Compléments d'algorithmique

L1 MPCIE

Pointeurs

Allocation dynamique

Introduction aux structures récursives

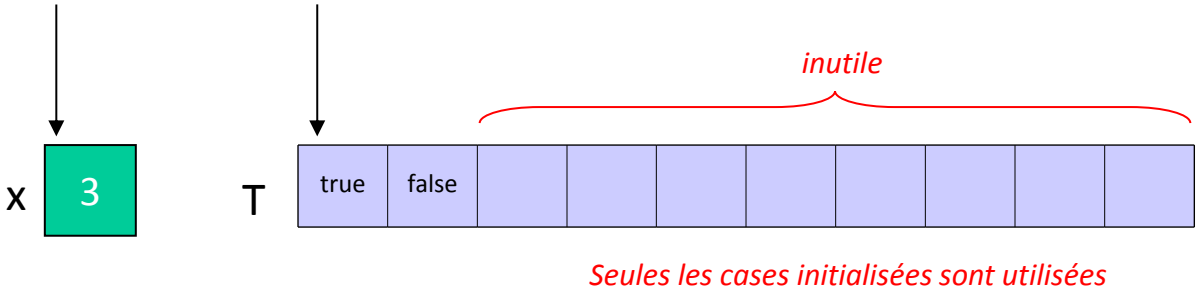
Variables statiques / dynamiques

Les variables utilisées jusqu'ici étaient **statiques** :

- elles sont toujours **déclarées** en tête du programme ou du bloc dans lequel elles sont initialisées
- elles occupent une place mémoire qui leur est allouée pendant toute l'exécution du programme ou du bloc
- leur adressage est **direct** : elles sont accessibles directement par leur identifiant

Exemple :

```
using tab_booleen = std::array <bool,10>;
…
int x ;     tab_booleen T;
```



La place mémoire de x et T est allouée dès la déclaration et jusqu'à la fin du programme ou sous-programme.

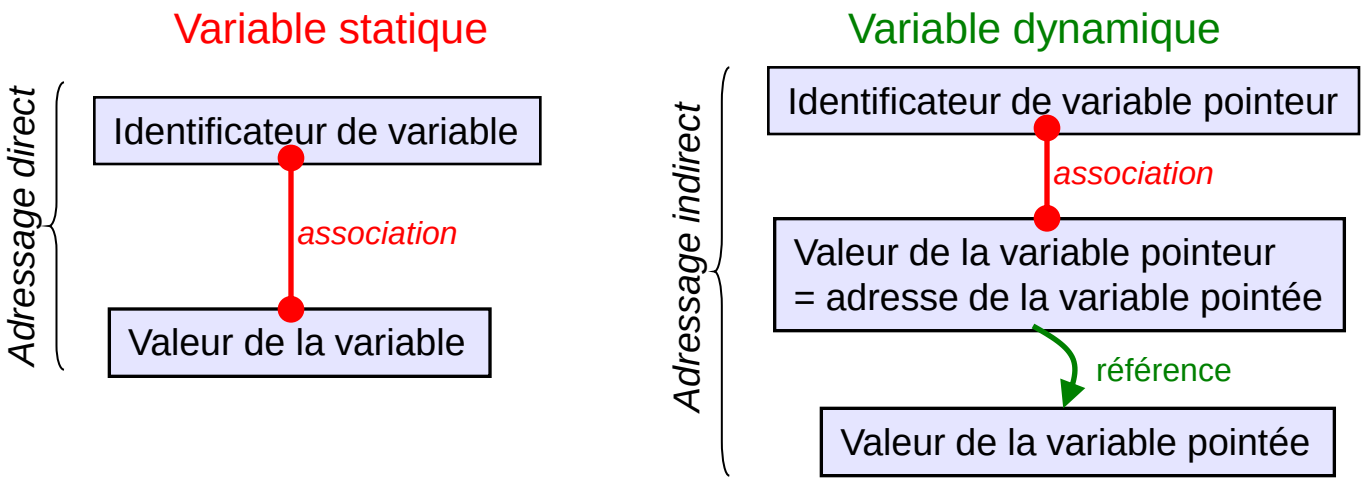
Variables statiques / dynamiques

Contrairement aux variables statiques, les variables **dynamiques** :

- peuvent être créées et utilisées uniquement au fur et à mesure des besoins
- peuvent être détruites à tout moment afin de récupérer l'espace mémoire devenu inutile
- permettent l'insertion et la suppression d'éléments sans toucher au reste des données

En revanche les variables dynamiques ne peuvent être adressées directement par un identificateur. Elles sont accessibles par l'intermédiaire d'une variable statique spéciale (dite *pointeur*) contenant l'adresse mémoire de la variable dynamique.

On dit que le pointeur *pointe* vers cette variable dynamique ou que cette variable dynamique *est pointée* par le pointeur.

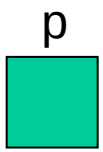


L'adressage indirect permet une gestion dynamique de la mémoire.

Manipulation des pointeurs

Exemple de définition d'un pointeur :

```
std::string *p;
```



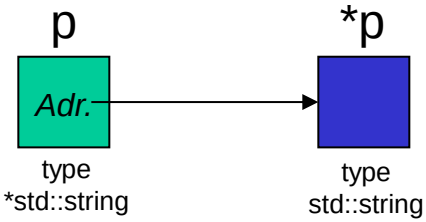
std::string* est le type de variable pointeur qui pointe vers une variable de type std::string.

*p est de type std::string** | ***p est de type std::string***

Il est possible de définir un type pchaine = std::string*, puis de déclarer une variable p de type pstring.

Pou créer une variable dynamique pointée :

```
p = new std::string;
```



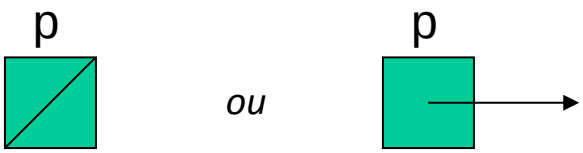
L'instruction **new** réserve un emplacement mémoire pour la variable dynamique pointée, et affecte au pointeur p (variable statique pointeur du type défini, par exemple *std::string) la valeur de l'adresse de l'emplacement mémoire réservé pour la variable dynamique pointée.

Si **p** est la variable statique pointeur,
la variable dynamique pointée sera désignée par ***p**

Manipulation des pointeurs

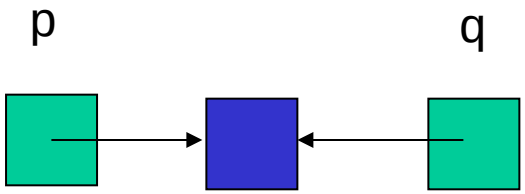
Il existe deux types d'**affectation** pour les pointeurs :

```
p = NULL;
```



NULL est la constante pointeur vide, ou adresse nulle, valeur non indéterminée qui signifie que la variable ne pointe sur aucune variable dynamique.

```
p = q;
```



On suppose que p et q ont été déclarées et sont deux variables pointeurs de même type (pointant sur des variables de même type).
Si p $\neq$ NULL et tant que p et q ne sont pas modifiées, *p et *q sont deux manières d'accéder à la même variable, puisque les deux pointeurs p et q contiennent la même adresse.

Pour supprimer une variable dynamique pointée :

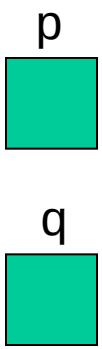
```
delete p;
```

L'instruction **delete** libère l'emplacement préalablement occupée par *p.
Attention : après un delete, la valeur de p n'a plus de signification et devra être réinitialisée avant d'être utilisée.

Manipulation des pointeurs

Exemple 1 :

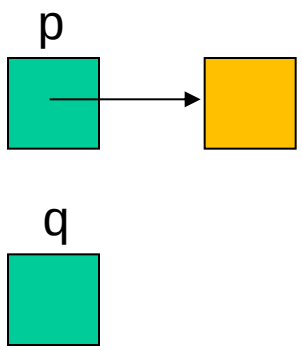
```
int *p, *q;
```



Manipulation des pointeurs

Exemple 1 :

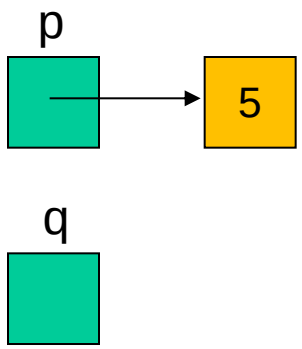
```
int *p, *q;
p = new int;
```



Manipulation des pointeurs

Exemple 1 :

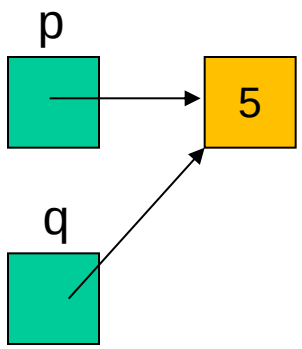
```
int *p, *q;
p = new int;
*p = 5;
```



Manipulation des pointeurs

Exemple 1 :

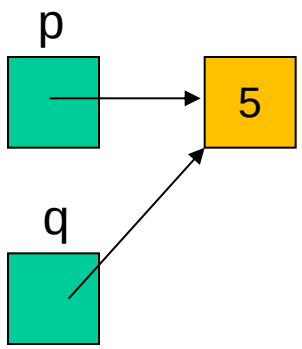
```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Manipulation des pointeurs

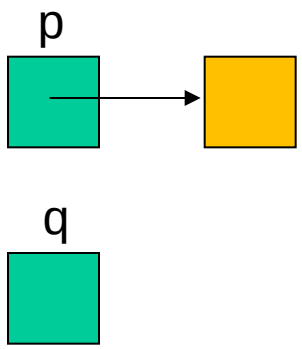
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

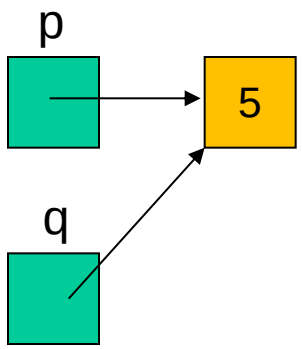
```
int *p, *q;
p = new int;
```



Manipulation des pointeurs

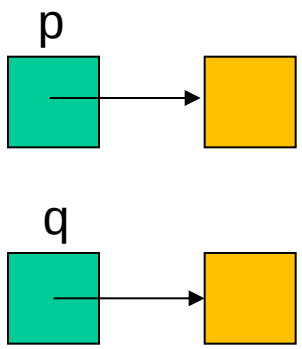
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

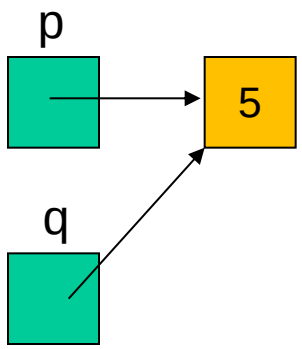
```
int *p, *q;
p = new int;
q = new int;
```



Manipulation des pointeurs

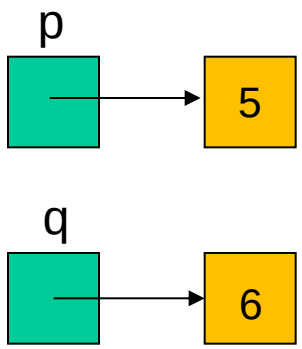
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

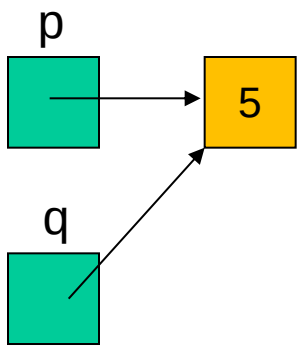
```
int *p, *q;
p = new int;
q = new int;
*p = 5;
*q = 6;
```



Manipulation des pointeurs

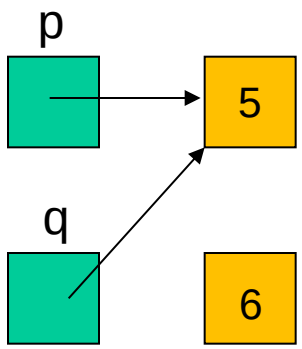
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

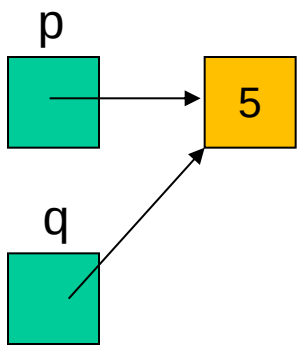
```
int *p, *q;
p = new int;
q = new int;
*p = 5;
*q = 6;
q = p;
```



Manipulation des pointeurs

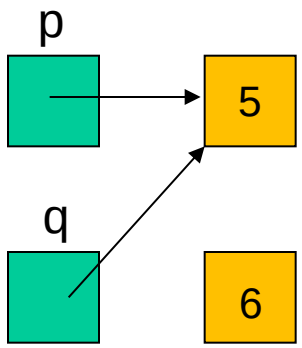
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

```
int *p, *q;
p = new int;
q = new int;
*p = 5;
*q = 6;
q = p;
```



Attention :

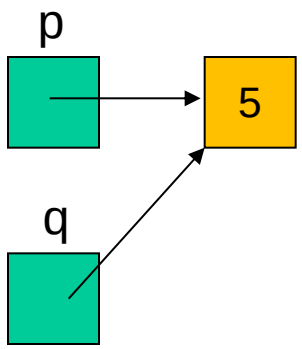
*- l'ancienne valeur de *q n'est plus accessible*

- cette variable dynamique ne pourra plus être supprimée (la mémoire ne pourra pas être désallouée)

Manipulation des pointeurs

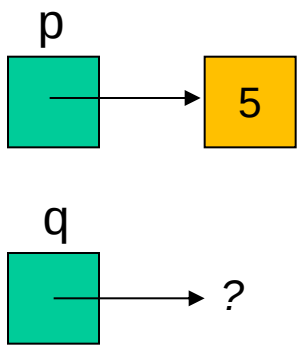
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

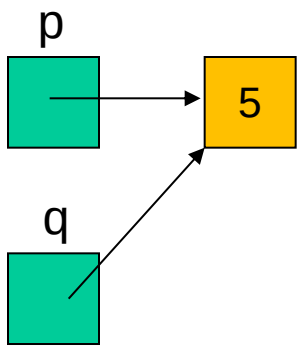
```
int *p, *q;
p = new int;
q = new int;
*p = 5;
*q = 6;
delete q;
```



Manipulation des pointeurs

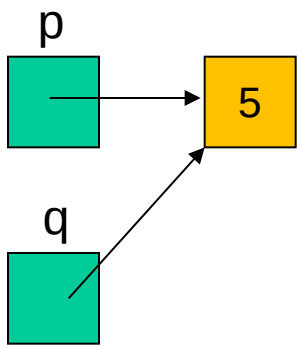
Exemple 1 :

```
int *p, *q;
p = new int;
*p = 5;
q = p;
```



Exemple 2 :

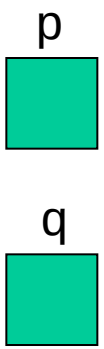
```
int *p, *q;
p = new int;
q = new int;
*p = 5;
*q = 6;
delete q;
q = p;
```



Manipulation des pointeurs

Exemple 3 :

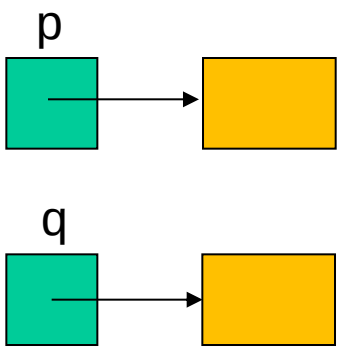
```
std::string *p, *q;
```



Manipulation des pointeurs

Exemple 3 :

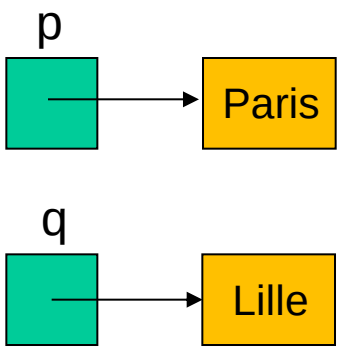
```
std::string *p, *q;
p = new std::string;
q = new std::string;
```



Manipulation des pointeurs

Exemple 3 :

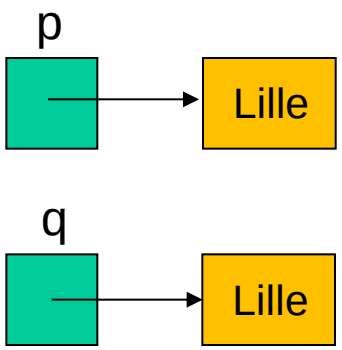
```
std::string *p, *q;
p = new std::string;
q = new std::string;
*p = "Paris";
std::cin >> *q; //Lille sera entré au clavier
```



Manipulation des pointeurs

Exemple 3 :

```
std::string *p, *q;
p = new std::string;
q = new std::string;
*p = "Paris";
std::cin >> *q; //Lille sera entré au clavier
*p = *q;
```



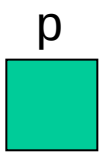
Ici *p contient la même valeur que *q, mais les pointeurs p et q contiennent des valeurs différentes.

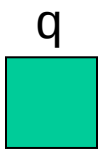
Ne pas confondre *p = *q et p = q

Manipulation des pointeurs

Exemple 4 :

```
struct fiche {
     std::string nom;
     int note;
};

int main () {
   fiche *p, *q, *best;
```

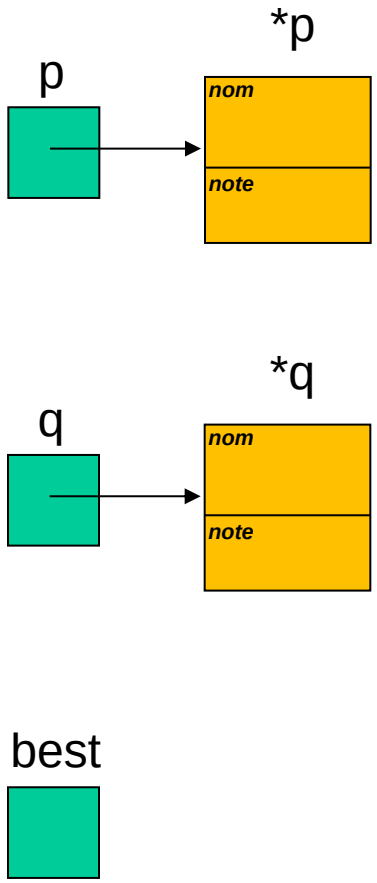




Manipulation des pointeurs

Exemple 4 :

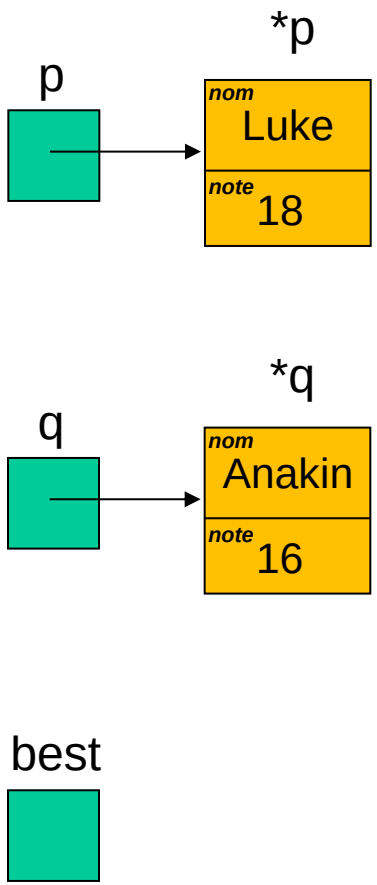
```
struct fiche {
     std::string nom;
     int note;
};

int main () {
   fiche *p, *q, *best;
   p = new fiche;
   q = new fiche;
```



Manipulation des pointeurs

Exemple 4 :

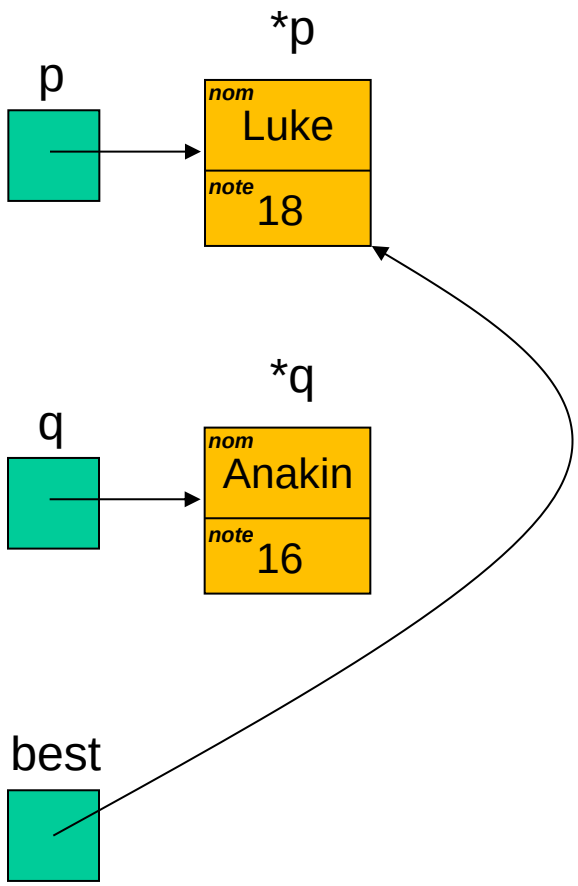
```
struct fiche {
     std::string nom;
     int note;
};

int main () {
   fiche *p, *q, *best;
   p = new fiche;
   q = new fiche;

   (*p).nom = "Luke";
   (*p).note = 18;
   (*q).nom = "Anakin";
   (*q).note = 16;
```



Manipulation des pointeurs

Exemple 4 :

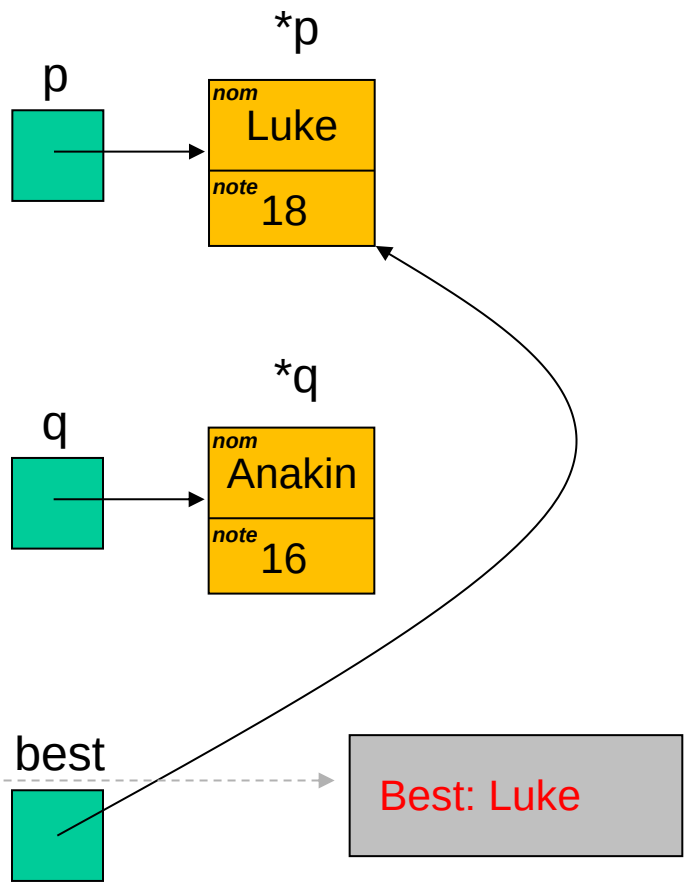
```
struct fiche {
     std::string nom;
     int note;
};

int main () {
   fiche *p, *q, *best;
   p = new fiche;
   q = new fiche;

   (*p).nom = "Luke";
   (*p).note = 18;
   (*q).nom = "Anakin";
   (*q).note = 16;

   if ((*p).note > (*q).note)
      best = p;
   else best = q;
```



Manipulation des pointeurs

Exemple 4 :

```
struct fiche {
     std::string nom;
     int note;
};

int main () {
   fiche *p, *q, *best;
   p = new fiche;
   q = new fiche;

   (*p).nom = "Luke";
   (*p).note = 18;
   (*q).nom = "Anakin";
   (*q).note = 16;

   if ((*p).note > (*q).note)
      best = p;
   else best = q;
   std::cout << "Best: " << (*best).nom;

   return 0;
end.
```

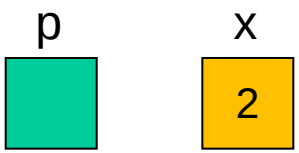


Pointeurs et variables statiques

Il est possible d'affecter à une variable pointeur l'adresse d'une variable statique, déjà définie.
Dans ce cas, il n'y a pas d'allocation de variable dynamique.
On récupère l'adresse d'une variable à l'aide du symbole **&**.

Exemple :

```
void inc (int *p)
{
   ++(*p);
}

int main ()
{
   int x, *p;
   x = 2;
```

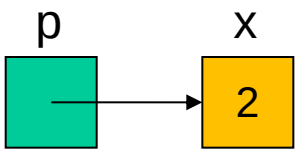


Pointeurs et variables statiques

Il est possible d'affecter à une variable pointeur l'adresse d'une variable statique, déjà définie.
Dans ce cas, il n'y a pas d'allocation de variable dynamique.
On récupère l'adresse d'une variable à l'aide du symbole **&**.

Exemple :

```
void inc (int *p)
{
   ++(*p);
}

int main ()
{
   int x, *p;
   x = 2;
   p = &x;
```

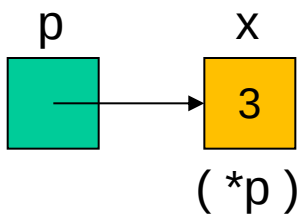


Pointeurs et variables statiques

Il est possible d'affecter à une variable pointeur l'adresse d'une variable statique, déjà définie.
Dans ce cas, il n'y a pas d'allocation de variable dynamique.
On récupère l'adresse d'une variable à l'aide du symbole **&**.

Exemple :

```
void inc (int *p)
{
   ++(*p);
}

int main ()
{
   int x, *p;
   x = 2;
   p = &x;
   inc(p);
```

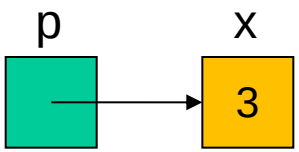


Pointeurs et variables statiques

Il est possible d'affecter à une variable pointeur l'adresse d'une variable statique, déjà définie.
Dans ce cas, il n'y a pas d'allocation de variable dynamique.
On récupère l'adresse d'une variable à l'aide du symbole **&**.

Exemple :

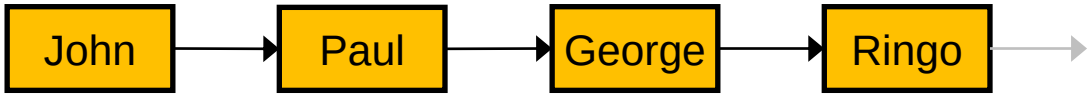
```
void inc (int *p)
{
   ++(*p);
}

int main ()
{
   int x, *p;
   x = 2;
   p = &x;
   inc(p);
   std::cout << x << std::endl;

   return 0;
}
```



Listes chainées

Une **liste chainée** est une **structure de données récursive** permettant de gérer de manière dynamique un ensemble d'éléments de taille variable. Chaque élément d'une liste chainée possède un successeur, sauf un (le dernier).

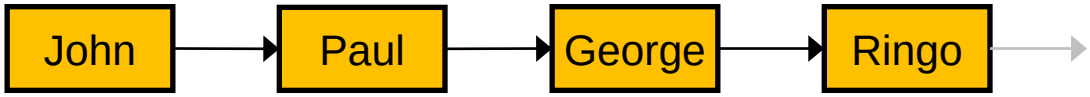
Contrairement aux éléments d'un tableau, les éléments d'une liste chainée ne sont pas indicés.



Listes chainées

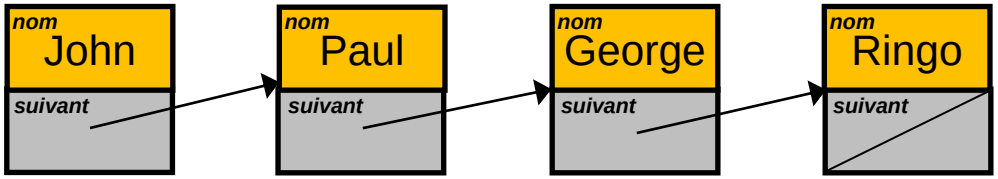
Une **liste chainée** est une **structure de données récursive** permettant de gérer de manière dynamique un ensemble d'éléments de taille variable. Chaque élément d'une liste chainée possède un successeur, sauf un (le dernier).

Contrairement aux éléments d'un tableau, les éléments d'une liste chainée ne sont pas indicés.



Chaque élément d'une liste chainée est constitué de deux parties : son contenu proprement dit, et un pointeur vers l'élément suivant.

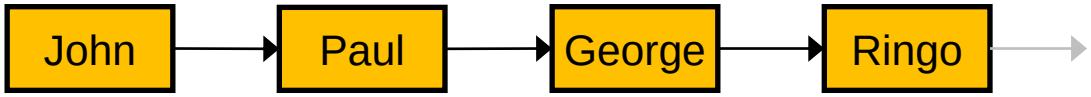
Ainsi chaque élément de la liste pointe sur le premier élément d'une liste plus petite (un élément en moins).



Listes chainées

Une **liste chainée** est une **structure de données récursive** permettant de gérer de manière dynamique un ensemble d'éléments de taille variable. Chaque élément d'une liste chainée possède un successeur, sauf un (le dernier).

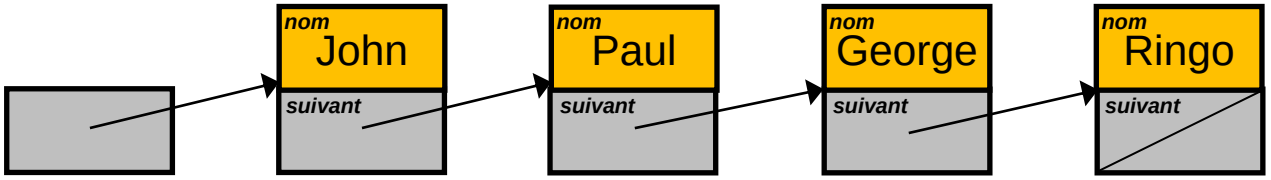
Contrairement aux éléments d'un tableau, les éléments d'une liste chainée ne sont pas indicés.



Chaque élément d'une liste chainée est constitué de deux parties : son contenu proprement dit, et un pointeur vers l'élément suivant.

Ainsi chaque élément de la liste pointe sur le premier élément d'une liste plus petite (un élément en moins).

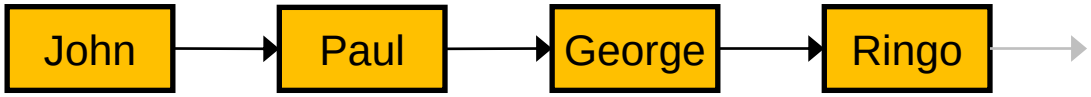
Une variable de type liste est donc un pointeur vers un élément, le premier de la liste chainée.



Listes chainées

Une **liste chainée** est une **structure de données récursive** permettant de gérer de manière dynamique un ensemble d'éléments de taille variable. Chaque élément d'une liste chainée possède un successeur, sauf un (le dernier).

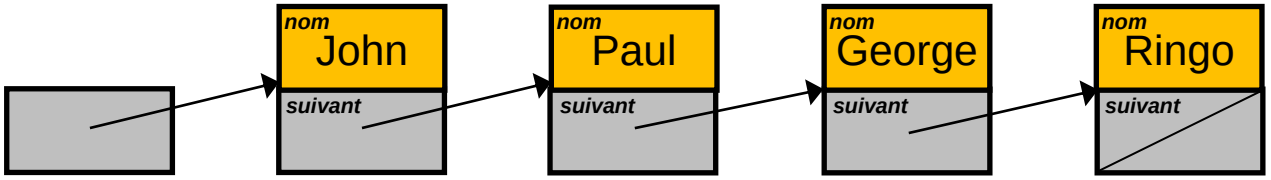
Contrairement aux éléments d'un tableau, les éléments d'une liste chainée ne sont pas indicés.



Chaque élément d'une liste chainée est constitué de deux parties : son contenu proprement dit, et un pointeur vers l'élément suivant.

Ainsi chaque élément de la liste pointe sur le premier élément d'une liste plus petite (un élément en moins).

Une variable de type liste est donc un pointeur vers un élément, le premier de la liste chainée.

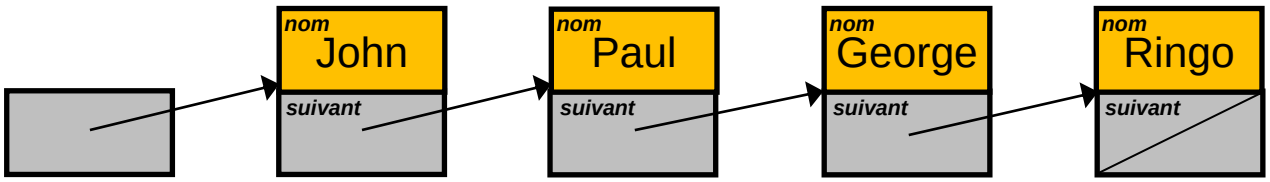


L'utilisation de listes passe par la déclaration du type et la définition de **primitives** de base (par exemple : initialiser, est_vide, ajouter, enlever, vider, longueur). Les traitements algorithmiques utilisant les listes peuvent ensuite s'opérer sans manipulation explicite de pointeurs ou de variables dynamiques.

Listes chainées

Déclaration :

```
struct element
{
        std::string nom;
        element *suivant;
};
using liste = element *;
```



Listes chainées

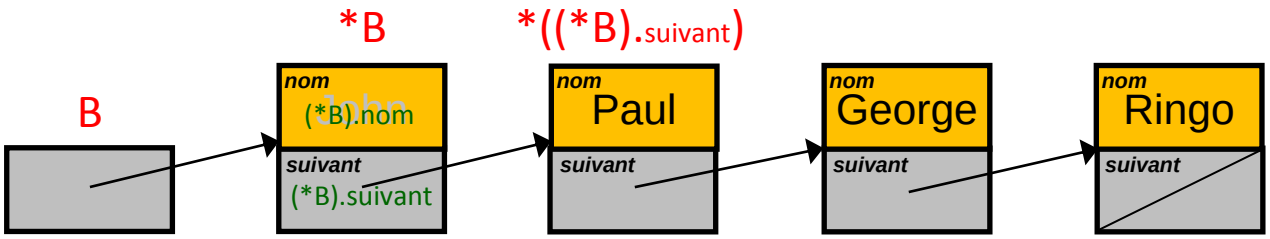
Déclaration :

```
struct element
{
        std::string nom;
        element *suivant;
};
using liste = element *;
```

Le type d'élément composant la liste est libre
(types simples, tableaux, enregistrements, listes, …)

Pour accéder à un élément, il faut parcourir la chaîne jusqu'à cet élément.

- Pour accéder directement aux éléments il faudrait utiliser autant de pointeurs externes que d'éléments.
- On privilégie ici l'espace mémoire utilisé par rapport au facteur temps.



Listes chainées

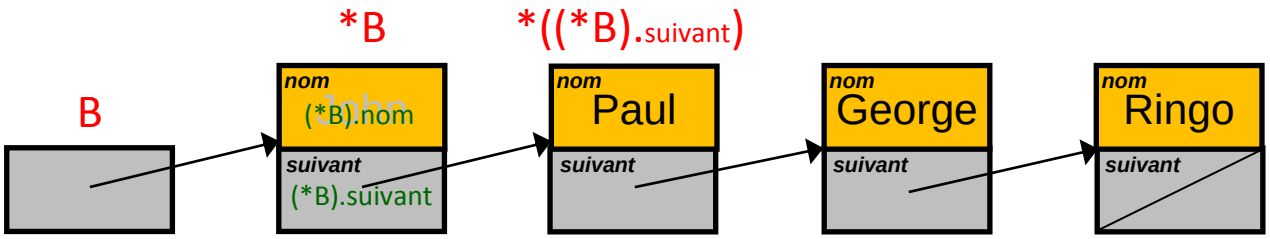
Déclaration :

```
struct element
{
        std::string nom;
        element *suivant;
};
using liste = element *;
```

Le type d'élément composant la liste est libre
(types simples, tableaux, enregistrements, listes, …)

Pour accéder à un élément, il faut parcourir la chaîne jusqu'à cet élément.

- Pour accéder directement aux éléments il faudrait utiliser autant de pointeurs externes que d'éléments.
- On privilégie ici l'espace mémoire utilisé par rapport au facteur temps.



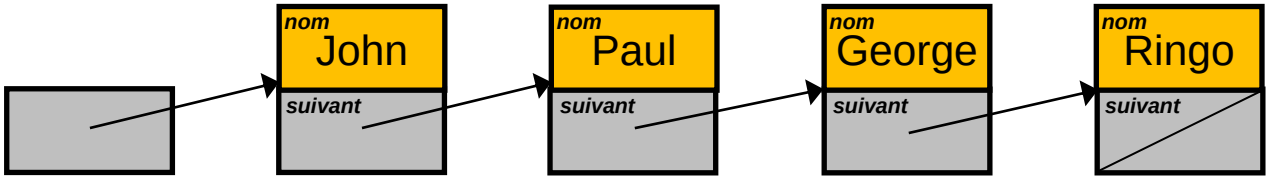
Notation. On pourra écrire :

- B->nom au lieu de (*B).nom
- B->suivant au lieu de (*B).suivant
- B->suivant->nom au lieu de (*((*B).suivant)).nom
- *etc.*

Listes chainées : affichage

Exemple : affichage de liste

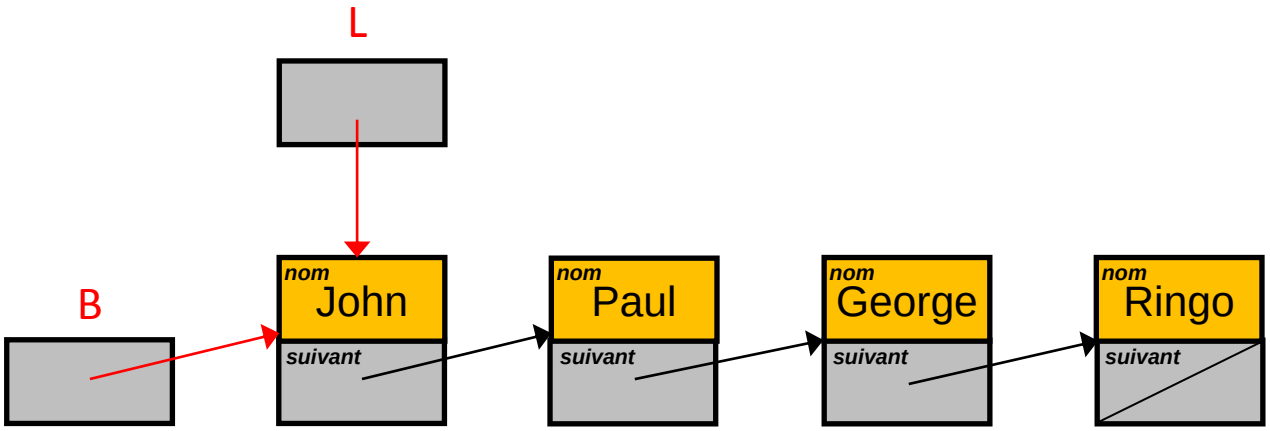
```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```



Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

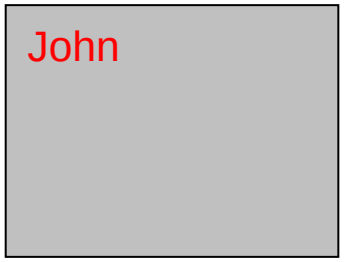


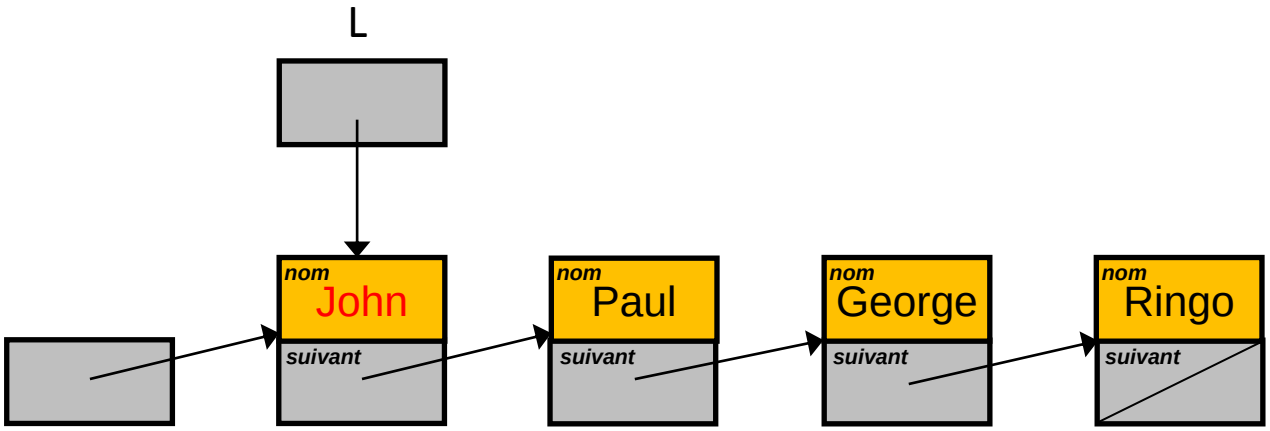

```
affiche(B);
```

Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```



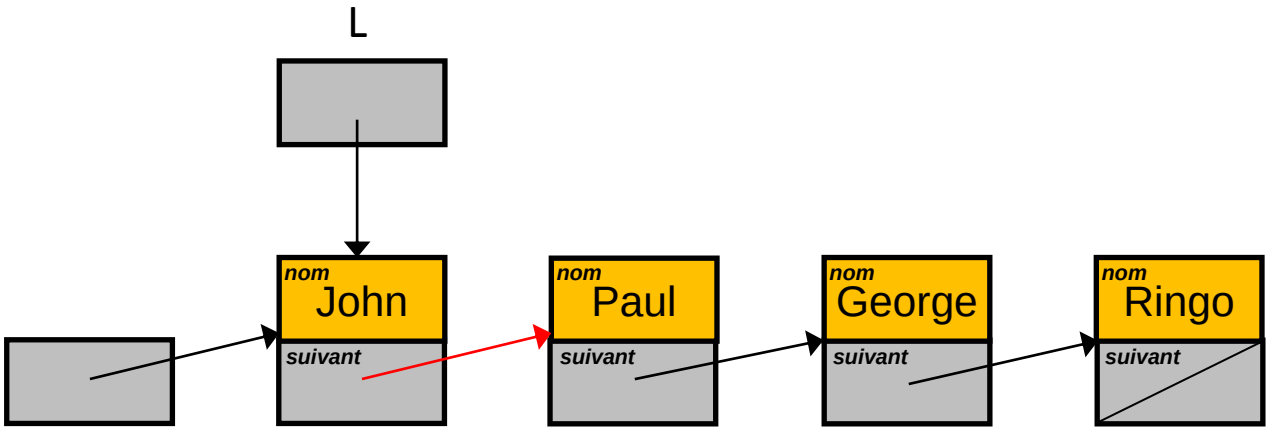


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

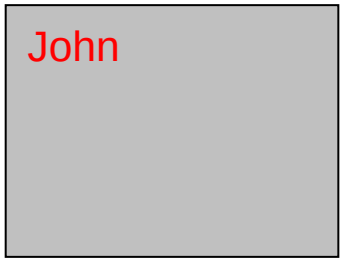
John

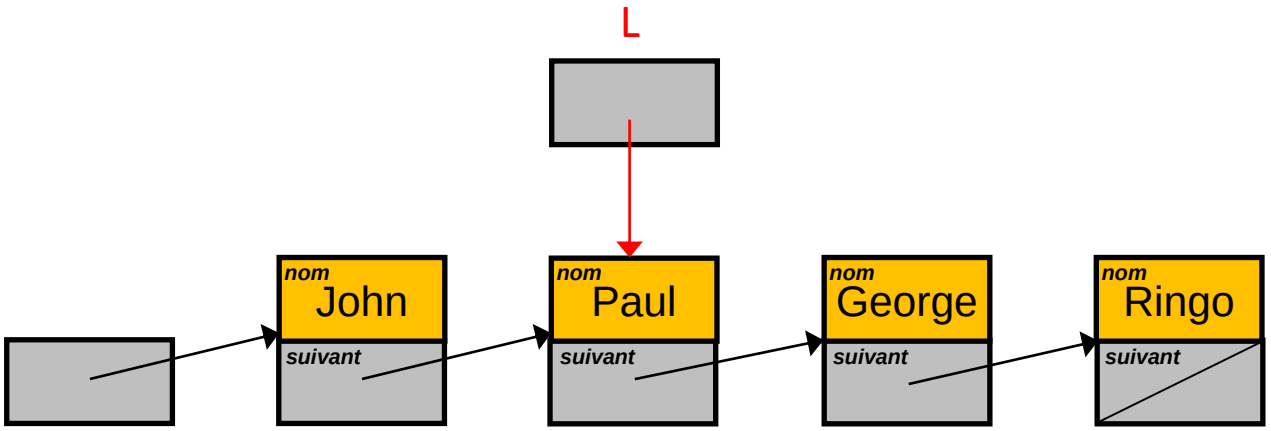


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```



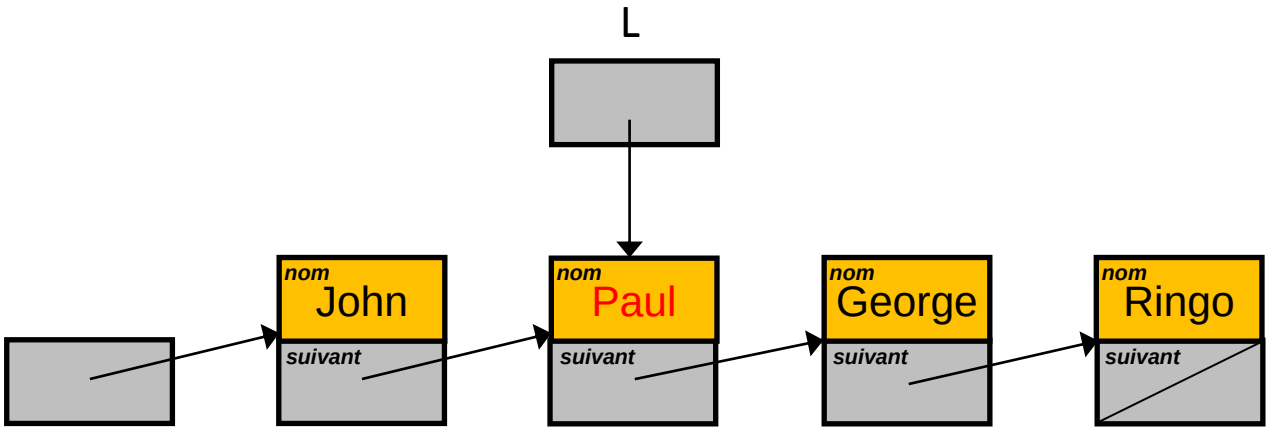


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

John
Paul

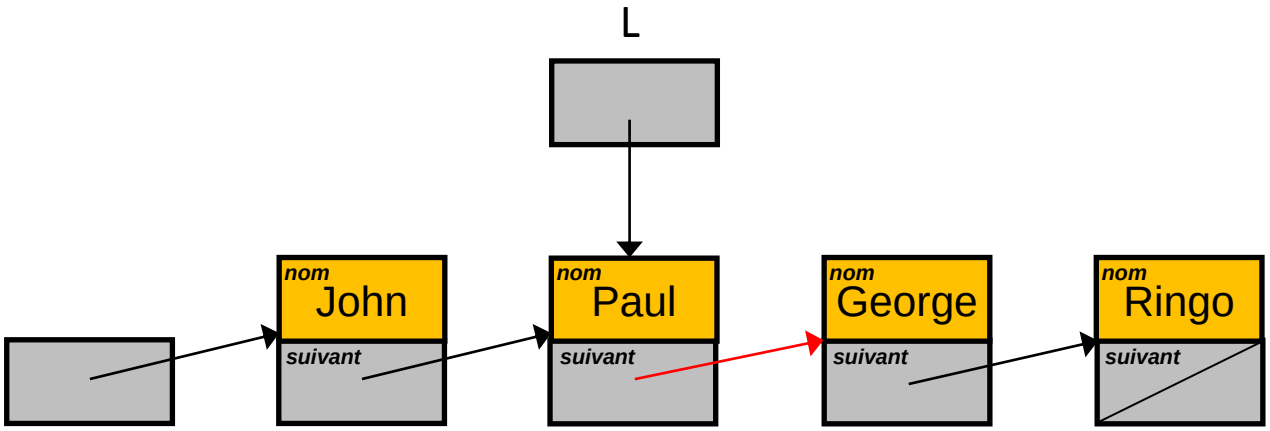


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

John
Paul

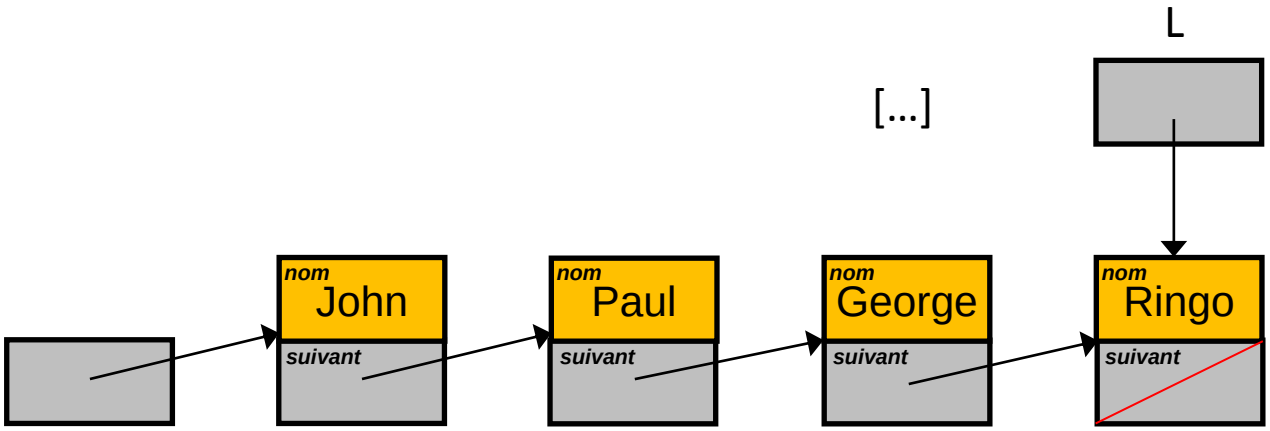


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

```
John
Paul
George
Ringo
```

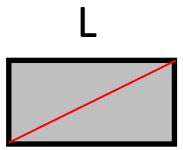


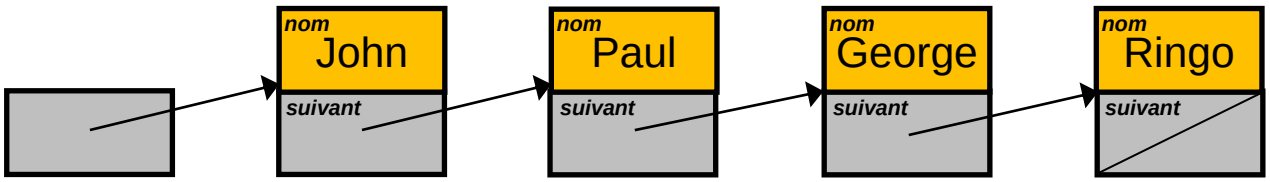
Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

John
Paul
George
Ringo



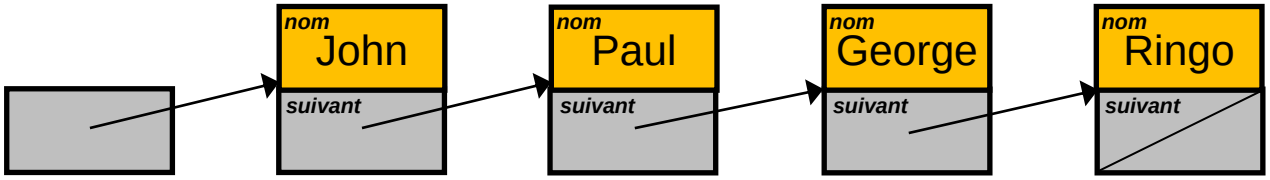


Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << (*L).nom << std::endl;
      affiche ((*L).suiv);
   }
}
```

John
Paul
George
Ringo



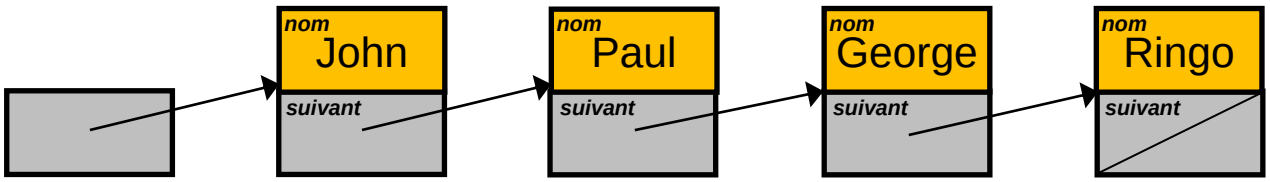
Listes chainées : affichage

Exemple : affichage de liste

```
void affiche(liste L)
{
   if (L != NULL)
   {
      std::cout << L->nom << std::endl;
      affiche (L->suiv);
   }
}
```

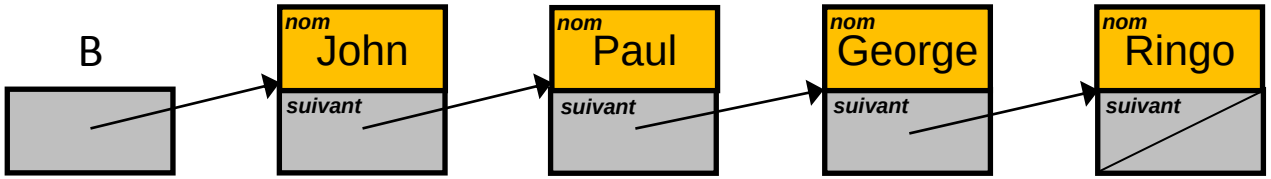
Écriture simplifiée

```
John
Paul
George
Ringo
```



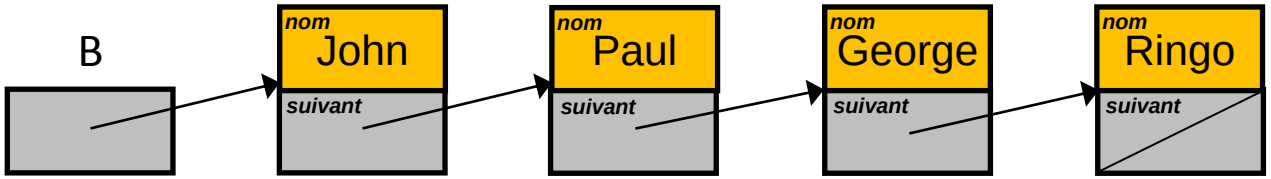
Listes chainées : ajout d'élément

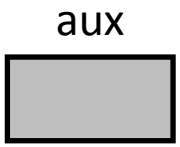
Ajouter **Andy** après **George**



Listes chainées : ajout d'élément

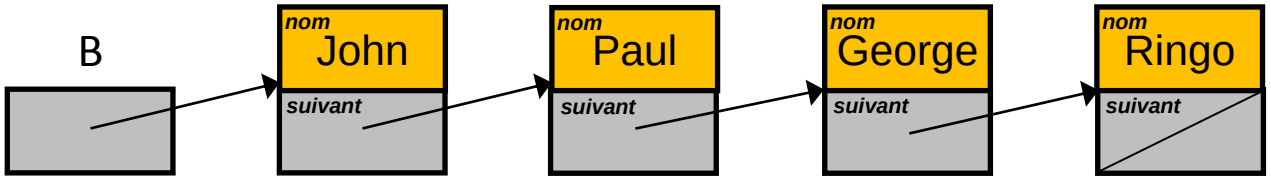
Ajouter **Andy** après **George**

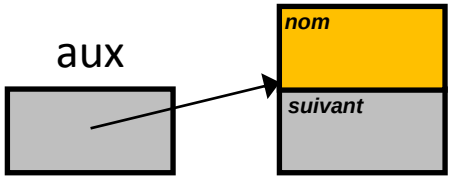




Listes chainées : ajout d'élément

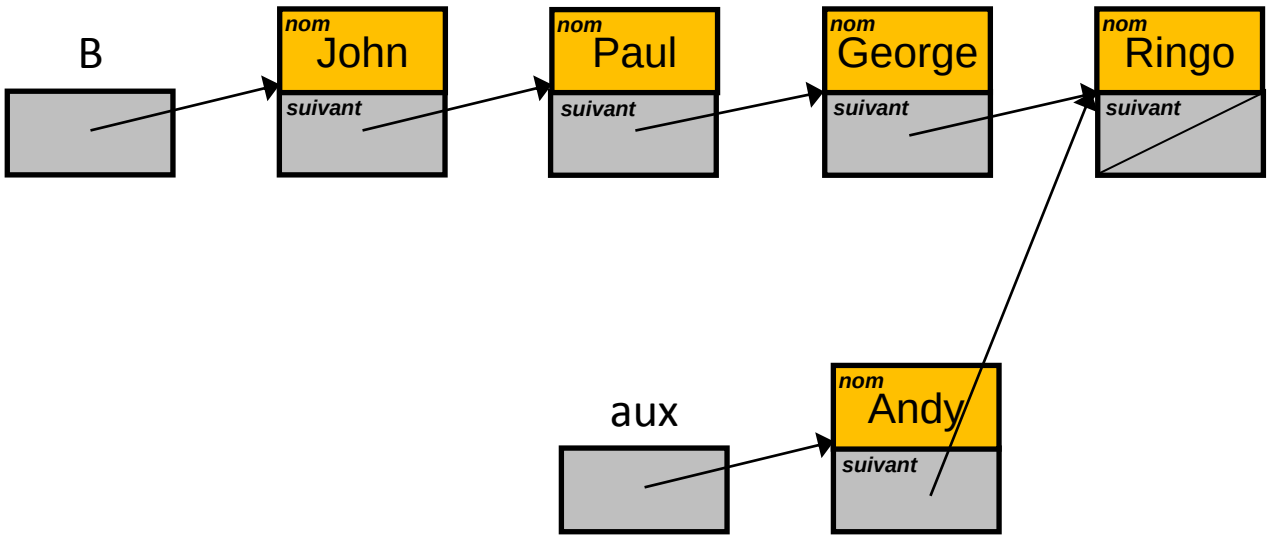
Ajouter **Andy** après **George**





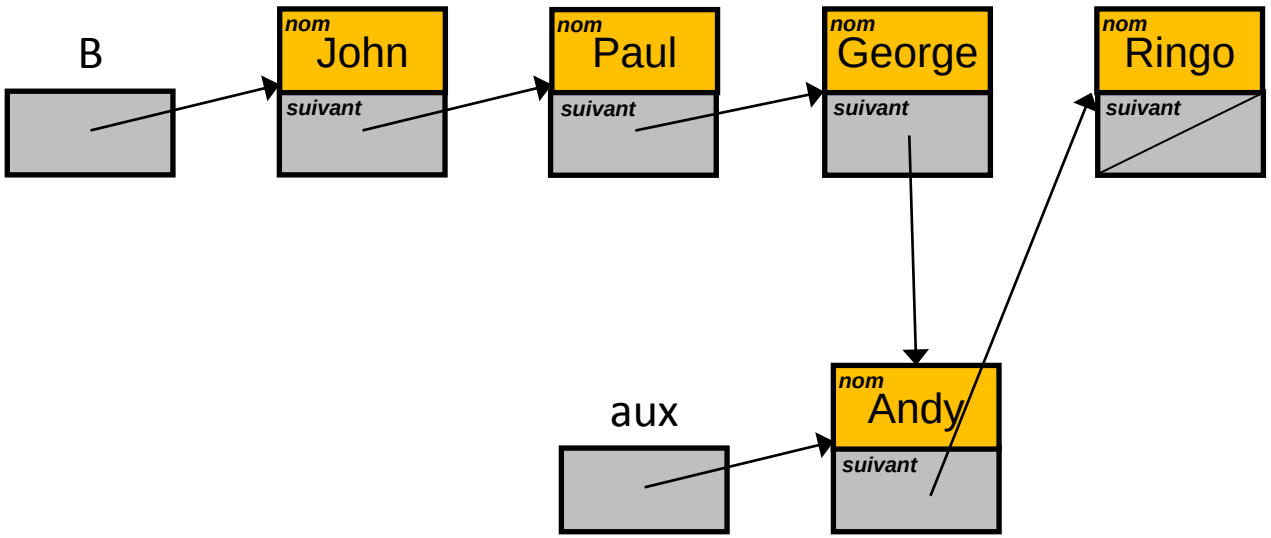
Listes chainées : ajout d'élément

Ajouter **Andy** après **George**



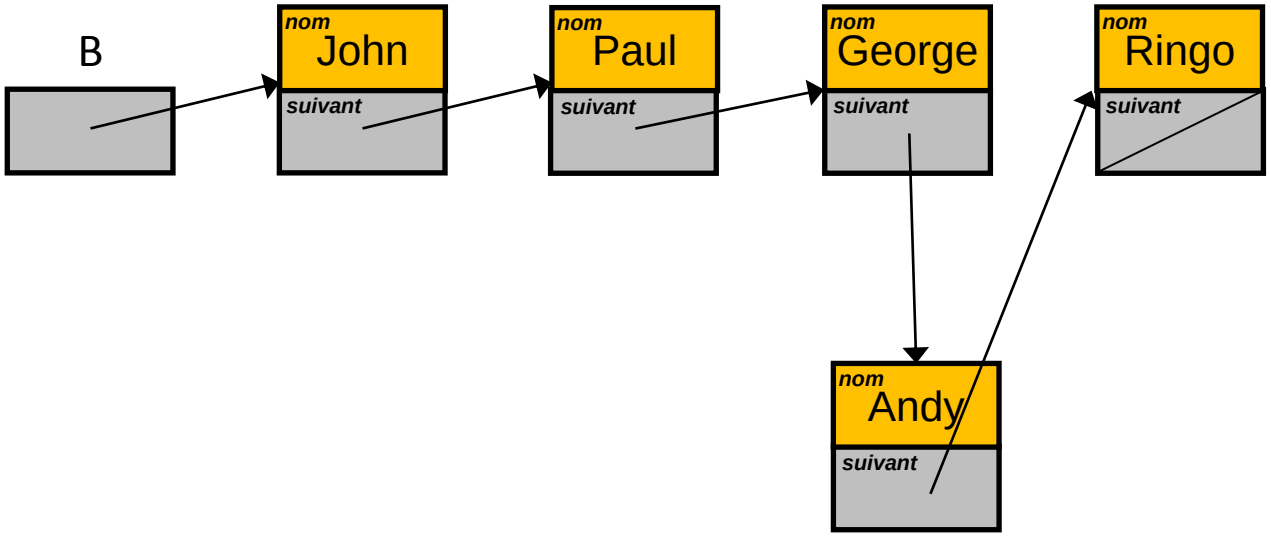
Listes chainées : ajout d'élément

Ajouter **Andy** après **George**



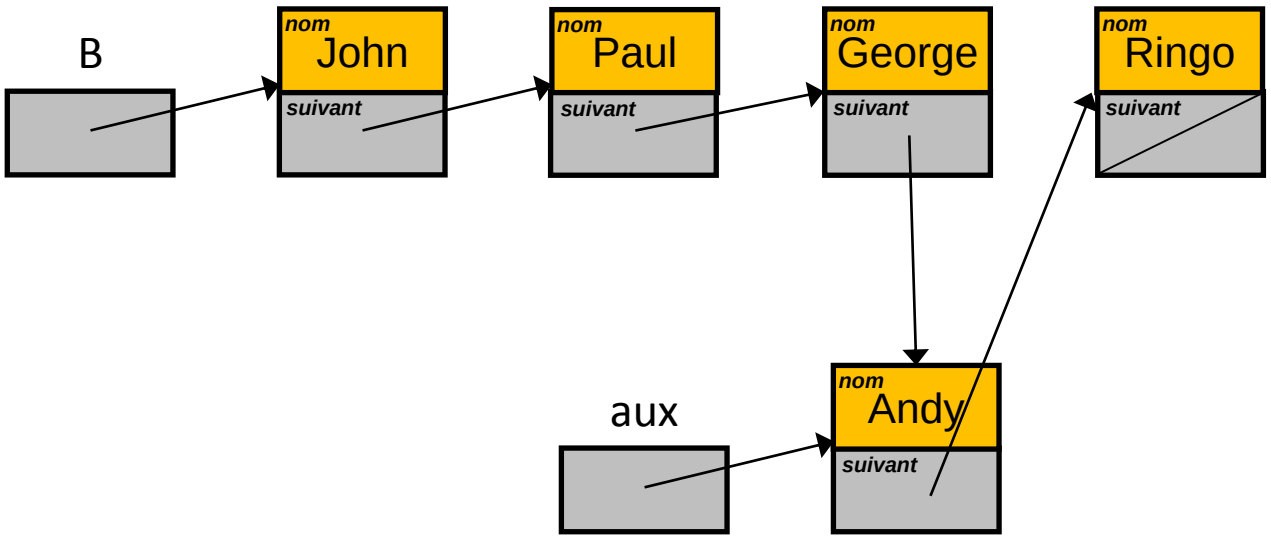
Listes chainées : suppression d'élément

Enlever **Andy**



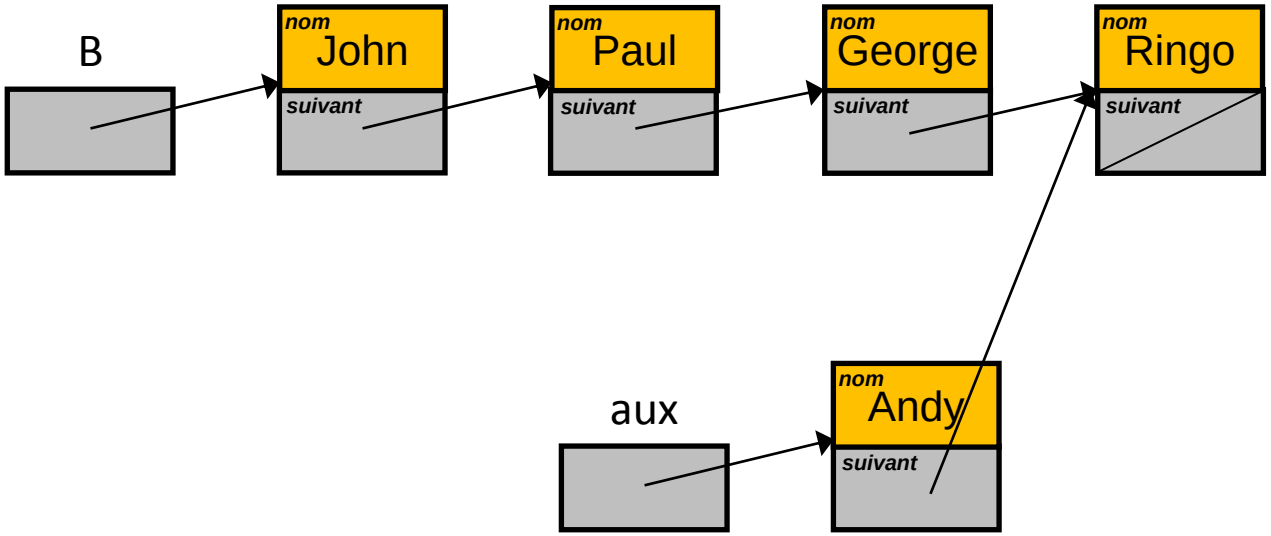
Listes chainées : suppression d'élément

Enlever **Andy**



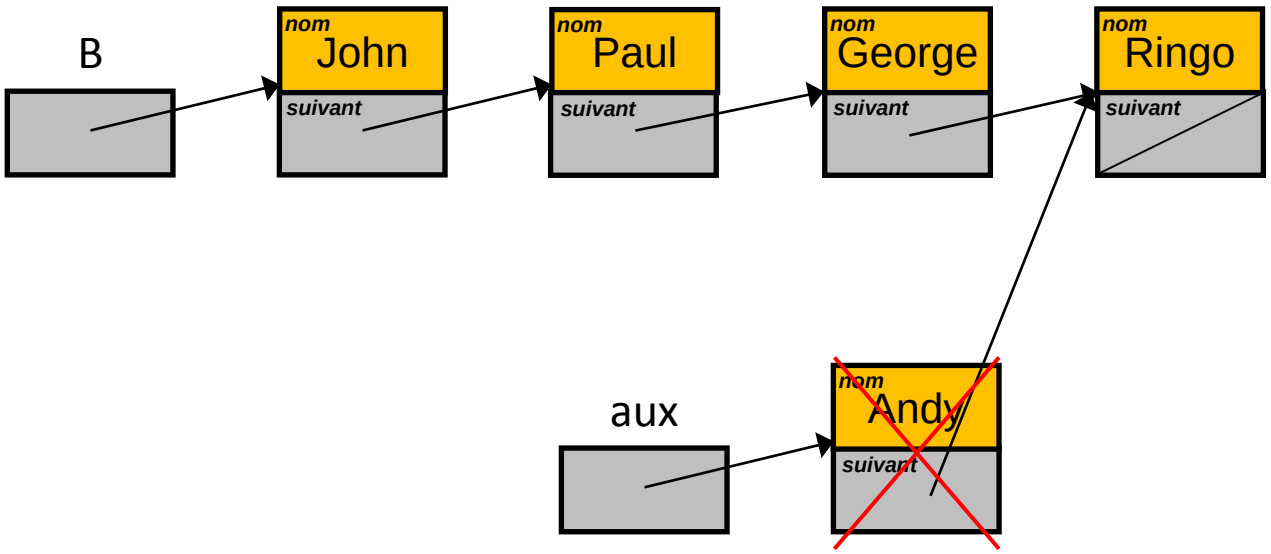
Listes chainées : suppression d'élément

Enlever **Andy**



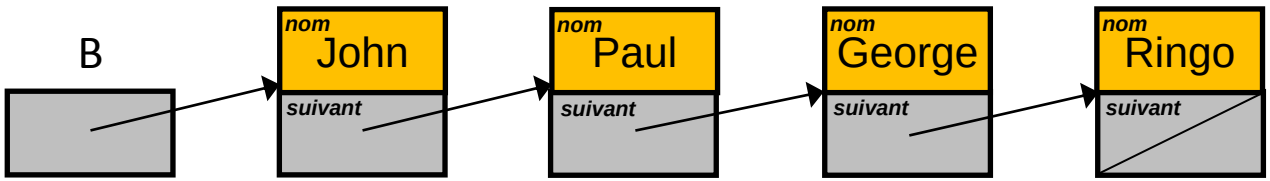
Listes chainées : suppression d'élément

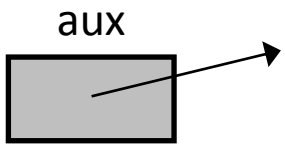
Enlever **Andy**



Listes chainées : suppression d'élément

Enlever **Andy**





Structures de données récursives usuelles

- Liste chainée
- Pile : liste où le dernier élément entré est le premier à pouvoir sortir (**LIFO**). Les éléments sont **empilés** (ajoutés) ou **désempilés** (enlevés et temporairement mémorisés).
- File : liste où le premier élément entré est le premier à pouvoir sortir (**FIFO**). Cela nécessite, à la fois un accès direct au premier élément (**tête**) pour les suppressions, et au dernier élément (**queue**) pour les ajouts.
- Liste doublement chainée : liste où chaque élément comporte un lien (pointeur) vers l'élément suivant et un lien vers l'élément précédent.
- Liste circulaire : liste où le premier élément est le successeur du dernier.
- Arbre binaire : sorte de liste où chaque élément (appelé **nœud**) possède deux successeurs (**fils** gauche et fils droit).
- Arbre binaire de recherche, Tas : arbres binaires particuliers où les valeurs des nœuds sont ordonnées selon des propriétés particulières.
- Arbre n-aire : arbre où chaque nœud possède jusqu'à n fils.