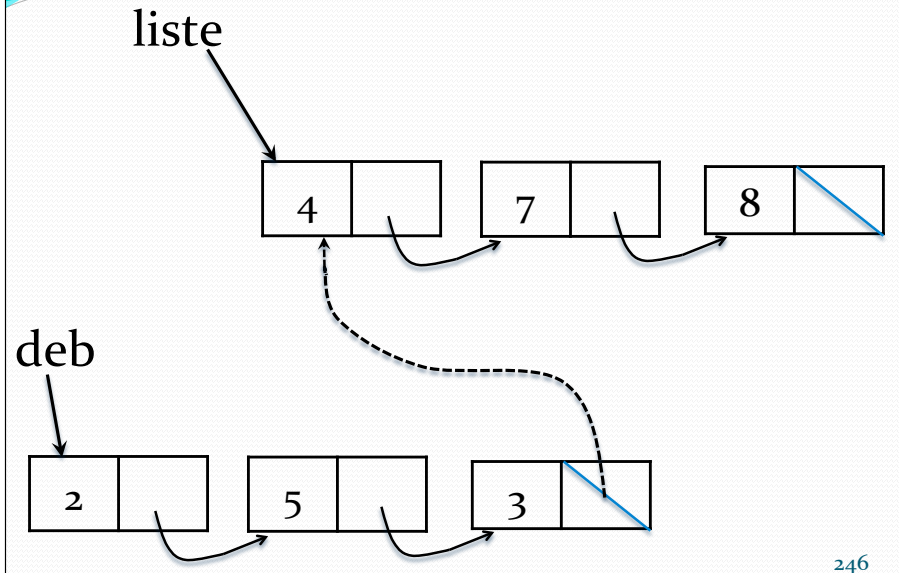
- insertion d'une liste en tête
- insertion d'une liste en queue

...

- recherche/suppression d'un élément dans une liste.

Ajouter une liste en tête de liste





Compilation séparée

Compilation séparée

Pour utiliser des listes chaînées dans un programme, on va écrire un fichier d'entête de suffixe .h pour définir ces listes (h pour *header* en anglais). On l'inclura dans le fichier contenant le main du programme, et dans d'autres fichiers de définitions de fonctions utilisées par le programme.

On va pour l'exemple prendre des listes chaînées d'éléments ayant un champ valeur de type int, mais on pourra généraliser avec un pointeur sur une structure de données.

fichier de définition de la liste

Dans le fichier `list.h`, on va définir une structure liste chaînée d'élément :

```
struct element {
   int val;  // ou void* data
   struct element *next;
} ;
typedef struct element element;
typedef     element*        list;
```


fichier de déclarations de fonctions sur ce type de liste

Dans un autre fichier `biblio.h`, on va déclarer des fonctions sur ces listes. Par exemple :

- création d'une liste d'un élément
- insertion d'une liste en tête
- insertion d'une liste en queue
- ...
- recherche/suppression d'un élément

Ces fonctions seront déclarées extern et non définies mais elles seront définies dans un (ou plusieurs) autres fichiers.

```
/* fichier biblio.h */
#include "list.h"

/* déclaration des fonctions */
/* utilisables par une autre */
/* unité de programme */
extern list newElem(int val);
extern list insertHead(list l);
extern list insertQueue(list l);
extern list searchFirstVal(
                    list l,
                    int val);
...etc.
```



```
/* fichier main.c */
#include "biblio.h"
...
/* définition du main*/
int main(int argc, char* argv[])
{
   ...
/* utilisation des fonctions
déclarées externes dans biblio.h
*/
   ....

}
```



```
/* fichier biblio.c */
#include <stdlib.h>
#include "list.h"

/* définitions des fonctions */
list newElement(int val) {
  list nouvElem
    = (element *) malloc
              (sizeof(element));
     nouvElem->val = val;
     nouvElem->next = NULL;
     return nouvElem;
}
...
```



```
list insertHead(const list l1,
                const list l2) {
  list pt, ptprec;

  if (l2==NULL) return l1;
  if (l1==NULL) return l2;
  for(pt=l2; pt!=NULL;
                    pt=pt->next){
     ptprec=pt;
  } /* ptprec dernier non nul */
  ptprec->next = l1;
  return l2;
}
```

... etc .... faire les autres en exercices

Compilation séparée

On compile alors séparément le fichier des définitions de fonctions et celui du programme main qui les utilise:
$ gcc -c biblio.c
et on obtient un fichier objet biblio.o
puis, on compile
$ gcc biblio.o main.c
ce qui crée un fichier exécutable a.out
qu'on aurait pu aussi appeler prog avec
$ gcc biblio.o main.c –o prog

Compilation séparée

Le fichier makefile de la commande make représente les dépendances entre les différents fichiers nécessaires pour obtenir une cible (ici un code compilé). On aurait :

```
prog: main.o biblio.o
        gcc biblio.o main.o –o prog
main.o: list.h biblio.h
        gcc main.c –o main.o
biblio.o: list.h biblio.c
        gcc –c biblio.c
```


Bibliothèque statique

Pour créer une bibliothèque statique de fonctions, il suffit d'archiver le code objet des fonctions :
$ ar -q liblist.a biblio.o

La commande archive, comme son nom l'indique, archive le fichier objet biblio.o en un fichier appelé ici liblist.a.

La bibliothèque est liée ensuite comme n'importe quel autre fichier objet dans la commande de compilation :
$ gcc fichier.o main.o liblist.a –o prog

Statique vs Dynamique

L'inconvénient d'une bibliothèque statique est qu'il faut recompiler le code des programmes qui l'utilisent en cas de modification des fonctions.

Le code objet produit pour le programme est également plus gros que celui que l'on obtient en utilisant une bibliothèque dynamique, laquelle n'est chargée en mémoire qu'au moment de l'exécution proprement dite du programme.

Plan

Bibliothèques dynamiques

Une bibliothèque dynamique (fichier suffixé par .so pour *Shared Object* comme lib*XXX*.so) est en effet chargée en mémoire au démarrage d'un processus. Le fichier exécutable et la bibliothèque sont indépendants avant le lancement du programme, et peuvent donc être maintenus séparément. Pour plus d'explication, voir https://www.blaess.fr/christophe/2012/01/28/mise-au-point-de-bibliotheque-dynamique-1-compilation-versions-et-liens-symboliques/