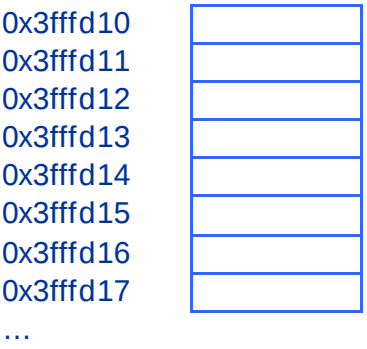
Notions de Pointeurs

I. Introduction: notion d'adresse

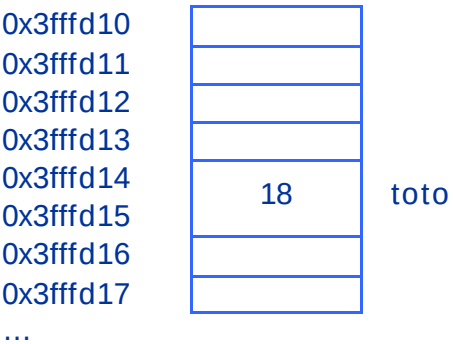
La mémoire centrale utilisée par les programmes, est découpée en octets. Chacun de ces octets est identifié par un numéro séquentiel appelé **adresse**. Par convention, une adresse est notée en hexadécimal et précédée par 0x.



Déclarer une variable, c'est attribuer un nom (l'identificateur) à une zone de la mémoire centrale. Cette zone est définie par :

- sa position c'est-à-dire l'adresse de son premier octet
- sa taille c'est-à-dire le nombre d'octets

short int toto = 18;



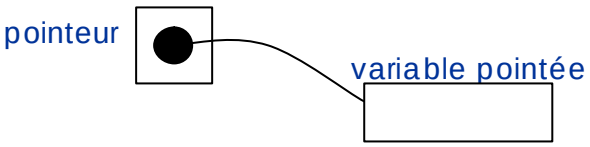
Pour accéder à la valeur contenue dans une variable, on utilise tout simplement son nom.
Mais il peut arriver qu'on veuille accéder à l'adresse d'une variable. Dans ce cas, on utilise l'**opérateur d'adresse &** (notation C++) suivi du nom de la variable.

&toto vaut 0x3fffd14 (adresse du premier octet de toto)

II. Notion de pointeur

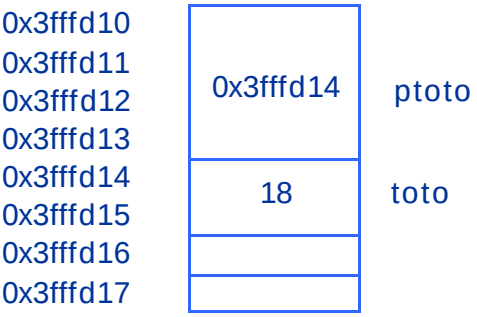
Un pointeur est une variable qui contient l'adresse d'une autre variable.

L'adresse contenue dans un pointeur est celle d'une variable qu'on appelle variable pointée. On dit que le pointeur pointe sur la variable dont il contient l'adresse.



Un pointeur est associé à un type de variable sur lequel il peut pointer. Par exemple, un pointeur sur entier ne peut pointer que sur des variables entières.

✋ Un pointeur est lui-même une variable et à ce titre il possède une adresse.
💣Il convient de ne pas confondre l'adresse de la variable pointeur et l'adresse contenue dans le pointeur (adresse de la variable pointée)

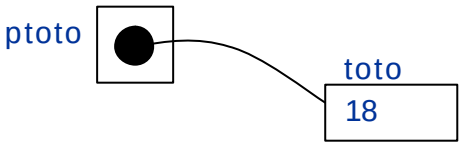


toto est placé à l'adresse 0x3fffd14

ptoto contient l'adresse de la variable toto → ptoto pointe sur toto

ptoto est contenu à l'adresse 0x3fffd14.

Représentation



✐ Petites révisions d'AMSI...
Pourquoi le pointeur ptoto est-il représenté sur 4 octets??
Pour pouvoir coder un hexa il faut ½ octet donc pour en coder 7 il faut 4 octets
7 valeurs hexadécimales ont une puissance lexicographique de 16^7 soit 256Méga.
Cela veut dire que la mémoire possèdent 256Mo.

A. Déclaration d'un pointeur

Par convention, les pointeurs commencent par la lettre p.
Il faut déclarer le type de la variable pointée.

en algorithmique	en C++
nom : pointeur sur *type pointé* exemple : ptoto: pointeur sur entier	*type * nom;* *exemple :* int * ptoto; * est appelé opérateur de déréférencement

B. Utilisation d'un pointeur

On utilise un pointeur pour mémoriser l'emplacement d'une autre variable.
Il est très rare d'affecter directement une adresse à un pointeur. On affecte en général l'adresse d'une variable existante.
Pour cela, en algorithmique, on utilise l'expression "adresse de" alors qu'en C, on utilise l'opérateur d'adresse &

☞ Exemple:

Algorithmique	C/C++

Var
va : entier
pent: pointeur sur entier
pcar: pointeur sur caractère
 	
Début
pcar ← 0x3fffd14 //affectation directe
pent ← **adresse** de va
...
	int va;
	int * pent;
	char * pcar;
	
	pcar = 0x3fffd14; //affectation directe
	pent = &va;
	...

✐ Soit v une variable de type réel et p un pointeur sur réel. Quelle instruction écrire pour que p pointe sur v? (en algorithmique et en C/C++)

Il est possible d'extraire la valeur d'une variable pointée.

Déréférencer un pointeur consiste à extraire la valeur de la variable sur laquelle il pointe.

On peut aussi modifier la valeur d'une variable pointée à travers un pointeur déréférencé (accès en écriture)

Algorithmique	C/C++
Var n: entier p: pointeur sur entier **Début** p← adresse de n *// p pointe sur n* afficher *p *//la valeur 33 est affichée* *p ← 34 *// n vaut maintenant 34*	int n = 33; int *p; *//déclaration du pointeur p* p = &n; *// p pointe sur n* cout << *p *//affiche 33 à l'écran, la valeur de n* *p = 34 *// n vaut maintenant 34*

☝ Attention à toujours initialiser un pointeur. Un pointeur qui n'est pas initialisé s'appelle un **pointeur pendant**. Un pointeur pendant ne pointe pas nul part mais n'importe où. Si l'on déréférence ce pointeur et qu'on affecte une nouvelle valeur, on va écraser un emplacement mémoire quelconque, et on risque de faire planter le programme.
Si on veut que le pointeur pointe nul part, il faut l'initialiser à NIL (not identified link) ou NULL en C/C++. C'est l'équivalent de 0 pour les pointeurs. Mais attention, il ne faut jamais déréférencer un pointeur nul. Avant de déréférencer un pointeur, il faut toujours s'assurer qu'il n'est pas nul.

p: pointeur sur entier p ← NIL //p pointe nul part	int *p = NULL;

C.Exemple récapitulatif

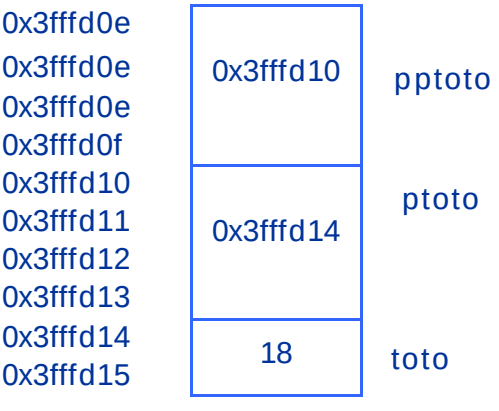
Var
n, x: entiers
p: pointeur sur entier

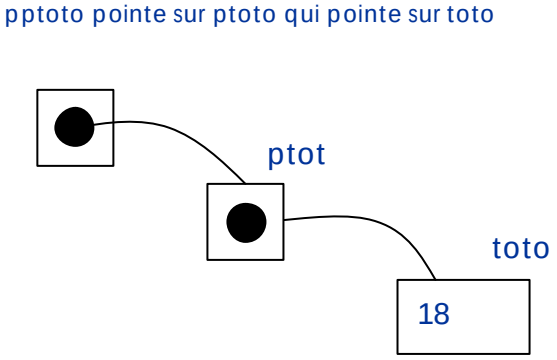
Début ***Etat des variables après exécution***

n ← 20	n 20	x ?	p ?
p ← adresse de n	n 20	x ?	p
x ← *p	n 20	x 20	p
*p ← 30	n 30	x 20	p
p ← adresse de x	n 30	x 20	p

Double indirection

Le fait qu'un pointeur pointe sur une variable s'appelle **indirection** (comme accès **indirect**). Quand un pointeur pointe sur un autre pointeur, il y a double indirection.





On peut accèder à toto par pptoto en utilisant deux fois l'opérateur de déréférencement *

* *pptoto est équivalent à toto et à *ptoto

On pourrait imaginer de la même façon des indirections triples voire quadruples.

D. Transmission de paramètres par adresse

Nous avons vu l'année dernière que pour pouvoir modifier la valeur d'un paramètre dans un sous-programme, il fallait passer ce paramètre non pas "par valeur" mais "par variable" (ou par référence en C++). Il existe une autre manière de passer un paramètre permettant de modifier sa valeur dans un sous-programme: il s'agit du passage par adresse. Nous n'étudierons celui-ci qu'en langage C++.
Le passage par adresse consiste à passer, non pas la valeur ni une référence de la variable paramètre, mais de passer l'adresse de cette variable. Pour cela, lors de l'appel, l'adresse du paramètre effectif est passée grâce à l'opérateur &. Dans l'en-tête de la fonction, le paramètre formel est donc un pointeur, déclaré grâce à l'opérateur *. Et dans le corps de la fonction, lorsqu'on veut manipuler la valeur de la variable pointée (le paramètre effectif), on déréférence le paramètre formel grâce à l'opérateur *;

☞Exemple:
Utilisation d'une fonction void qui permet d'échanger les valeurs de deux paramètres entiers.

```
//prototype de la fonction. Les paramètres formels sont des pointeurs sur entiers
void echange (int *a, int *b);        // a et b sont des pointeurs

main( )
{
int x, y;
x = 10;
y = 20;
cout << "avant appel" << x << y
echange(&x, &y)     // on passe l'adresse des variables à échanger
cout << "après appel" << x << y
}

void echange (int *a, int *b)
{
  int temp;     // variable temporaire pour l'échange
  temp = *a;  // temp prend la valeur pointée par a
  *a = *b;       // la variable pointée par a prend pour valeur celle pointée par b
  *b = temp;  // la variable pointée par b prend pour valeur temp
}
```

Lors de l'appel, la valeur pointée par a est x et la valeur pointée par b est y.

✍ Remarque: La transmission des paramètres se fait toujours par valeur! (la valeur des adresses !) On passe la valeur des expressions &x et &y. Les paramètres effectifs sont ici des adresses, qui sont passées par valeur.

E. Pointeur vers un enregistrement

Il est très courant d'utiliser un pointeur pour mémoriser l'adresse d'un enregistrement. Avec un pointeur sur un enregistrement, on peut accéder aux champs de l'enregistrement pointé en utilisant l'opérateur "->" (moins et supérieur)

☞ Exemple

```
Type
tpersonne = enregistrement
                nom : chaine
                tel : chaine
              finenreg

Var
    pers: tpersonne
    pointpers : pointeur sur tpersonne
Début

  pers.nom ← "Mohamed"
  pers.tel ← "0145698521"
  pointpers ← &pers

  Afficher pointpers -> nom                    donne à l'écran
  Afficher pointpers -> tel                    Mohamed  0145698521
  ...
Fin
```

pointpers->nom *est équivalent à* momo.nom

pointeur sur l'enregistrement -> champ *est équivalent à* enregistrement . champ

F. Lien entre enregistrements grâce à un pointeur

Dans un enregistrement, un des champs peut très bien être un pointeur, par exemple sur un autre enregistrement

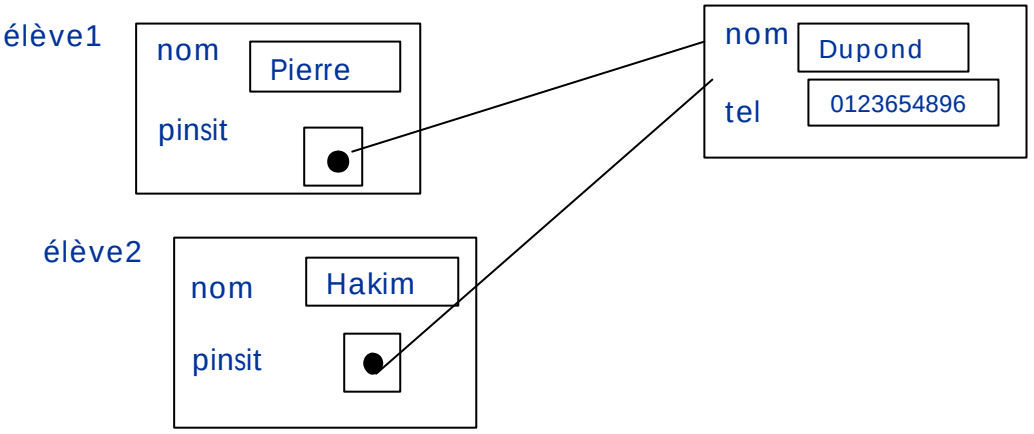
☞ Exemple

```
Type
tpersonne = enregistrement
                nom_ins : chaine
                tel : chaine
              finenreg

télève = enregistrement
           nom_el: chaine
           pinstit : pointeur sur tpersonne
         finenreg
```

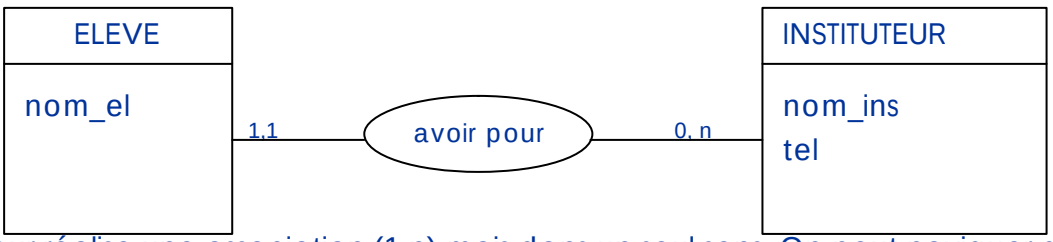
```
Var
  instituteur : tinstit
  élève1, élève2: télève

Début
  instituteur.nom_ins ← "Dupond"
  instituteur.tel ← "0123654896"
  élève1.nom_el ← "Pierre"
  élève1.pinstit ← &insituteur
  élève2.nom_el ← "Hakim"
  élève2.pinstit ← &insituteur
  Afficher élève1.pinstit->nom          // affiche Dupond à l'écran
  Afficher élève2.pinstit->nom          // idem
...
```



A partir d'un élève, on peut connaître les caractéristiques de son instituteur grâce au pointeur contenu dans l'enregistrement de l'élève. Le pointeur réalise un lien entre un élève et son instituteur.
Mais l'enregistrement instituteur ne possède pas de pointeur vers la liste de ses élèves. A partir de l'instituteur, on ne peut pas accéder aux champs des élèves.
Un pointeur est donc un lien univoque (dans un seul sens)
Plusieurs pointeurs peuvent pointer sur un même enregistrement (ici, deux élèves ont le même instituteur), mais un pointeur ne peut pointer que sur une seule variable (enregistrement) à la fois.

✎ Comment représenteriez-vous le lien réalisé par le pointeur grâce au modèle entité-association?



Un pointeur réalise une association (1,n) mais dans un seul sens. On peut naviguer du 1 vers le n mais pas dans le sens inverse.

III. Allocation dynamique de la mémoire

Il y a deux façons d'allouer un emplacement mémoire pour une variable :

- Soit par une **déclaration**, telle qu'on l'a vu jusqu'à présent
- Soit par une **allocation dynamique** explicite, ce que nous allons voir maintenant

La déclaration permet d'allouer un emplacement dans une partie de la mémoire appelée **pile**. Les variables déclarées dans la pile possèdent un nom. Leur portée (c'est-à-dire là où elles peuvent être utilisées) est déterminée par l'endroit où elles sont déclarées. Elles sont crées dès l'entrée dans la portée et détruites dès la sortie de leur portée.

L'allocation dynamique permet d'allouer un emplacement dans une autre partie de la mémoire appelé **tas**. Les variables du tas ne sont pas déclarées, elles sont créées par une instruction spécifique. Elles ne portent pas de nom mais on peut y accéder par leur adresse. Elles peuvent être utilisées partout et jusqu'à leur destruction (il n'y a pas de portée pour les variables du tas).

En algorithmique

Pour créer une variable dans le tas, on utiliser l'opérateur **nouveau**, suivi du type de la variable qu'on veut créer. Cet opérateur renvoie l'adresse de la variable qu'on affecte à un pointeur du même type.

```
Var
p : pointeur sur réel
Début
p= nouveau réel
```

Ensuite on peut accéder à la variable créée en déréférençant le pointeur

```
*p = 12.5
```

Par cette instruction, on affecte 12.5 à la variable du tas.

En C++

Pour créer une variable dans le tas, on utiliser l'opérateur **new**, suivi du type de la variable qu'on veut créer. Cet opérateur retourne l'adresse d'un emplacement mémoire libre, que l'on affecte à un pointeur du même type.

```
float* p ;
p = new float;
```

p contient l'adresse d'un emplacement mémoire dans le tas. Pour accéder à cet emplacement, on déréférence le pointeur p.

```
*p = 12.5 ;
```

On peut déclarer le pointeur et lui affecter l'adresse d'un emplacement du tas en une seule instruction :

```
float* p = new float ;
```

Désallouer la mémoire

En C++, il est nécessaire de désallouer explicitement la mémoire allouée dans le tas. Si la mémoire n'est pas désallouée, il peut arriver qu'elle sature et fasse planter le programme et même l'ordinateur tout entier. En effet, quand plus aucun pointeur ne pointe sur un emplacement alloué du tas, cet emplacement n'est plus accessible pour le système. (En algorithmique, nous supposons qu'il existe un mécanisme permettant de désallouer automatiquement les emplacements qui ne sont plus pointés, comme en Java)
Pour éviter la saturation de la mémoire, il est donc nécessaire de désallouer les emplacements alloués dans le tas. Pour cela, on utilise l'opérateur **delete** en C++ ou **désallouer** en algo, suivi du pointeur qui pointe sur l'emplacement à désallouer.

Algorithmique	C++
désallouer p	delete p ;

- Cette instruction n'efface pas le pointeur mais la variable pointée.
- Ne jamais utiliser cette instruction sur un pointeur qui pointe sur une variable de la pile (un variable nommée)

Les structures de données dynamiques

I. Présentation

Les structures de données permettent de relier entre elles des données. Nous avons déjà étudié les tableaux et les enregistrements.

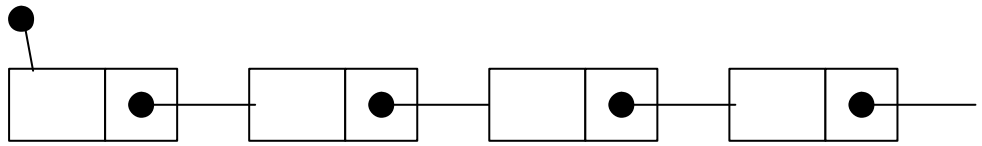
On peut classer les structures de données en deux grandes catégories

- Les structures de données linéaires, qui permettent de relier des données en séquence (on peut numéroter les éléments)
- Les structures de données non linéaires, qui permettent de relier un élément à plusieurs autres éléments

Principales structures de données **LINEAIRES**

- Tableaux
- Listes chaînées
  - à simple chaînage
    - piles
    - files
  - à double chaînage

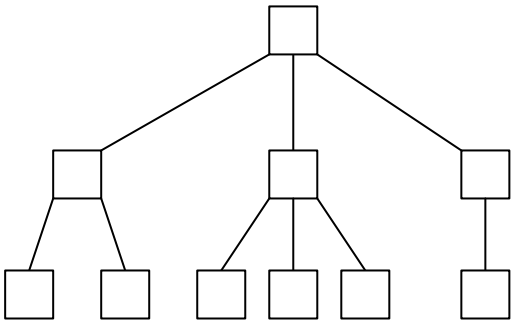
Dans une liste chaînée, les éléments sont reliés grâce à un pointeur. Le pointeur indique l'adresse de l'élément suivant. Les éléments ne sont pas forcément contigus en mémoire, contrairement aux éléments des tableaux.



Principales structures de données **NON LINEAIRES**

- Arbres
  - binaire
  - n-aire
- Graphes
- Enregistrements

Un arbre permet de représenter des structures hiérarchiques. Chaque élément (appelé nœud) a un seul « père » et plusieurs « enfants ».



Un graphe permet de relier des éléments deux à deux de façon quelconque. Cela permet, par exemple, de réaliser des calculs d'optimisation tels que le PERT.

Les structures de données **dynamiques** sont celles dont **les éléments sont mémorisés dans le tas et non dans la pile**. Cela permet de réserver des emplacements dynamiquement à l'exécution et non statiquement à la compilation.
Les structures de données dynamiques sont :

- Les listes chaînées
- Les arbres
- Les graphes

II. Les listes chaînées particulières

A. Les piles [stack]

Présentation

Les piles sont des structures de données, où **l'ajout et le retrait d'un élément se fait toujours au sommet** (en tête de liste).

Le plus souvent, les piles sont implantées sous forme de liste chaînée[1],
Une pile suit la règle **LIFO** (Last In, First Out): dernier entré, premier sorti.

Terminologie particulière:

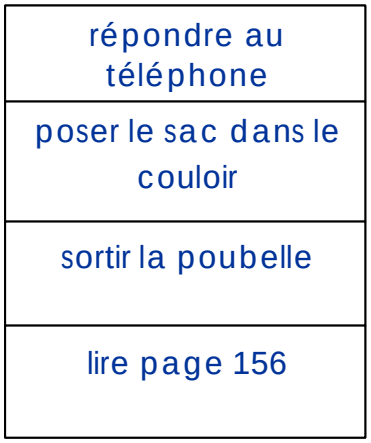
- l'extrémité où s'effectue l'ajout et le retrait s'appelle **sommet**
- ajouter un élément à une pile se dit **empiler** ou **push**
- le fait de retirer un élément d'une pile et de récupérer son information s'appelle **dépiler** ou **pop**

[1] Parfois, une pile est implémentée à partir d'un tableau

Application

Vous lisez tranquillement un livre chez vous quand votre père vous demande de sortir la poubelle. Vous marquez la page de votre livre et vous vous dirigez vers la poubelle. Pendant que votre marchez vers la porte avec le sac, un coup de téléphone vous interrompt. Vous posez le sac dans le couloir, puis vous allez répondre au téléphone. Quand le coup de fil est terminé, vous vous rappelez où il faut que vous récupériez le sac pour le sortir. Vous le sortez et revenez à votre livre, à la page que vous avez marquée et continuez votre lecture.

On peut représenter cette scène par une pile qui sera empilée puis dépilée au cours du déroulement des événements.



C'est sur le même principe que fonctionne un programme.

A chaque fois qu'on appelle une fonction ou procédure, on impose au programme d'arrêter l'exécution du code en cours et de se brancher sur le sous-programme. Le programme doit enregistrer l'endroit où il se trouve dans le programme appelant et la valeur des variables à ce moment avant de se brancher vers le sous-programme appelant. Il enregistre ces informations dans une pile. Au retour d'appel du sous-programme, la pile est dépilée, ainsi les informations de l'état avant appel sont récupérées.

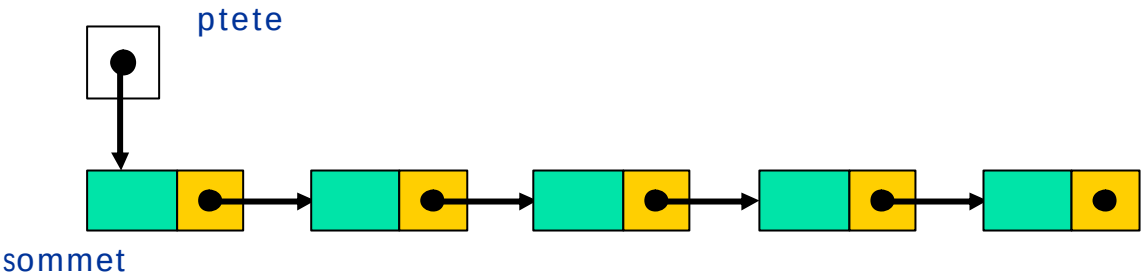
C'est donc grâce à une pile que le programme gère l'appel en cascade de plusieurs sous-programmes. Cette pile est gérée automatiquement par le système d'exploitation, le programmeur n'a pas à s'en préoccuper.

Exemple:
Un programme A appelle un sous-programme B qui appelle lui-même un sous-programme C qui appelle D. La pile du programme va évoluer comme suit:

lancement de A	pile vide
appel de B par A	l'état de A est empilé
appel de C par B	l'état de B est empilé sur A
appel de D par C	l'état de C est empilé sur B
Fin de D	dépilage de l'état de C → poursuite de C
Fin de C	dépilage de l'état de B → poursuite de B
Fin de B	dépilage de l'état de A → poursuite de A

Représentation sous forme de liste chaînée

On ne peut manipuler une pile qu'à travers son sommet. Le sommet d'une pile représentée sous forme de liste chaînée est la tête de cette liste. Le seul pointeur sur la liste est donc le pointeur de tête. La création d'un pile se fait "à l'envers": le dernier élément créé se retrouve en tête (au sommet).



B. Les files ou queues [queue]

Présentation

Une file ou queue est une structure de données où l'insertion d'un élément se fait à une extrémité appelée queue et le retrait d'un élément se fait à une autre extrémité appelée tête.

Une file est le plus souvent représentée par une liste chaînée.
Une file suit la règle **FIFO** (First In, First Out): premier entré, premier sorti

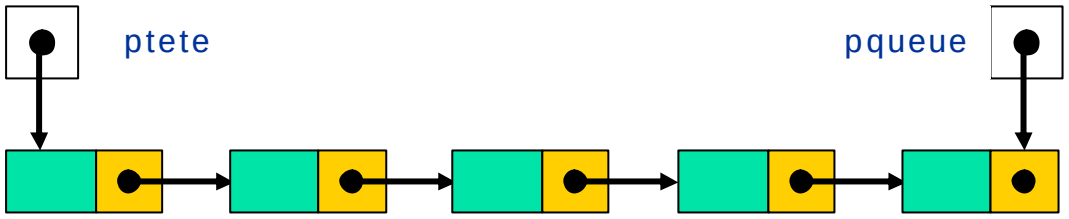
Application

Les queues servent à traiter des données dans l'ordre où elles arrivent, comme dans une file d'attente à un guichet où le premier arrivé est le premier servi.

Représentation sous forme de liste chaînée

Une file doit être accessible à la fois par la tête pour retirer un élément et par la queue pour ajouter un nouvel élément. La tête et la queue d'une file correspondent à la tête et la queue de la liste chaînée qui l'implémente.

Dans une file, le premier élément créé doit être en tête et le dernier élément créé doit être en queue. Or, si l'on n'utilise qu'un seul pointeur ptete pour créer la liste, le premier élément créé se retrouve en queue.
Comment faire pour résoudre ce problème? On va utiliser deux pointeurs sur la liste: un pointeur sur la tête et un pointeur sur la queue. **La création se fera par ajout successif en queue** (alors qu'une pile est créée par insertion successive en tête).



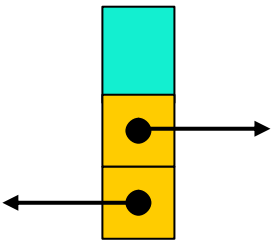
Les variantes de files

Il existe plusieurs variantes de files, notamment les files à deux queues. Dans les files à deux queues, l'entrée peut se faire en queue mais aussi en tête de file. Cela permet de gérer des données exceptionnelles qui doivent être traitées avant toutes les autres.

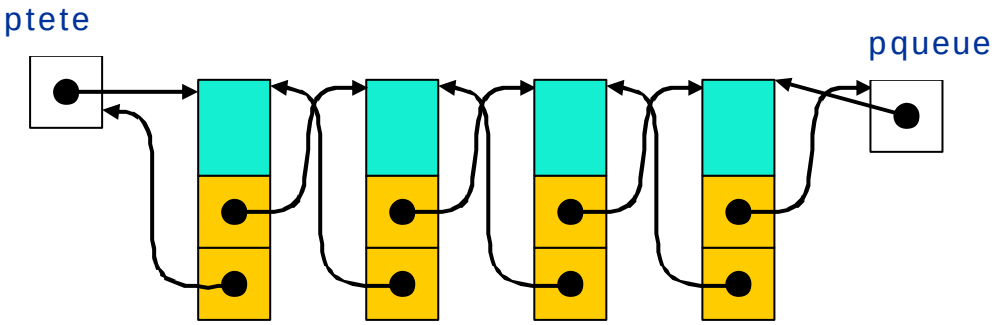
C. Les listes doublement chaînées

Les listes doublement chaînées sont des listes où chaque élément pointe sur son suivant et sur son précédent. Le parcours d'une telle liste peut se faire dans les deux sens. Ce type de liste facilite en outre l'ajout d'un élément au milieu de la liste de sorte que celle-ci reste triée. Le premier élément possède un pointeur qui pointe sur la tête. Le dernier élément possède un pointeur qui pointe sur la queue.

Telem : **enregsitrement**
    info : type de l'info
    psuiv : pointeur sur Telem
    pprec : pointeur sur Telem
Fin **enregistrement**



Remarque1 : Les pointeurs doivent toujours être les derniers champs déclarés.



Remarque 2: ptête est pqueue sont déclarés dans la pile (et non dans le tas)

Création d'une liste chaînée simple (pile)

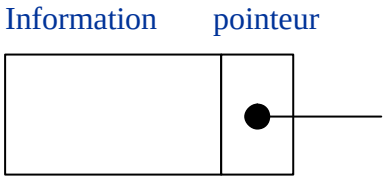
Rappel

Une liste chaînée est une collection linéaire d'éléments d'information. A chaque élément est associé un pointeur qui contient l'adresse de l'élément suivant dans la liste.

Chaque élément est de type enregistrement composé d'au moins deux champs :

- un ou plusieurs champs contiennent l'information
- le dernier champ contient le pointeur sur l'élément suivant

Le pointeur est du type enregistrement de l'élément



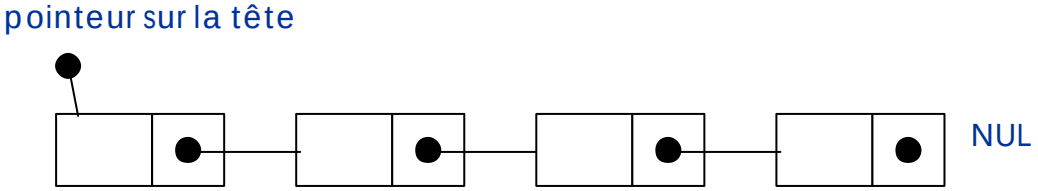
Exemple :
Nous allons réaliser une liste chaînée permettant de mémoriser un répertoire téléphonique simplifié. Chaque personne est caractérisée par un nom et un numéro de téléphone.

Nous déclarons le type des éléments de la liste chaînée

```
Type
tElément :  Enregistrement
                Nom : chaine
                Tel : chaîne
                pSuivant : pointeur sur Elément
            Finenreg
```

Les éléments d'une liste chaînée sont créés dans le tas. Ainsi, le nombre d'élément de la liste n'est pas déterminé à la compilation comme avec les tableaux. Les éléments sont crées au fur et à mesure des besoins et ainsi il n'y a pas de perte d'espace mémoire.

Les éléments d'une liste chaînée ne sont donc pas nommés. Ils sont accessibles par le pointeur contenu dans l'élément qui les précède.
Le premier élément, appelé **tête**, n'a pas de précédant. Pour mémoriser son adresse, on déclare un pointeur à part. Grâce à ce pointeur, on peut accéder au premier élément, qui permet d'accéder au deuxième, qui permet d'accéder au troisième, etc … Ainsi, à partir du pointeur sur le premier élément, on peut parcourir toute la liste.
Le dernier élément ne pointe sur rien. Son pointeur est nul.



La création d'une liste chaînée simple (de type pile)

Pour des raisons de simplicité, la création d'une liste chaînée se fait «à l'envers » c'est-à-dire du premier au dernier élément. On crée en dernier l'élément de tête.

Etape de création d'une liste

1. Déclarer un pointeur sur la tête de la liste et un pointeur courant (pointant les autres éléments en cours de création)
2. Créer un premier élément et affecter son adresse à pel
3. Initialiser les infos du premier élément
4. Initialiser son pointeur sur suivant à NIL (ou NULL en C++) car cet élément sera le dernier et donc n'aura pas de suivant
5. Affecter cet élément au pointeur ptete (le dernier élément créé est toujours la tête et ainsi on peut réutiliser le pointeur courant pour créer un nouvel élément)
6. Créer un deuxième élément et affecter son adresse au pointeur courant
7. Initialiser les infos de cet élément
8. L'élément en cours devient la tête. En effet, le dernier élément crée est toujours la tête.
9. Recommencer à l'étape 6 pour les autres éléments

```
Var
ptête : pointeur sur tElément   //Etape 1
pel: pointeur sur tElément

Début
/*création du premier élément (l'élément qui sera en queue)*/
pel← nouveau tElément          // Etape 2
pel ->Nom ← « Jalila »                  // Etape 3
pel ->Tel ← « 0146598632 »
pel->pSuivant ← NIL            //Etape 4
pel← ptete                              //Etape 5

/*création du deuxième élément*/
pel ← nouveau tElément                  //Etape 6
pel ->Nom ← « Pierrot »                 //Etape 7
pel ->Tel ← « 0635987541 »
pel->pSuivant ← ptête                   //Etape 8

/*Création d'un autre élément*/
pel← nouveau tElement
pel->Nom ← …
…
pel->pSuivant ← ptête
```

Algorithmes sur une liste doublement chaînée triée

On va créer une liste doublement chaînée d'éléments entiers où les ajouts s'effectuent de façon à ce que l'ordre reste croissant.
On va écrire les sous-programmes qui permettent:

- d'ajouter un élément
- de supprimer un élément

➢ **déclaration du type des éléments**

Telem : **enregistrement**
    info: entier
    psuiv: pointeur sur Telem //pointeur sur l'élément suivant
    pprec : pointeur sur Telem //pointeur sur l'élément précédent
FinEnregistrement

➢ **déclarations à faire dans le programme appelant**

Var
// déclaration des pointeurs utiles pour accéder à la liste
ptete: pointeur sur Telem
pqueue: pointeur sur Telem
…

➢ **Sous programme de suppression d'un élément**

On suppose qu'on dispose d'un pointeur (passé en paramètre)sur l'élément à supprimer.

On est certain que ce pointeur pointe sur un élément de la liste. Pour y accéder, on n'a pas besoin de connaître la tête. En revanche, on a besoin de connaître la tête et la queue pour savoir si l'élément à supprimer n'est pas justement en tête ou en queue de liste.

Trois cas peuvent se présenter:

- soit l'élément à supprimer est un élément situé au milieu de la liste (cas général)
- soit l'élément à supprimer est le premier → la tête est modifiée
- soit l'élément à supprimer est le dernier → la queue est modifiée

Procédure suppr(E psuppr : pointeur sur Telem, E/S ptete, pqueue: pointeur sur Telem)
Var
//pointeurs sur le précédent et le suivant de l'élément à supprimer
    ppreced, psuivant : pointeur sur Telem
Début
ppreced ← (*psuppr).pprec
psuivant ←(*psuppr).psuiv
Si ppreced = ptete **Alors** *//suppression en tete*
    ptete ← (*psuppr).psuiv
    (*psuivant).pprec ← ptete
Sinon si psuivant = pqueue **Alors** *//suppression en queue*
    pqueue ← ppreced
    (*ppreced).psuiv ← pqueue
Sinon *//suppression au milieu*
    (*ppreced).psuiv ← psuivant
    (*psuivant).pprec ← ppreced
Finsi
Finsi
//En C++, il faudrait ici détruire l'élément pointé par psuppr (delete psuppr)
Fin

➢ Sous-programme d'ajout d'un élément

Analyse préalable:

- Quelles sont les données:

l'entier à ajouter

la liste→ en entrée a-t-on besoin des deux pointeurs? Non, on peut accéder à toute la liste seulement avec le pointeur de tête.

- Quels sont les résultats

la nouvelle liste → elle peut avoir une nouvelle tête ou une nouvelle queue → paramètres en sortie

- L'entier doit être inséré à l'endroit où il devient supérieur aux autres éléments de la liste. On va déclarer et utiliser un pointeur (pcour) pour parcourir la liste à la recherche de l'endroit où il faut insérer.

- Evidemment, l'entier e est inséré à l'intérieur d'un nouvel élément Telem → il faut déclarer un pointeur (pnv) pour créer ce nouvel élément

- Il faut veiller à ce programme fonctionne dans tous les cas de figure

- l'élément est ajouté en milieu de liste (cas général)
- la liste est vide → l'élément pointe sur ptete et pqueue
- l'élément doit être ajouté en tête → modification de ptete
- l'élément doit être ajouté en queue → modification de la queue

```
Procédure ajout(E e: entier, E/S ptete: pointeur sur Telem, S pqueue: pointeur sur Telem)
Var
    pnv, pcour : pointeurs sur Telem

Début
// création d'un nouvel élément pour ajouter e
pnv ← nouveau Telem
(*pnv).info ← e

// recherche de l'élément qui est immédiatement supérieur à e
pcour ← ptete
Tantque pcour ? NULL et (*pcour).info <= e Faire
   pcour ← (*pcour).psuiv
FinTantque
/* à la sortie de cette boucle, soit on est arrivé à la fin de la liste, soit on pointe sur l'élément qui est
immédiatement supérieur à e. C'est juste avant cet élément que l'on veut insérer e. */

Si pcour = ptete Alors          // si l'insertion doit se faire en tête
        ptete ← pnv
        (*pvn). prec ← ptete
        (*pnv).psuiv ← pcour
        (*pcour).pprec ← pete
Sinon Si pcour = NULL Alors     //on est arrivé à la queue de la liste, l'insertion se fait en queue
        pqueue ← pnv
        (*pnv).pprec ← pqueue
        (*pnv).psuiv ← pcour
        (*pcour). psuiv ← pqueue
Sinon       //l'insertion se fait au milieu de la liste
        pelemprec ←(*pcour).pprec     /*on mémorise l'élément précédent de l'élément courant. Ce sera
        l'élément précédent de celui qu'on ajoute*/
        (*pelemprec).psuiv ← pnv
        (*pcour).pprec ← pnv
        (*pnv).psuiv ← pcour
        (*pnv).pprec ←pelemprec
Finsi
Finsi
Fin
```

SQL Intégré (à l'algorithmique)

Présentation

SQL Intégré est une technique permettant d'introduire des instructions SQL (seulement du LMD) dans un programme en langage classique dans un environnement non client-serveur.

Le programme qui contient des instructions SQL est appelé programme hôte.
La communication entre le programme hôte (non SQL) et les instructions SQL passe par des variables communes : ce sont les variables hôtes. Les variables hôtes peuvent être à la fois utilisées par le programme hôte et par les instructions SQL.

Ces variables particulières sont déclarées dans une section particulière du programme hôte, après les variables "traditionnelles" qui ne seront pas utilisées par SQL. En algorithmique, on fait commencer cette section tout simplement par l'expression Variable(s) hôte(s).

Les variables hôtes sont utilisées par le programme hôte comme toutes les autres variables. Dans les instructions SQL, les variables hôtes utilisées sont précédées d'un préfixe **(:** en algo, @ avec SQL Server) pour les différencier des noms d'attributs des tables.

Pour les distinguer du reste du code, les instructions SQL sont entourées par les mots clé SQL et finSQL. Si l'instruction SQL ne fait qu'une ligne, il suffit de placer SQL au début de la ligne (pas besoin alors de finSQL).

Utilisation de SQL intégré

❑ Pour traiter un t-uple, résultat d'une requête

Pour placer le résultat d'une requête à l'intérieur d'une variable hôte, on utilise la clause INTO suivi de la (ou des) variables hôtes à valoriser. La requête dont on veut extraire le résultat peut être paramétrée par une ou plusieurs variables hôtes.

```
SQL
    SELECT attribut1 INTO :variable_hôte1,  attribut2 INTO :variable_hôte2, …
    FROM table(s)
    WHERE attribut = :variable_hôte
finSQL
```

La clause INTO ne fonctionne que si la requête ne retourne qu'une seule ligne. Si la requête ne retourne aucune ligne ou si au contraire elle retourne plusieurs lignes, l'affectation ne va pas pouvoir se faire : une erreur va apparaître. Il faut traiter ces cas, soit en faisant l'hypothèse qu'il n'y a jamais d'erreur, soit en indiquant les traitements à réaliser le cas échéant. Les erreurs sont signalées par le SGBD au programme hôte par l'intermédiaire d'une variable hôte spéciale, avec un nom spécifique (SQLCODE), qui prend une valeur spécifique (100) dans le cas où la requête précédente n'a renvoyé aucune ligne. Pour les autres types d'erreurs, il n'existe pas de codification standard : soit la notation est précisée dans l'énoncé, soit c'est à vous de préciser celle que vous utiliser.

❑ Pour mettre à jour une base de données

On peut insérer des t-uples dans une base de données à partir de données tirées du programme hôte.

SQL
INSERT INTO *table*
VALUES :*variable_hôte1*, :*variable_hôte2*, …
finSQL

On peut aussi modifier des t-uples à partir de données tirées du programme hôte.

SQL
UPDATE *table*
SET *attribut = :variable_hôte*
WHERE *attribut = :variable_hôte*
AND …
finSQL

Exemple complet

Soit le schéma relationnel simplifié suivant :

CLIENT (clt_num, clt_nom, clt_prenom, clt_ca)
COMMANDE(com_clt_num #, com_prod_ref# , com_qte)
PRODUIT(prod_ref, prod_libelle, prod_prix)

Pour simplifier, nous supposons qu'une commande ne porte que sur un seul produit.
Le numéro de client est un entier séquentiel

Nous allons écrire un algorithme utilisant SQL intégré pour traiter l'arrivée d'une commande. Les informations saisies pour une commande sont le nom et le prénom du client ainsi que la référence et la quantité du produit désiré. L'algorithme doit permettre de mettre à jour les tables client (maj du chiffre d'affaire) et commande (insertion d'une nouvelle commande dans la base). Si le client n'existe pas, il est nécessaire de le créer. On suppose que deux clients ne peuvent pas porter le même nom et le même prénom.

Programme exSQLintégré

VARIABLES :
//déclarer ici les variables utilisées seulement dans le programme hôte et pas dans les
//instructions SQL. Il n'y a pas de telles variables ici.

VARIABLES HOTES :
nom, prenom, refprod : chaînes
qte, num : entier
SQLCODE : entier //variable permettant de communiquer les erreurs de SQL au programme
ca : réel

DEBUT
Afficher "Nom et prénom du client : "
Saisir nom, prenom
Afficher "Référence du produit et quantité désirée"
Saisir refprod, qte

```
SQL
    SELECT clt_num INTO :num
    FROM CLIENT
    WHERE clt_nom = :nom AND clt_prenom = :prenom
finSQL
Si SQLCODE = 100 Alors // aucune ligne retournée par la requête
   //création du nouveau client
   SQL
       SELECT max(clt_num) INTO :num      //on récupère le plus grand numéro de client
       FROM CLIENT;
  finSQL
  num ← num + 1 // le numéro du nouveau client est le numéro du dernier + 1
  SQL
       INSERT INTO CLIENT                 //on ajoute le nouveau client dans la table CLIENT
       VALUES :num, :nom, :prenom, 0;   //le chiffre d'affaire est initialisé à 0 (maj plus bas)
   finSQL
FinSi
//mise à jour du chiffre d'affaire du client
//il faut d'abord calculer le montant de la commande
// on cherche le prix du produit
SQL
  SELECT prod_prix INTO :ca
  FROM PRODUIT
  WHERE prod_ref = :refprod
finSQL
Si SQLCODE = 100 Alors  //si le résultat de la requête n'a aucune ligne
  Afficher "Erreur : la référence du produit n'existe pas"
  Sortir
FinSi
// on calcule le chiffre d'affaire de la commande= prix (résultat de la requête)* quantité
ca ← ca * qte
// on met à jour le chiffre d'affaire du client
SQL
  UPDATE CLIENT
  SET clt_ca = clt_ca + :ca
  WHERE clt_num = :num
finSQL
//ajout de la commande dans la table COMMANDE
SQL
  INSERT INTO COMMANDE
   VALUES :num, :refprod, :qte
finSQL

 SQL COMMIT WORK
  //indique la fin de la transaction (les mises à jour deviennent permanentes)
FIN
```

❑ les curseurs : pour traiter le résultat d'une requête contenant plusieurs tuples

Rappel sur les curseurs :

Un curseur représente un ensemble de tuple résultat d'une requête.
Curseur et jeu d'enregistrement sont synonymes. La différence entre ces deux notions provient du langage qui les manipule. Les curseurs sont manipulés par des instructions en langage SQL, alors que les jeux d'enregistrement sont manipulés par des instructions spécifiques du langage hôte (ex: les RecordSet en VB).
La manipulation des curseurs suit le standard SQL alors que la manipulation des jeux d'enregistrement dépend des langages. (A l'étude de cas du BTS, lorsque ce n'est pas précisé dans l'énoncé, et que vous devez traiter le résultat d'une requête comportant plusieurs tuples, utilisez les curseurs et la notation SQL).

On peut comparer une curseur à une sorte de fichier séquentiel défini par un ordre SQL, qu'il est possible d'ouvrir (OPEN) et de lire en séquentiel (FETCH).

Pour utiliser un curseur :

- DECLARATION

Déclarer le ou les curseurs utilisés dans une section de déclaration spécifique intitulée Curseur(s). Attention, la déclaration n'entraîne pas l'exécution de la requête associée.

```
Curseur(s) :

DECLARE moncurseur CURSOR FOR
SELECT …
```

- OUVERTURE

Elle **permet d'exécuter la requête** correspondant au curseur. Le résultat de la requête est affecté au curseur, et le "pointeur" du curseur est positionné juste avant le premier tuple.

```
SQL OPEN nomcurseur
```

- LECTURE

La lecture par l'instruction FETCH permet de copier le tuple suivant (tuple courant) dans les variables hôtes déclarées préalablement (à chaque champ du tuple doit correspondre une variable hôte). Lorsqu'il n'y a plus de tuple à lire, l'instruction FETCH affecte la valeur 100 à la variable SQLCODE.

```
//lecture du premier tuple
SQL FETCH moncurseur INTO :variablehote1, : variablehote2, :variablehote3
Tantque SQLCODE ? 100 Faire                    // tant que la lecture est possible
        // traitement sur le tuple par l'intermédiaire des variables hôtes
        …
        //puis lecture du tuple suivant
        SQL FETCH moncurseur INTO :variablehote1, : variablehote2, :variablehote
FinTq
```

- FERMETURE

Il faut fermer les curseurs (tout comme il faut fermer les fichiers), pour désallouer les ressources mémoires utilisées.

```
SQL CLOSE moncurseur
```