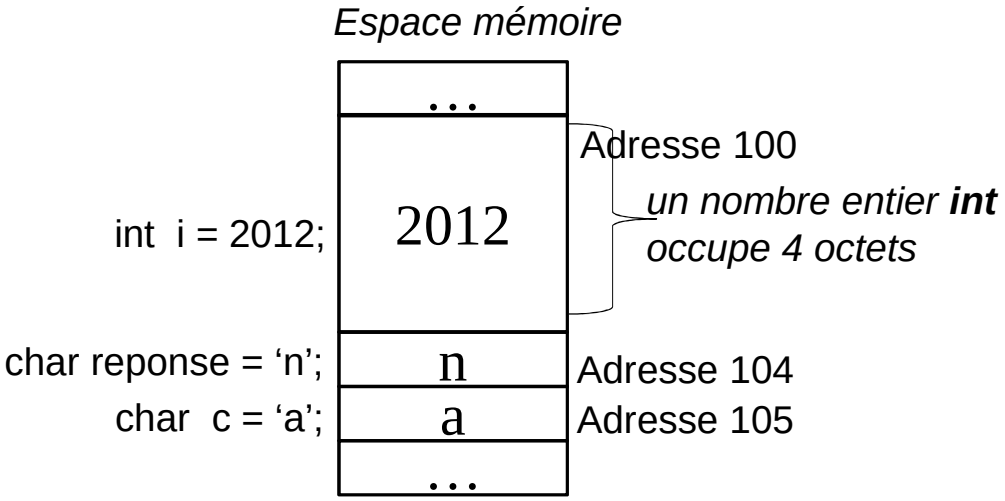
C

Ch 7 – Pointeurs et structures dynamiques

I. Pointeurs

A. Variables et espace mémoire

Chaque variable déclarée est associée à un espace mémoire qui va pouvoir stocker son contenu. Le type de la variable va déterminer la taille de l'espace mémoire réservé : int = 4 octets, char = 1 octet, etc.



Chaque variable d'un programme source est transformée en une adresse mémoire (relative au programme) au moment de la compilation.
Une variable aura toujours la même adresse tout au long de l'exécution du programme. L'adresse de la variable donne un accès direct à son contenu (adressage direct).

B. Pointeurs sur types de données de base

Un POINTEUR est un VARIABLE QUI peut CONTENIR L'ADRESSE MEMOIRE D'UNE AUTRE VARIABLE d'un type donné.

On parle alors d'ADRESSAGE INDIRECT : l'accès au contenu d'une variable passe par un pointeur qui contient l'adresse de la variable.

On dit que le pointeur pointe sur la variable, fait référence à la variable pointée (l'adresse mémoire où est stocké son contenu).

Les pointeurs sont utilisés :

- Pour passer de grands volumes de données à des fonctions : on transmettra plutôt l'adresse de début et la taille, plutôt que la totalité des données
- Pour accéder aux tableaux dans des fonctions : un tableau est définit en fait par son adresse de début d'un tableau, on peut accéder aux différents éléments par décalage de l'adresse mémoire
- Lors de l'allocation de structures de données dynamiques (listes, arbres) dont la taille n'est pas prévisible

Un pointeur étant une variable, il occupe un espace mémoire d'un mot machine : le mot machine défini une capacité d'espace adressable (le nombre de mots mémoire adressables) ; on a généralement :

- sur une machine 16 bits, un mot mémoire de 16 bits, soit 2 octets
- sur une machine 32 bits, un mot mémoire de 32 bits, soit 4 octets
- sur une machine 64 bits, un mot mémoire de 64 bits, soit 8 octets

Sa taille ne dépend donc pas du type de donnée pointée.

1. Déclaration d'un pointeur

Un pointeur est une variable qui nécessite une déclaration particulière.

Syntaxe :

```
Type_de_donnee * id_pointeur ;
```

Avec :

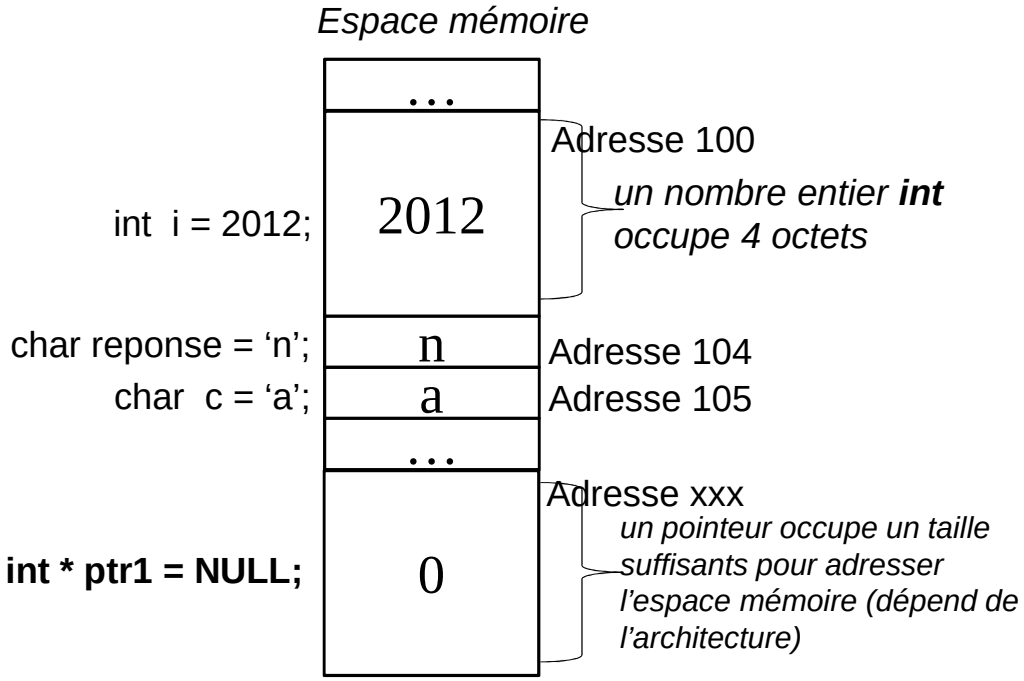
- type de données : type de donnée pointé (= type de données des variables qui pourront être pontées par ce pointeur)
- * : opérateur de déclaration d'un pointeur
- id_pointeur : identificateur donné au pointeur
  - on utilise parfois une convention de nommage de pointeur en commençant leur nom par 'ptr'

Exemple :

```
int * ptr1; // déclaration d'un pointeur vers un entier
char * ptr2; // déclaration d'un pointeur vers un caractère
```

Une fois déclaré, le pointeur n'est pas initialisé : il ne pointe donc vers aucune adresse. L'initialisation à NULL permet de l'initialiser explicitement :

```
int * ptr1 = NULL; // déclaration et initialisation
char * ptr2 = NULL; // déclaration et initialisation
```



2. Initialisation d'un pointeur : opérateur de référencement &

L'initialisation d'un pointeur consiste à lui affecté l'adresse de la variable pointée.

Syntaxe :

```
id_pointeur = & id_variable_pointee ;
```

Avec :

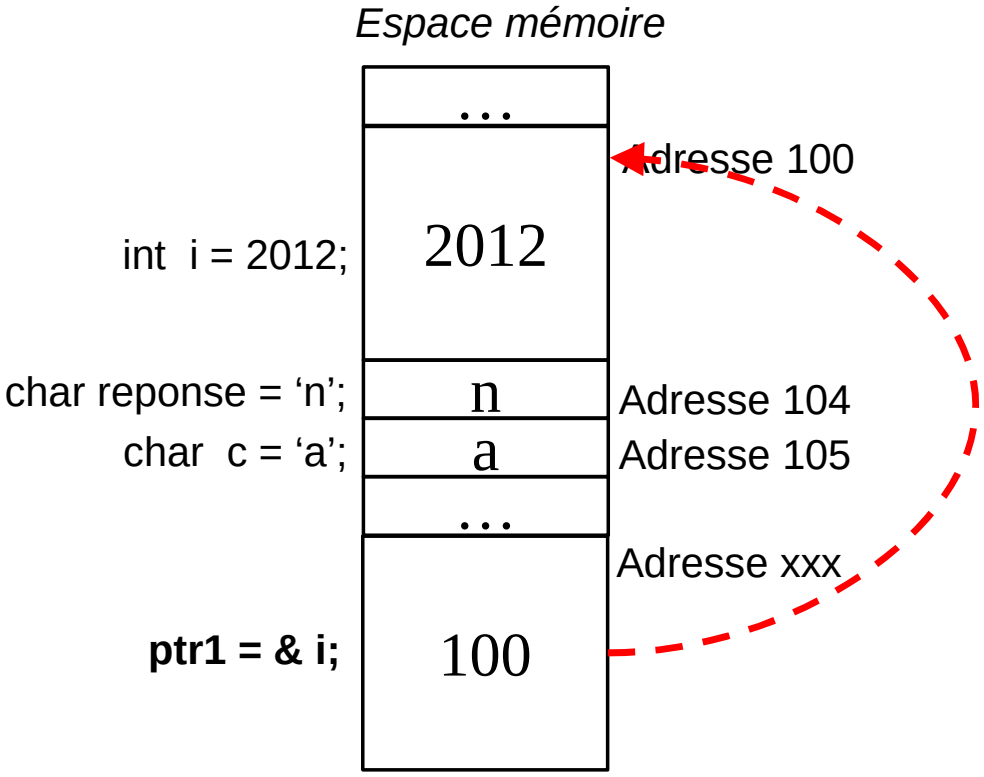
- id_pointeur : identificateur du pointeur
- & : opérateur d'adresse ou de référencement
- id_variable_pointee : identificateur de la variable pointée

Exemple :

- `int i = 0 ;`

```
• char a = 'a' ;
• ptr_1 = & i ; // ptr_1 pointe maintenant vers i
• ptr_2 = & a ; // ptr_2 pointe maintenant vers a
```

L'opérateur **&** permet la récupération de l'adresse (de la référence) d'une variable.



3. Utilisation d'un pointeur : opérateur de déréférencement *

Une fois initialisé, le pointeur permet d'accéder au contenu de la variable pointée en utilisant l'opérateur de déréférencement * (ou opérateur d'indirection), qui permet de récupérer le contenu à l'adresse pointée :

Exemple :

```
printf("%d %c",* ptr_1, * ptr_2) ; // afficher le contenu
*ptr_1=*ptr_1 + 1 ; // incrémenter le contenu
(*ptr_1)++; // Incrémenter le contenu
```

La priorité de l'opérateur de déréférencement * est inférieure à l'opérateur d'incrémentation (++), d'où la nécessité d'utiliser des parenthèses dans la 2eme incrémentation ; elle est supérieure aux opérateurs arithmétiques (+, etc.).

L'opérateur ***** donne accès au contenu de l'adresse pointée.

C. Pointeurs sur structure

Les pointeurs peuvent être utilisés également avec des données de type structure.

1. Déclaration d'un pointeur vers structure

L'opérateur * permet la déclaration d'un pointeur. La déclaration d'un pointeur précise toujours le type de la variable pointée.

Exemple : une structure et une variable de ce type

```
struct Point
{
    double x,y,z ;
```

```
    char etiquette[12] ;
};
struct Point p1 ;
```

Exemple : un pointeur vers cette structure

```
struct Point * ptr1; // déclaration pointeur vers Point
```

2. Initialisation d'un pointeur

Exemple : faire pointer vers la variable

```
ptr1 = & p1 ; // le pointeur pointe vers p1
```

3. Utilisation d'un pointeur

Pour accéder aux membres d'une structure, 2 possibilités sont offertes :

- utiliser l'opérateur **->** de sélection de membre par déréférencement :

Exemple : initialiser x, y et z

```
ptr1->x = 0 ;
ptr1->y = 0 ;
ptr1->z = 0 ;
strcpy(ptr1->etiquette,"inconnu");
```

- ou bien : utiliser l'opérateur de déréférencement * et la notation pointée :

Exemple : initialiser x, y et z

```
(*ptr1).x = 0 ;
(*ptr1).y = 0 ;
(*ptr1).z = 0 ;
strcpy((*ptr1).etiquette,"inconnu");
```

D. Pointeurs sur tableaux

Les pointeurs peuvent également faire référence à des tableaux : dans ce cas, la valeur initiale du pointeur pointe vers le premier élément du tableau.

1. Déclaration d'un pointeur

L'opérateur ***** permet la déclaration d'un pointeur. La déclaration d'un pointeur précise toujours le type de la variable pointée.

Exemple : déclarations d'un tableau d'entiers et d'un pointeur vers entier

```
int tab[100] ;
int * ptr; // déclaration d'un pointeur vers entier
```

2. Initialisation d'un pointeur

L'indirection est implicite dans le cas d'un tableau : le pointeur pointe vers l'adresse de début de tableau :

```
ptr=tab; // le pointeur pointe vers le premier élément de tab
```

équivaut à :

```
ptr=& tab[0]; // le pointeur pointe vers l'adresse de tab[0]
```

3. Utilisation d'un pointeur : incrémentation de l'adresse

L'incrémentation de la valeur stockée dans le pointeur (l'adresse de la variable pointée) peut être incrémentée ou décrémentée : la nouvelle valeur de l'adresse du pointeur fait référence à l'élément suivant du tableau (quel que soit sa longueur : tableau de type de base ou de structure).

Exemple : lister les élements d'un tableau

```
for (i=0;i<100;i++)
  printf("\n%d %d",i,*(ptr + i)); // affiche chaque élément
```

Dans cet exemple, (ptr + i) ajoute au pointeur (i * taille d'un entier).

4. Utilisation d'un pointeur : accès direct aux éléments

L'accès aux éléments du tableau par pointeur est très similaire à l'accès classique :
Exemple : accès au 5eme élément d'un tableau d'entiers

```
ptr[4] = 10 ;
*(ptr+3) = 10 ;
```

est équivalent à :

```
tab[4] = 10 ;
*(tab+3) = 10 ; // tab = adresse du 1ère élément de tab
```

5. Passage des tableaux en argument (dimension 1)

Le passage d'un tableau en argument de l'appel d'une fonction, transforme ce tableau en pointeur vers son 1er élément.
Aussi, une fonction affichant les éléments d'un tableau pourrait être écrite ainsi :

```
void affTab(int * t, int nb_elem)
{
int i ;
for (i=0;i<nb_elem;i++) printf("%d", *(t+i));
}
```

avec un appel similaire à : :

```
cons tint NB_ELEM = 10 ;
int tab[NB_ELEM];
affTab(tab,NB_ELEM) ;
```

Le prototype de cette fonction n'est cependant pas lisible directement, l'attente d'un argument tableau n'y est pas soulignée :

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

Il semble donc préférable d'écrire le sous-programme de cette manière

```
void affTab(int t[], int nb_elem)
{
int i ;
for (i=0;i<nb_elem;i++) printf("%d", t[i]);
}
```

6. Passage des tableaux en argument (dimension > 1)

Les remarques ci-dessus s'appliquent aussi à un tableau multidimensionnel. Cependant le langage impose que seule la première dimension puisse ne pas être définie, les autres devant l'être.
Aussi, une fonction affichant les éléments d'un tableau pourrait être écrite ainsi :

```
const unsigned int NB_COL = 10;
void affTab(int t[][NB_COL], int nb_lig)
{
int i,j ;
for (i=0;i<nb_lig;i++)
{
     for (j=0;i<NB_COL;i++) printf("%d", t[i][j]);
}
```

avec un appel similaire à : :

```
cons tint NB_LIG = 10 ;
int tab[NB_COL][NB_LIG];
affTab(tab,NB_LIG) ;
```

E. Pointeurs de fonctions

Les pointeurs peuvent également faire référence à des fonctions. En effet, les sous-programmes sont également chargés en mémoire et possèdent tous une adresse d'appel.

1. Déclaration d'un pointeur sur fonction

Exemple : déclaration d'une fonction et d'un pointeur vers fonction

```
int fSomme(int Vnb1, int Vnb2)
{
    return (Vnb1+Vnb2) ;
};

int (*ptrf)(int, int); // decl. pointeur sur fonction
```

Le pointeur ptrf pointe vers un entier (le type de la fonction) et définit 2 paramètres entiers (ceux de la fonction).

2. Initialisation d'un pointeur sur fonction

L'opérateur & (opérateur d'indirection) permet la récupération de l'adresse d'une variable.

```
ptrf=&fSomme; // le pointeur pointe sur la fonction fSomme
```

3. Utilisation d'un pointeur sur fonction

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (opérateur de déréférencement) :

```
int n1 = 10;
int n2 = 20;
int n3 = (*ptrf)(n1, n2) ;
```

La variable n3 reçoit la valeur de l'appel de la fonction pointée par ptrf à laquelle les valeurs de n1 et n2 ont été passées.

4. Un exemple complet

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (opérateur de déréférencement) :

```
// Définition de fonctions de calculs arithmétiques
int fSomme(int nb1, int nb2)
  {    return (nb1+nb2) ; }
int fDiff(int nb1, int nb2)
  {    return (nb1-nb2) ;}
int fProduit(int nb1, int nb2)
  {    return (nb1*nb2) ; }
int fQuotient(int nb1, int nb2)
  {    return (nb1/nb2) ;}
int fModulo(int nb1, int nb2)
  {    return (nb1%nb2) ;}

typedef int (*ptrf)(int, int); // décl. d'un type pointeur
sur fonction entier à 2 paramètres entiers

ptrf Tfonc[5] ; // allocation d'un tableau de pointeurs ptrf

int main (void)
{
    int n1, n2, n3 ;
    Tfonc[0]= & fSomme ;
```

```
    Tfonc[1]= & fDiff ;
    Tfonc[2]= & fProduit ;
    Tfonc[3]= & fQuotient ;
    Tfonc[4]= & fModulo ;

    printf( "\nentrez 2 nombres " );
    scanf("%d %d" , & n1, & n2 );

    printf( "\nentrez un numéro de fonction : " );
    scanf("%d" , & n3);

    printf( "\nrésultat : " );
    printf( "%d" , (*(Tfonc[n3]))(n1,n2) );

    return EXIT_SUCCESS ;
}
```

F. Double indirection

On peut utiliser les pointeurs pour contenir l'adresses d'autres pointeurs

1. Déclaration d'un pointeur

Déclaration d'un pointeur vers un pointeur vers char :

```
char ** ptr ;
```

2. Initialisation d'un pointeur

Allocation dynamique et initialisation du pointeur :.

```
char ch1[] = "abc" ;
char ch2[] = "def" ;
ptr = malloc(sizeof(char) * 2);
ptr[0] = ch1;
ptr[1] = ch2;
```

3. Utilisation d'un pointeur

On peut utiliser un pointeur comme si on utilisait la variable pointée en utilisant l'opérateur * (opérateur de déréférencement) :

```
char c;
for(i=0;i<3;i++)
{
    c = *(*(ptr)+i); // contenu à l'adresse de ptr
    printf("%c",c);
}
printf("\n");
for(i=0;i<3;i++)
{
    c = *(*(ptr+1)+i); // contenu à l'adresse de ptr +1
    printf("%c",c);
}
```

G. Risques sur l'utilisation des pointeurs

Les pointeurs permettent d'accéder à n'importe quel emplacement de la mémoire réservée à un programme : ils représentent un danger s'ils sont mal initialisés ou pas correctement utilisés.
Les pointeurs non initialisés (ou mal initialisés) sont la cause d'erreurs fréquentes d'exécution (erreur Windows).

II. Allocation dynamique

L'allocation dynamique est la capacité à réservée un espace mémoire en fonction des besoins du programme en cours d'exécution.
Ainsi contrairement aux déclarations classiques, généralement fixées à la compilation, l'allocation dynamique va autoriser l'allocation d'espaces mémoire et la récupération de l'adresse de départ de l'espace alloué dans un pointeur.

A. Allocation et libération d'espace mémoire

1. Allocation mémoire : fonction malloc

L'entête de la fonction **malloc** est déclarée dans le fichier d'entête <stdlib.h>. Le prototype de la fonction est :

```
void * malloc(size_t size) ;
```

Avec

- size_t : type de donnée entire non signé (vient de : typedef unisgned int size_t)
- void * : pointeur sur void, c'est-à-dire pointeur vers n'importe quoi

Syntaxe :

```
type_de_donnée  * id_pointeur = malloc(taille);
```

ou (*si le pointeur est déjà déclaré*) :

```
id_pointeur = (transtypage)malloc(taille);
```

Avec

- type_de_donnée : type de données pointé
- id_pointeur : identificateur du pointeur
- transtypage : la fonction **malloc** renvoyant un pointeur sur void, il sera utile de le transtyper vers le type requis par la déclaration du pointeur
- taille : nombre d'octets alloués
  - ce nombre d'octets devra tenir compte de la taille du type de donnée pointée, d'où l'utilisation de la fonction **sizeof(type)** qui renvoie le nombre d'octets utilisé par le type

Exemple : allocation d'un espace pour 1 entier pointé par ptr

```
int * ptr = (int *) malloc(sizeof(int));
```

Exemple : allocation d'un espace pour 10 entiers pointés par ptr

```
int * ptr = (int *) malloc(sizeof(int) * 10);
```

2. Allocation mémoire initialisée à 0 : fonction calloc

La fonction **calloc** offre une allocation mémoire dans laquelle l'espace utilisé est initialisé à 0 (attention : pour les nombres réels, il est nécessaire d'initialiser chaque élément avec 0.0). Son utilisation est identique à **malloc**.
L'entête de la fonction **calloc** est déclaré dans le fichier d'entête <stdlib.h>. Le prototype de la fonction est :

```
void * calloc(size_t nb_element, size_t taille);
```

Avec

- nb_elemen : le nombre d'éléments à allouer
- taille : la taille d'un élément
- void * : pointeur sur void, c'est-à-dire pointeur vers n'importe quoi

Exemple : allocation d'un espace pour 1 entier pointé par ptr

```
int * ptr = (int *) calloc(1, sizeof(int));
```

Exemple : allocation d'un espace pour 10 entiers pointés par ptr

```
int * ptr = (int *) calloc(10, sizeof(int));
```

3. Changement de taille d'une mémoire allouée : fonction realloc

La fonction **realloc** permet de réallouer un espace existant en augmentant ou en réduisant sa taille. Son utilisation est identique à la fonction **malloc**.
L'entête de la fonction **realloc** est déclaré dans le fichier d'entête <stdlib.h>. Le prototype de la fonction est :

```
void * realloc(void * ancien_bloc, size_t nouvelle_taille);
```

Avec

- ancien_bloc : le pointeur vers l'ancien espace alloué
- nouvelle_taille : la nouvelle taille à allouer (au total)

SI la nouvelle taille est inférieure à l'ancienne, les valeurs de l'ancien espace mémoire sont tronquées.

Exemple : augmentation de l'espace

```
int * ptr = (int *) malloc(ptr, 10 * sizeof(int)); // alloc
int * temp = (int *) realloc(ptr, 15 * sizeof(int)); // re-
if (temp == NULL)
{
     erreur à traiter !
}
ptr = temp; // ptr pointe vers le nouvel espace alloué
```

4. Libération de la mémoire allouée : fonction free

La fonction **free** permet de libérer l'espace mémoire alloué lorsqu'il ne sera plus utile.

Syntaxe :

```
free(id_pointeur);
```

Exemple :

```
free(ptr); // libération de la mémoire
ptr = NULL ; // pour la propreté du code
```

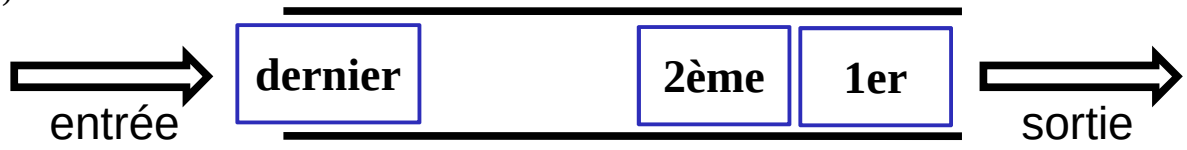
B. Structures dynamiques et pointeurs

Contrairement aux structures de données statiques (types de base, tableaux et types structurés) dont la déclaration précise la taille à la compilation, les structures dynamiques peuvent être créées dynamiquement en fonction des besoins de l'application.
L'utilisation des structures fixes présente un risque si le flux de données dépasse les réservations effectuées. Pour éviter ce problème, on peut surévaluer la taille mais les structures sont sous-utilisées (espace mémoire réservé mais non utilisé)

Les structures dynamiques permettent ainsi une gestion optimisée des données dont le flux d'arrivée n'est pas connu à l'avance grâce à une allocation dynamique , chaque espace ainsi réservé étant « pointé » par un pointeur.

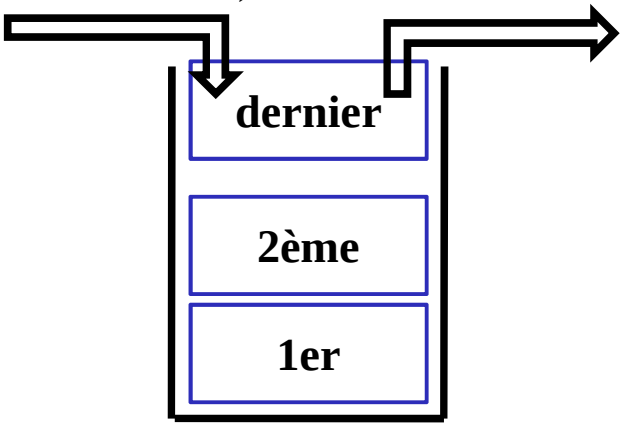
1. Les files d'attentes

Les files d'attentes (queues) sont des structures dynamiques dont les éléments apparaissent de manière aléatoires et dont le flux n'est pas contrôlable (files d'attente au péage, au guichet, au fromage !, etc.)
La suite des éléments d'une file est ordonnée, chaque élément ayant un ordre d'arrivée qui va conditionner sont ordre de sortie : le premier arrivé sera le premier sorti (file FIFO, First In First Out).



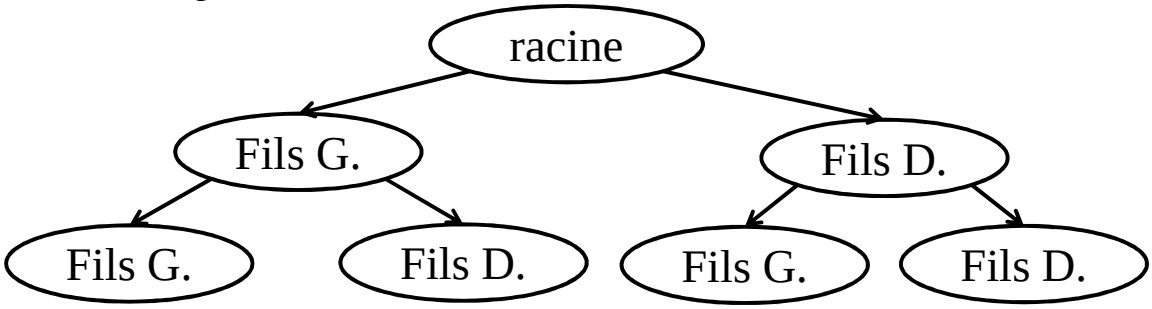
2. Les piles

La suite d'éléments d'une pile est ordonnées, chaque élément possède un ordre d'arrivée qui va également conditionner sont ordre de sortie, mais de manière inverse à la file : le dernier entré sera le premier sorti (pile LIFO, Last In First Out)



3. Les graphes et arbres

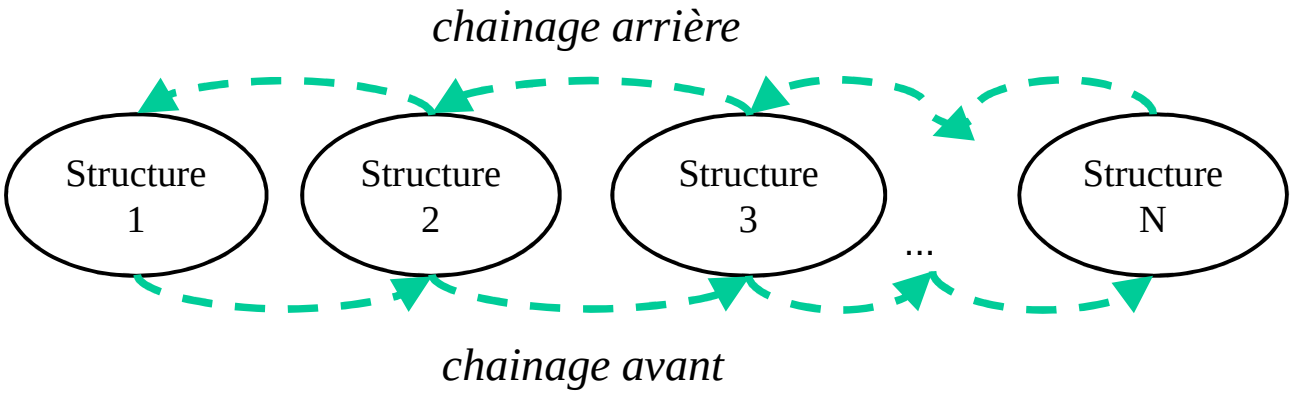
Les arbres et arbres représentent des structures dynamiques composées d'un ensemble de nœuds reliés par des arcs. Ils permettent de représenter les réseaux (trains, métro – une station est un nœud, le trajet d'une station vers une autre est un arc -, objets graphiques complexes, etc.).
Un nœud est le point de départ du graphe, c'est la racine.
Un cas particulier d'arbre est l'arbre binaire dans lequel chaque nœud est relié à 2 autres nœuds fils, le fils droit et le fils gauche, eux-mêmes sous-arbres.



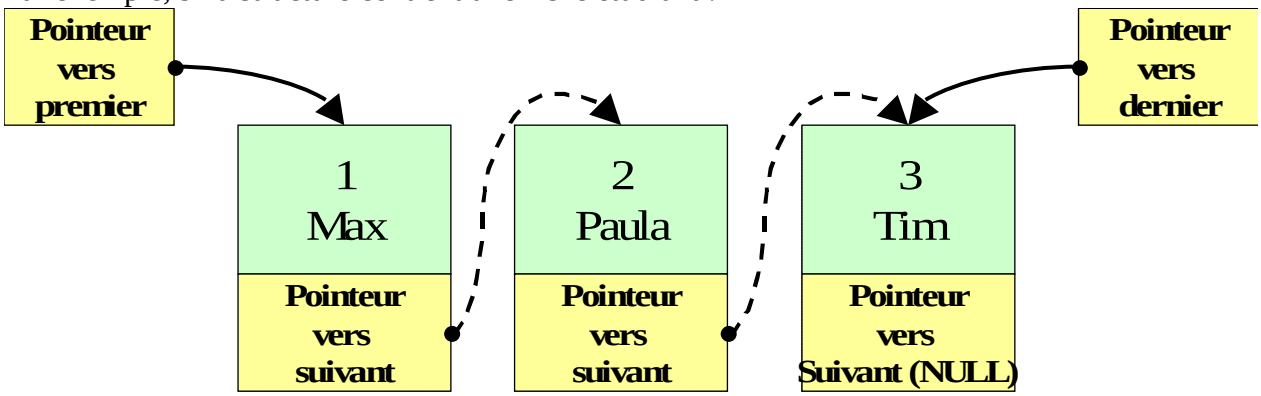
4. Listes chaînées

On connaît la limitation des tableaux déclarés avec un nombre d'indices déterminé.

Une liste chaînée est une structure de données constituée d'éléments chaînés, où chaque élément possède des **données propres** et un **pointeur vers l'élément suivant** qui permet le parcours de la liste du premier au dernier élément (chainage avant)**, et éventuellement un second pointeur vers l'élément précédent** qui permet le parcours du dernier au premier (chainage arrière) .
Le **nombre d'élément de cette structure n'est pas limité** (sauf par la taille mémoire disponible)
La liste chainée a un **inconvénient majeur** : par atteindre le nième élément, il fait avoir parcouru le chainage des (n – 1) éléments précédents.



Par exemple, si la structure contient une fiche étudiant :



a) Déclarer une structure de données

Déclarer la structure de données avec le pointeur sur l'élément suivant :

```
struct etudiant
   {
    int num ;
    char nom[20] ;
    etudiant * ptrSuivant ;
   }
```

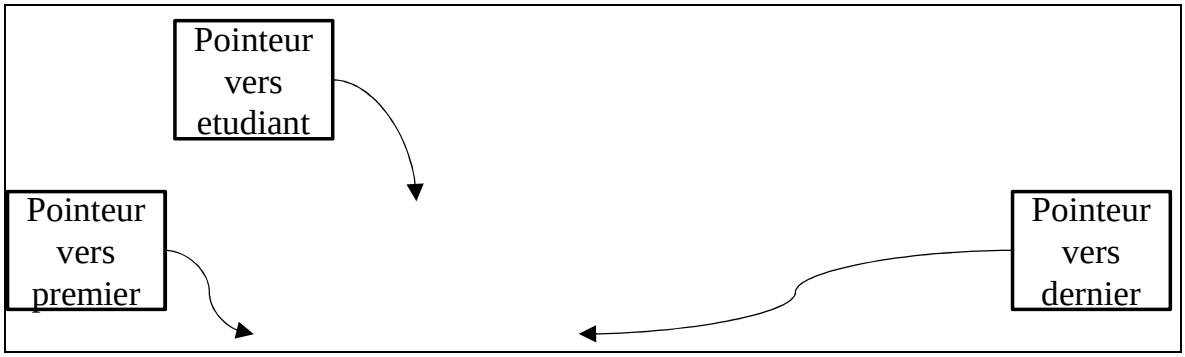
Déclarer les pointeurs qui vont permettre l'entrée dans la liste (ptrPrem) et d'accéder directement au dernier élément de la liste (ptrDern) :

```
struct etudiant * ptrPrem = NULL, * ptrDern = NULL;
```

Déclarer un pointeur de l'élément qu'on est en train de créer (ptrUnEtudiant) :

```
struct etudiant * ptrUnEtudiant = NULL;
```

Figure 1 : des pointeurs sont déclarés



b) Construire chaque élément et chaînage avec le suivant

Construire un nouvel élément « etudiant » et conserver un pointeur sur celui-ci ; gérer l'erreur d'allocation mémoire :

```
ptrUnEtudiant = (struct etudiant *)malloc(sizeof(etudiant)) ;
if (ptrUnEtudiant==NULL)
   {
    printf( "erreur allocation mémoire ") ;
    return 1 ; // erreur à gérer
   }
```

Initialiser les valeurs de l'élément :

```
scanf("%d", &(ptrUnEtudiant->num));
scanf("%s", ptrUnEtudiant->nom );
ptrUnEtudiant->ptrSuivant = NULL ;
```

Si c'est le premier élément, conserver un pointeur sur cet élément, sinon le dernier élément qui avait été créé pointe sur ce nouvel élément, et le dernier élément devient ce nouvel élément :

```
if (ptrPrem==NULL)
   {
    ptrPrem = ptrUnEtudiant;
    ptrDern = ptrUnEtudiant;
   }
else
   {
    ptrDern -> ptrSuivant = ptrUnEtudiant;
    ptrDern = ptrUnEtudiant;
   }
```

Figure 2 : un 1er étudiant est créé (1, Max)

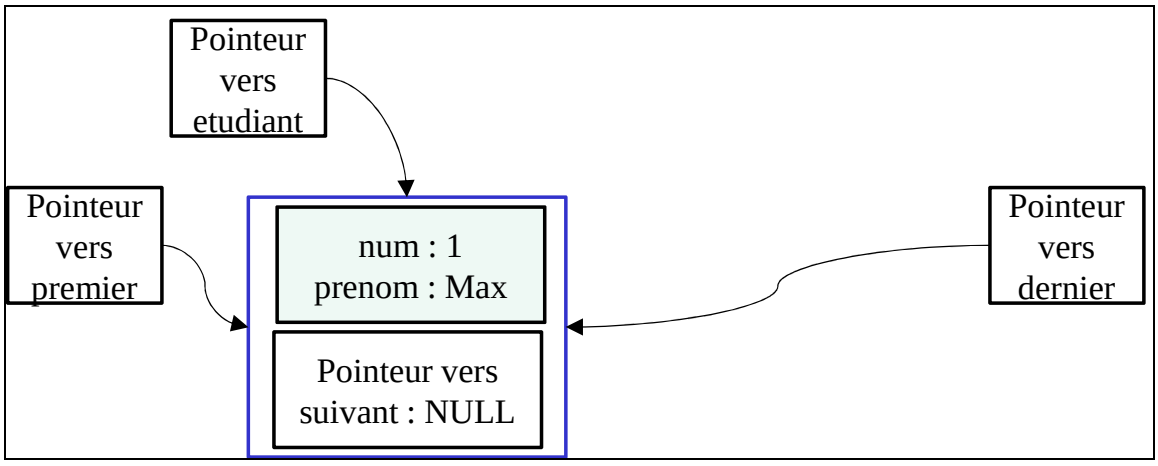


Figure 3 : un 2eme étudiant est créé (2, Paula), le chaine de Max vers Paula est réalisé

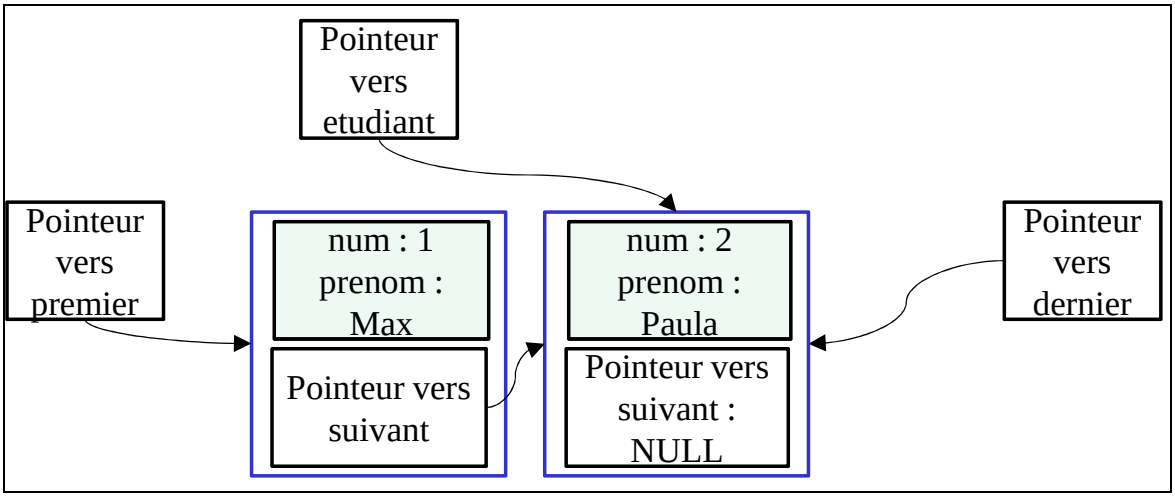
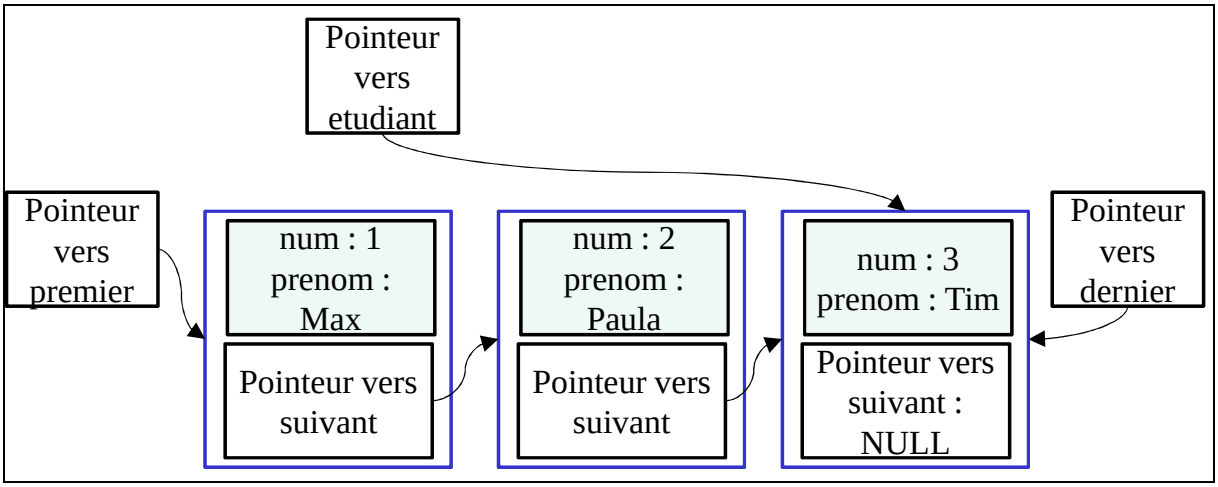


Figure 4 : un 3ème étudiant est créé, le chainage de Paula vers Tim est réalisé



c) Parcourir la liste

Démarrer par le premier élément :

```
ptrUnEtudiant = ptrPrem;
```

Si le premier élément vaut NULL, la liste est vide, sinon parcourir la liste par le lien de chainage de pointeurs sur l'élément suivant tant que le pointeur suivant n'est pas nul::

```
if (ptrUnEtudiant==NULL)
    printf("liste vide");
else
    while(ptrUnEtudiant!= NULL)
        {
         printf("\n%s", ptrUnEtudiant->nom);
         ptrUnEtudiant = ptrUnEtudiant ->ptrSuivant;
        }
```

d) Désallouer les espaces alloués

```
ptr_etudiant = ptr_premier;
struct etudiant * sv_ptr = NULL;
if (ptr_etudiant == NULL)
    printf("liste vide");
else
```

```
    while(ptr_etudiant!= NULL)
        {
         printf("\nDesallouer %s" ,ptr_etudiant->prenom);
         sv_ptr = ptr_etudiant->ptr_suivant;
         free(ptr_etudiant);
         ptr_etudiant = sv_ptr;
        }
ptr_prem = ptr_dern = ptr_etudiant = sv_ptr = NULL;
```

C. Tableaux dynamiques

On connaît la limitation des tableaux déclarés avec un nombre d'indices déterminé.
Les pointeurs offrent une possibilité de créer dynamiquement des tableaux : le nombre d'éléments est donné par l'utilisateur (saisie d'un nombre au clavier).

1. Déclarer les variables et pointeur

Déclarer la structure de données avec le pointeur sur l'élément suivant :

```
int taille, i ;
```

Déclarer le pointeur qui va devenir le tableau lui-même :

```
int * ptrTab ;
```

2. Construire l'espace réservé pour le tableau et pointer dessus

Initialiser les valeurs de l'élément, allouer l'espace :

```
printf("entrer un nombre d'elements : ");
scanf("%d",&taille );
ptrTab = (int *) malloc(taille * sizeof(int)) ;
if (ptrTab==NULL)
   {
    printf( "erreur allocation mémoire " );
    return 1 ;
   }
```

3. Utiliser le tableau

Le tableau ainsi alloué peut être utilisé comme tout autre tableau :

```
for (i=0 ;i<taille ;i++)
   {
    ptrTab[i]=i ;
   }
```

ou bien :

```
for (i=0 ;i<taille ;i++)
   {
   (*(ptrTab+i)) = i ;
   }
```

4. Libérer la mémoire

La destruction du tableau permet la libération de la mémoire qui lui avait été allouée :

```
free( ptrTab);
ptrTab = NULL;
```

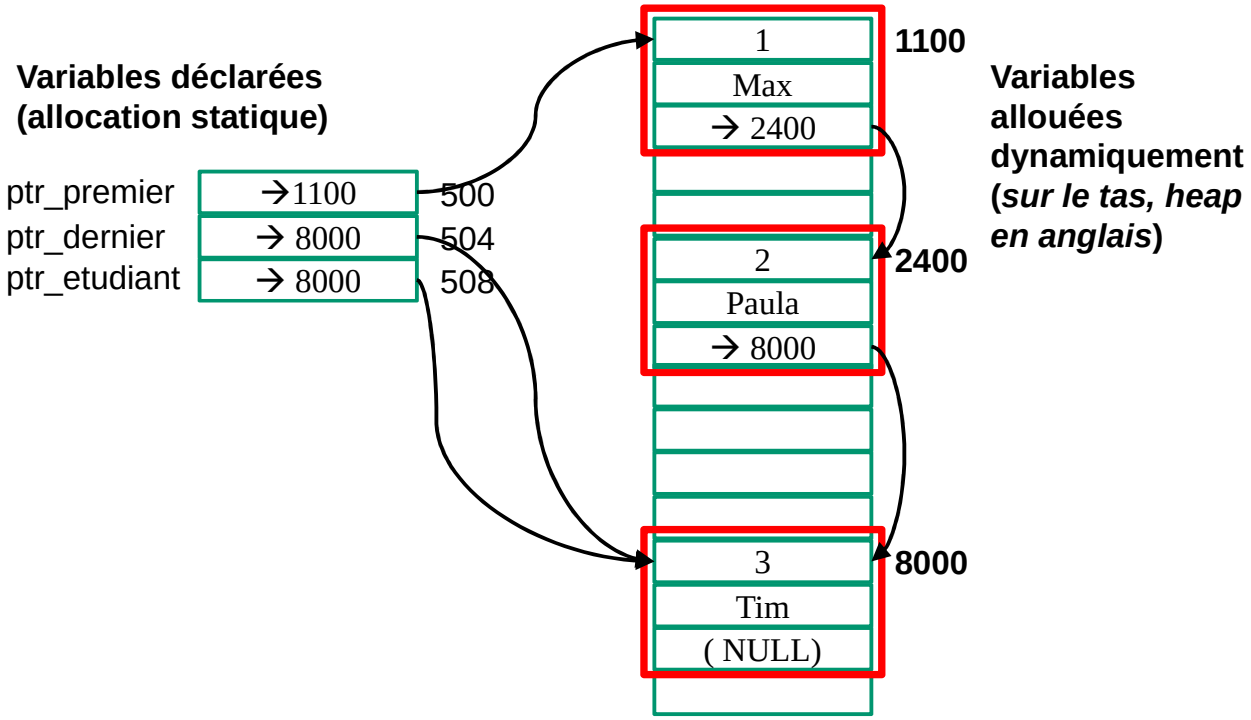
III. Différents espaces d'allocation mémoire

En réalité, l'espace mémoire dédié à l'allocation des variables est découpé en différentes zones :

- Une zone statique, pour l'allocation de variables globales qui existeront tout au long de l'exécution (c'est le principe d'allocation le plus sûr, mais rigide)

- Une zone dynamique (« la pile », stack en anglais), dédiée à l'allocation automatique de variables dans des blocs d'instructions : une fois le bloc quitté, les variables sont automatiquement desallouées, à moins d'avoir été qualifiées de 'static' (c'est un principe sûr car il utilise un empilement puis un dépilement au fur et à mesure de la libération).
- Et une zone dynamique (« le tas », heap en anglais), utilisé à la demande du programmeur : c'est dans cet espace que seront placées les variables allouées dynamiquement, à la demande (c'est le principe le moins sûr, car à la charge du développeur. Il est source d'erreurs d'exécution.)

Par exemple, dans le cas de la liste chainée, après qu'elle ait été créée :



(Remarque : les adresses sont fictives)