

# *Programmation orientée objet*

## Listes liées

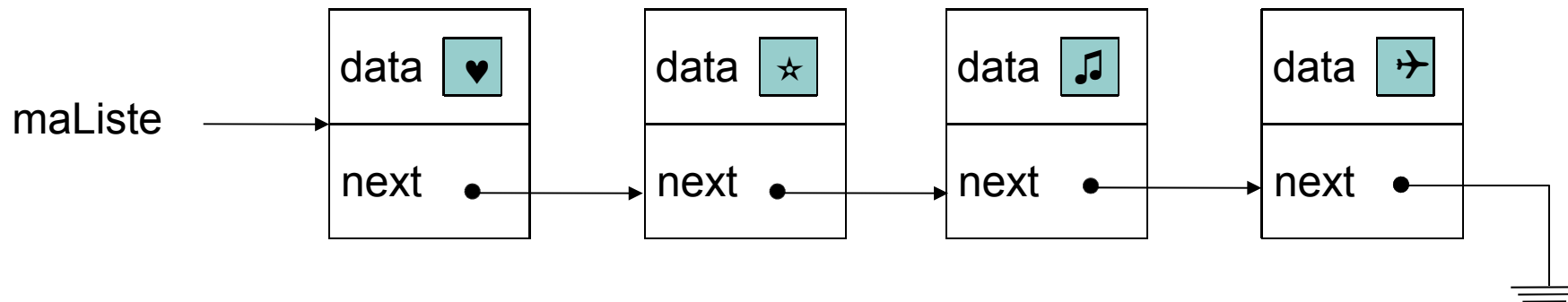
# Motivation

---

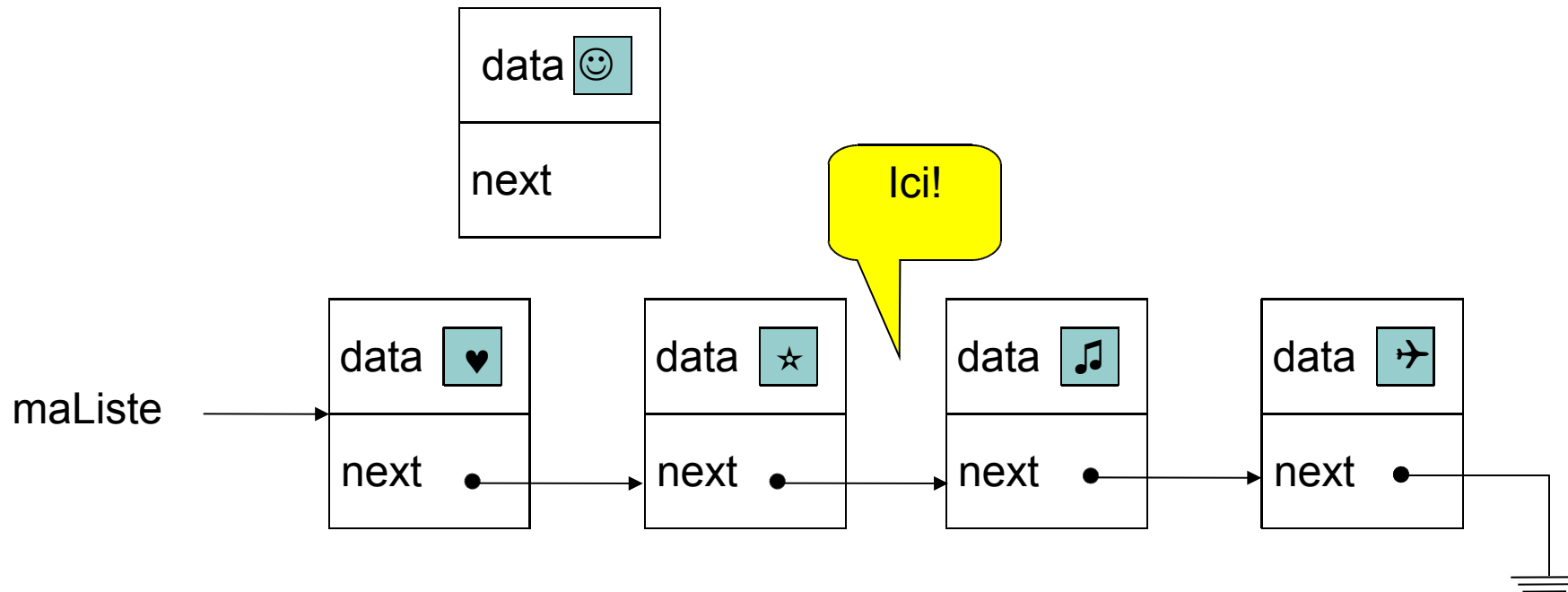
- Imaginez un tableau contenant un certain nombre d'éléments
- On veut maintenant ajouter un élément au milieu (ou, pire encore, au début) du tableau
- Il faudra décaler tous les éléments à partir de cette position pour insérer le nouvel élément
- Il s'agit d'une procédure coûteuse
- Solution: liste liée

# Liste liée simple

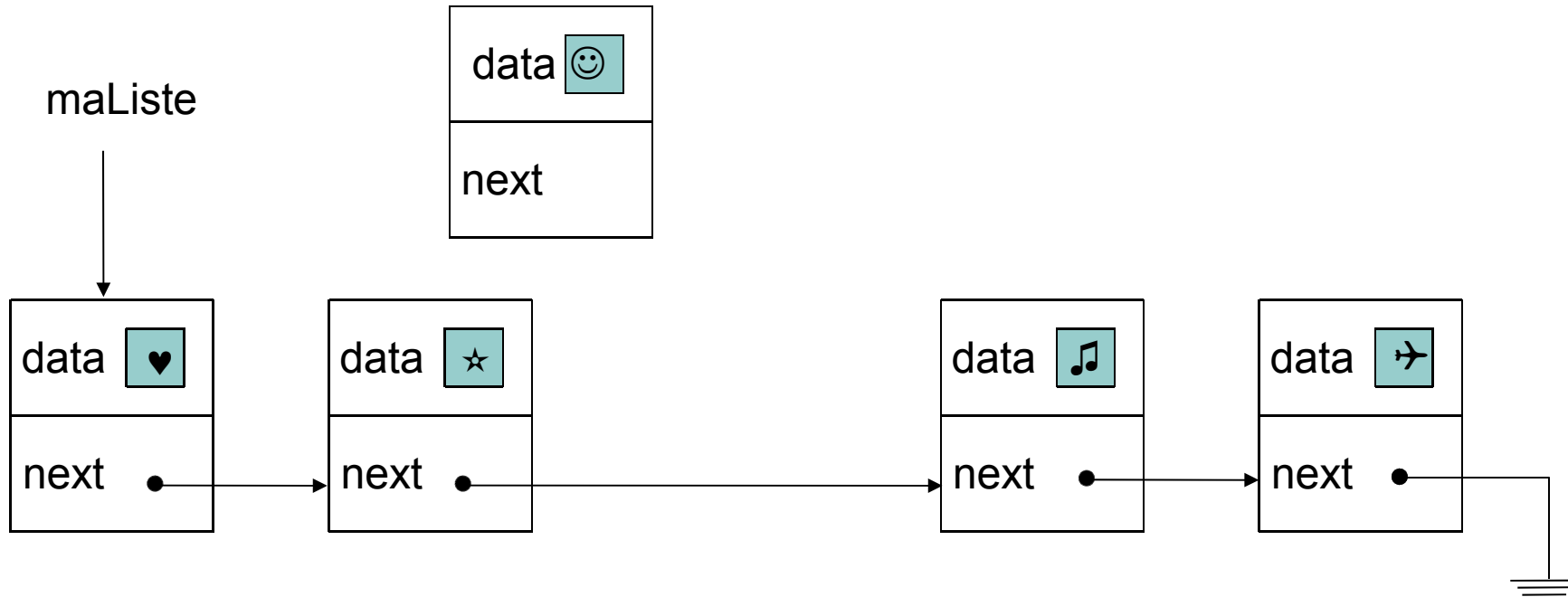
---



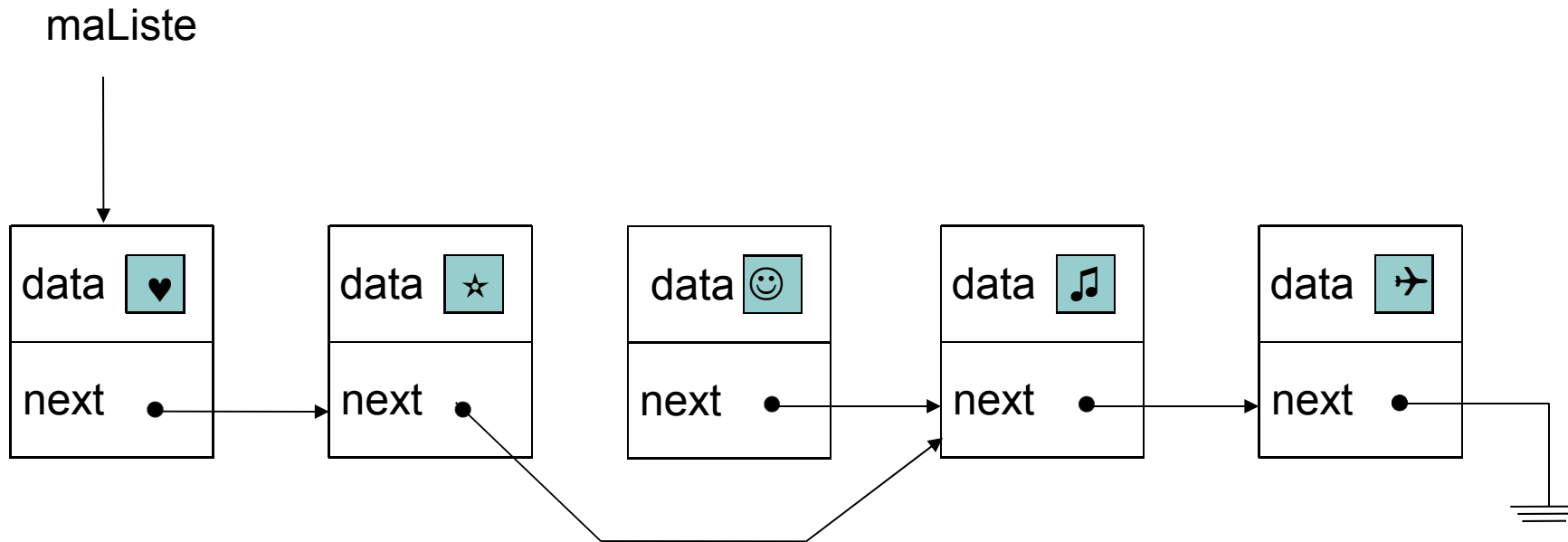
# Ajout d'un élément



# Ajout d'un élément (suite)

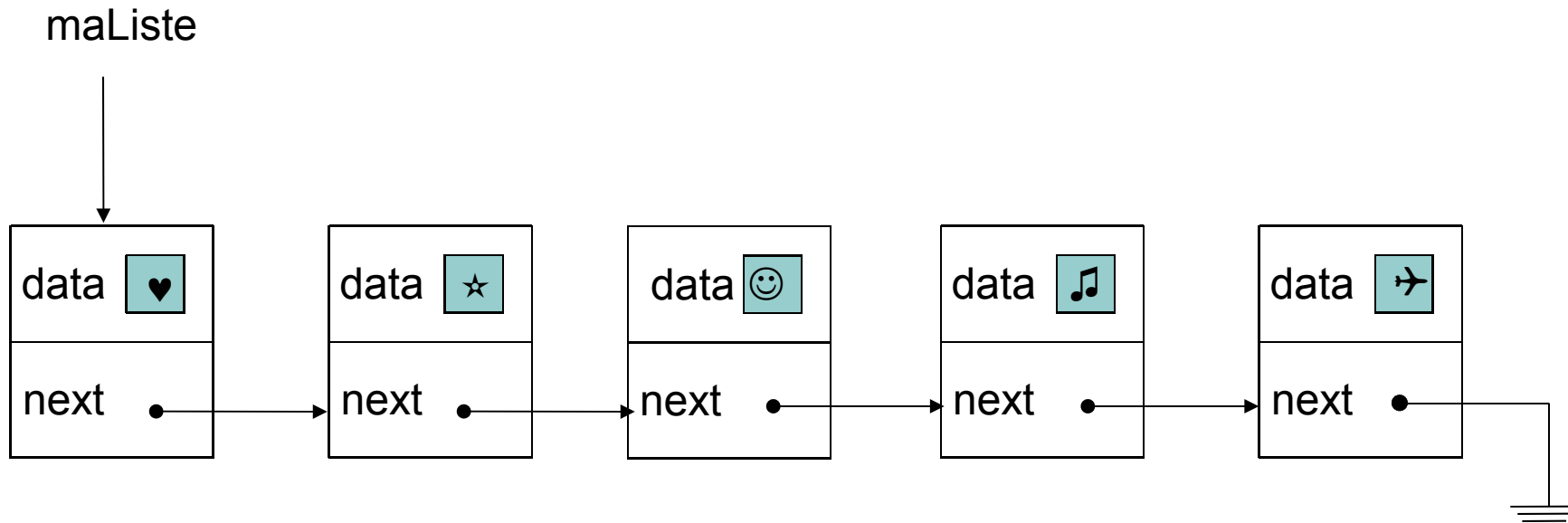


# Ajout d'un élément (suite)



# Ajout d'un élément (suite)

---



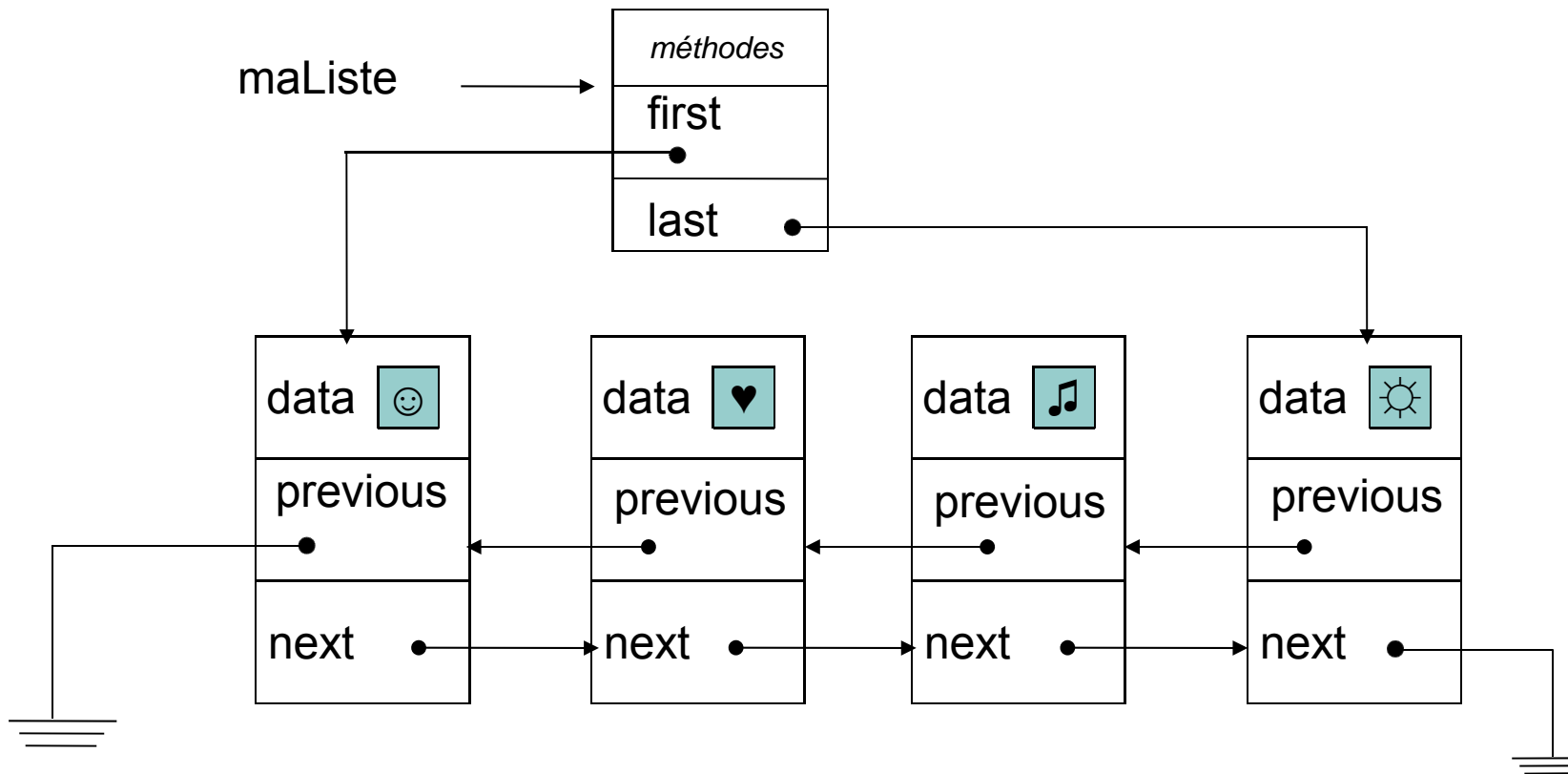
# Ajout d'un élément

---

- On voit donc que l'ajout d'un élément est peu coûteux
- Il suffit de deux affectations de pointeurs
- Il y a un prix à payer par contre: pour trouver l'endroit d'insertion il faut parcourir la liste à partir du début



# Liste à liens doubles




# Implémentation des listes

---

- Pour implémenter les listes, il faut trois classes:
  - La classe **Node**, pour représenter un élément de la liste
  - La classe **List**, pour représenter la liste
  - La classe **Iterator**, pour implémenter le parcours de la liste

# Classe Node - interface

---

```
class Node
{
public:
    Node(string s);
private:
    string data_;
    Node* previous_;
    Node* next_;
friend class List; 
friend class Iterator;
};
```

La déclaration *friend* permet à tout objet de classe **List** ou **Iterator** de manipuler directement les attributs privés d'un objet de la classe **Node**.

## Remarques sur *friend*

---

- Cela brise le principe d'encapsulation
- Il faut donc éviter de l'utiliser
- Dans ce cas-ci, il s'impose parce que les classes **List** et **Iterator** sont intimement liées à la classe **Node**
- **Node** n'est en fait qu'une implémentation des données qui restera cachée à l'utilisateur de la liste (il n'a aucune méthode publique à part le constructeur)
- On aurait pu implémenter les listes sans utiliser *friend* en implémentant des méthodes d'accès (voir exemple sur le site du cours)

# Itérateur

---

- Un itérateur est une sorte de «pointeur»
- Il «pointe» vers un élément de la liste
- On peut lui demander de nous retourner la valeur de l'élément pointé
- On peut lui demander de se déplacer à l'item suivant ou précédent
- Finalement, on peut vérifier si deux itérateurs pointent vers le même item

# Classe Iterator - interface

---

```
class Iterator
{
public:
    Iterator();
    string get() const;
    void next();
    void previous();
    bool equals(Iterator b) const;
private:
    Node* position_;
    Node* last_;
friend class List;
};
```

Si on appelle **next()** alors qu'on est à la fin de la liste, **position\_** aura alors la valeur 0. Si immédiatement après on appelle **previous()**, on utilisera l'attribut **last\_** pour revenir à la liste.

# Classe Iterator - implémentation

```
void Iterator::next()
{
    assert(position != 0);
    position_ = position->next_;
}

void Iterator::previous()
{
    if (position_ == 0)
        position_ = last_;
    else
        position_ = position->previous_;
    assert(position_ != 0);
}
```

Remarquez qu'on accède directement à l'attribut privé d'un objet de la classe **Node**.

Si la position est 0 cela signifie qu'on s'est retrouvé après le dernier item. Il faut donc se repositionner au dernier item de la liste.

Sinon, on se positionne à l'item précédent.

# Classe Iterator – implémentation (suite)

---

```
string Iterator::get() const
{
    assert(position_ != 0);
    return position_>data_;
}
```

Ne serait-il pas mieux de faire un passage par référence?

```
bool Iterator::equals(Iterator b) const
{
    return position_ == b.position_;
}
```

Pourquoi ce const?



# Classe List - interface

---

```
class List
{
public:
    List();
    void push_back(string s) // pour insérer à la fin
    void insert(Iterator pos, string s)
    Iterator erase(Iterator pos);
    Iterator begin(); // retourne un itér. qui pointe
                     // sur le premier item
    Iterator end();  // retourne un itér. qui pointe
                     // apres le dernier item

private:
    Node* first_;
    Node* last_;
}
```

# Parcours d'une liste

---

- On demande d'abord à la liste de nous retourner un itérateur qui pointe à son premier item
- On demande aussi un itérateur qui pointe *après* le dernier item
- Soient **pos** et **fin** ces deux itérateurs, respectivement

## Parcours d'une liste (suite)

---

- Tant que **pos** est différent de **fin** (pour vérifier cela on appelle la méthode **equals()**):
  - Vérifier si l'item pointé par **pos** est celui qu'on recherche
  - Si c'est le cas, on arrête
  - Sinon on fait pointer **pos** à l'item suivant

# Exemple de parcours de liste

---

```
int main()
{
    List personnel;
    personnel.push_back("Michel");
    personnel.push_back("Roberta");
    personnel.push_back("Claudia");
    personnel.push_back("Mohamed");

    Iterator pos = personnel.begin();
    Iterator fin = personnel.end();

    while (!pos.equals(fin) &&
           pos.get() != "Claudia")
        pos.next();
    if (!pos.equals(fin)){
        personnel.erase(pos);
        ...
    }
}
```

# Exemple de parcours de liste

---

```
int main()
{
    List personnel;
    personnel.push_back("Michel");
    personnel.push_back("Roberta");
    personnel.push_back("Claudia");
    personnel.push_back("Mohamed");

    Iterator pos = personnel.begin();
    Iterator fin = personnel.end();

    while (!pos.equals(fin) &&
           pos.get() != "Claudia")
        pos.next();
    if (!pos.equals(fin)){
        personnel.erase(pos);
        ...
    }
}
```

On remplit la liste.

# Exemple de parcours de liste

---

```
int main()
{
    List personnel;
    personnel.push_back("Michel");
    personnel.push_back("Roberta");
    personnel.push_back("Claudia");
    personnel.push_back("Mohamed");

    Iterator pos = personnel.begin();
    Iterator fin = personnel.end();

    while (!pos.equals(fin) &&
           pos.get() != "Claudia")
        pos.next();
    if (!pos.equals(fin)){
        personnel.erase(pos);
        ...
    }
}
```

On initialise un itérateur qui pointe sur le premier item de la liste et un autre qui pointe après le dernier élément de la liste.

# Exemple de parcours de liste

---

```
int main()
{
    List personnel;
    personnel.push_back("Michel");
    personnel.push_back("Roberta");
    personnel.push_back("Claudia");
    personnel.push_back("Mohamed");

    Iterator pos = personnel.begin();
    Iterator fin = personnel.end();

    while (!pos.equals(fin) &&
           pos.get() != "Claudia")
        pos.next();
    if (!pos.equals(fin)){
        personnel.erase(pos);
        ...
    }
}
```

On parcourt la liste jusqu'à ce qu'on trouve l'item recherché.

# Exemple de parcours de liste

---

```
int main()
{
    List personnel;
    personnel.push_back("Michel");
    personnel.push_back("Roberta");
    personnel.push_back("Claudia");
    personnel.push_back("Mohamed");

    Iterator pos = personnel.begin();
    Iterator fin = personnel.end();
    while (!pos.equals(fin) &&
           pos.get() != "Claudia") {
        pos.next();
    };
    if (!pos.equals(fin)) {
        personnel.erase(pos);
        ...
    }
}
```

Si on ne se retrouve pas après la fin de la liste, c'est qu'on a trouvé l'item recherché.



# Classe List - implémentation

---

```
Iterator List::begin()
{
    Iterator iter;
    iter.position_ = first_;
    iter.last_ = last_;
    return iter;
}
```

On crée un itérateur qui pointe au premier élément et on retourne cet itérateur.

```
Iterator List::end()
{
    Iterator iter;
    iter.position_ = 0;
    iter.last_ = last_;
    return iter;
}
```

On crée un itérateur dont la position est 0. Dans notre implémentation, cela signifie qu'on est positionné *après* le dernier élément.

# Classe List – insertion d'un item à la fin

```
void List::push_back(string s)
{
    Node* newnode = new Node(s);
    if (last_ == 0) /* la liste est vide */
    {
        first_ = newnode;
        last_ = newnode;
    }
    else
    {
        newnode->previous_ = last_;
        last_->next_ = newnode;
        last_ = newnode;
    }
}
```

On crée un nouveau noeud.

Si la liste est vide, les deux pointeurs pointeront sur le nouveau noeud.

Sinon on fait pointer le nouveau noeud sur le dernier de la liste.

L' « ancien » dernier noeud doit maintenant pointer sur le nouveau noeud.

La dernier item de la liste est maintenant le « nouveau » dernier noeud.

# Classe List – Insertion à la position indiquée par l'itérateur

---

```
void List::insert(Iterator iter, string s)
{
    if (iter.position_ == 0)
    {
        push_back(s);
        return;
    }

    ...
}
```

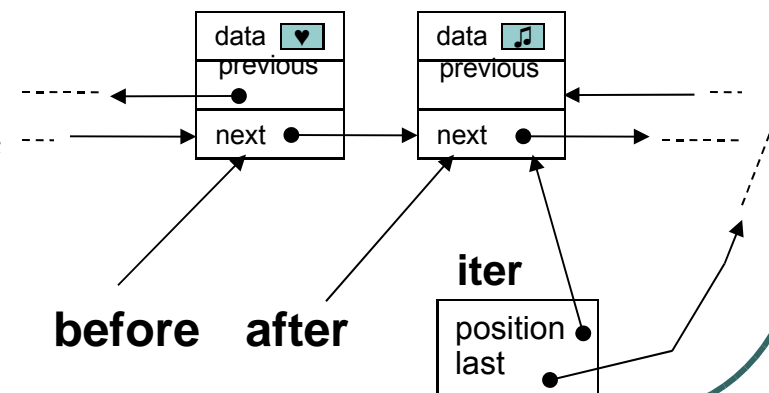
Si la position demandée est après la fin de la liste, on ne se complique pas la vie et on appelle la méthode `push_back()`.

# Insertion à la position indiquée par l'itérateur (suite)

...

```
Node* after = iter.position_;  
Node* before = after->previous_;  
Node* newnode = new Node(s);  
newnode->previous_ = before;  
newnode->next_ = after;  
after->previous_ = newnode;  
if (before == 0)  
    first_ = newnode;  
else  
    before->next_ = newnode;  
}
```

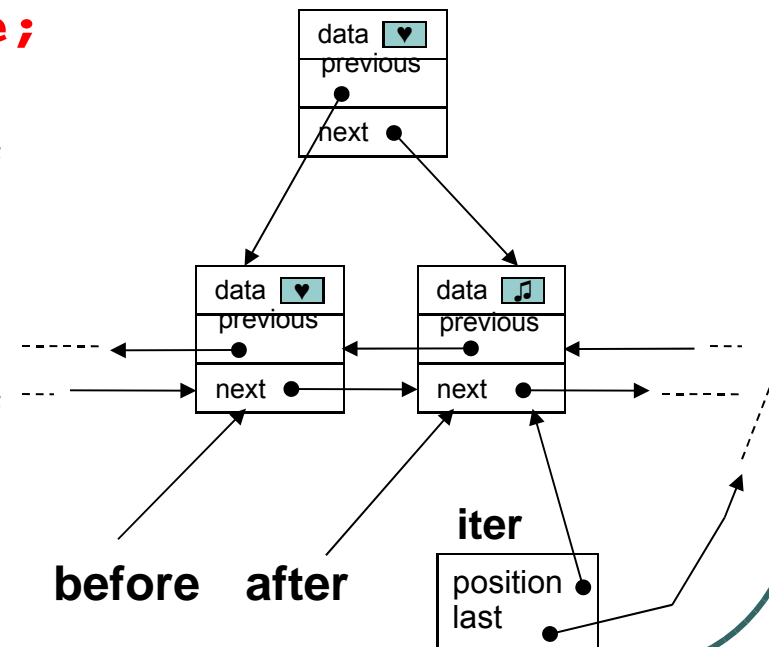
On crée deux pointeurs pour désigner le noeud qui se trouve à la position d'insertion et celui qui se trouve avant.



# Insertion à la position indiquée par l'itérateur (suite)

```
...  
Node* after = iter.position_  
Node* before = after->previous_  
Node* newnode = new Node(s);  
newnode->previous_ = before;  
newnode->next_ = after;  
after->previous_ = newnode;  
if (before == 0)  
    first_ = newnode;  
else  
    before->next_ = newnode;  
}
```

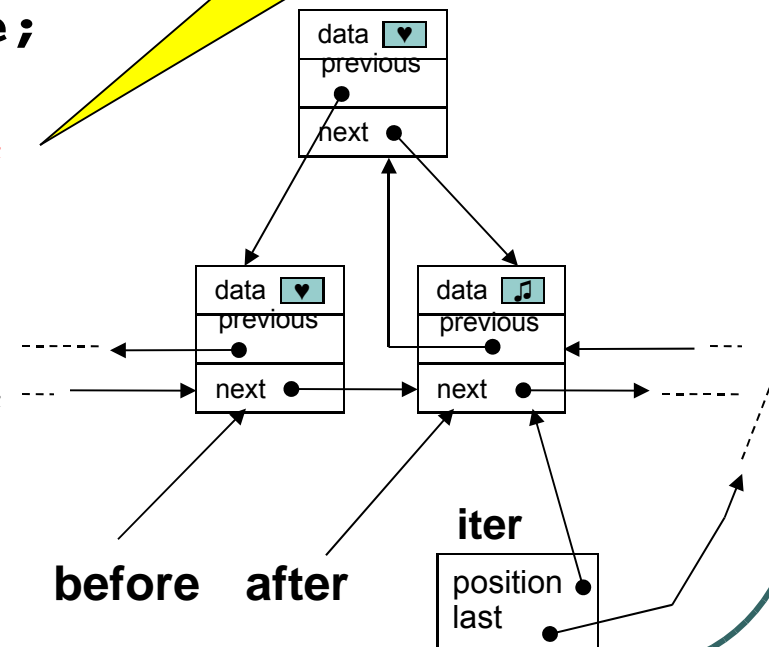
On crée le nouveau noeud et on le fait pointer vers les noeuds qui seront son précédent et son suivant.



# Insertion à la position indiquée par l'itérateur (suite)

```
...  
Node* after = iter.position_  
Node* before = after->previous_  
Node* newnode = new Node(s);  
newnode->previous_ = before;  
newnode->next_ = after;  
after->previous_ = newnode;  
if (before == 0)  
    first_ = newnode;  
else  
    before->next_ = newnode;  
}
```

Le noeud suivant doit maintenant pointer sur le nouveau noeud.

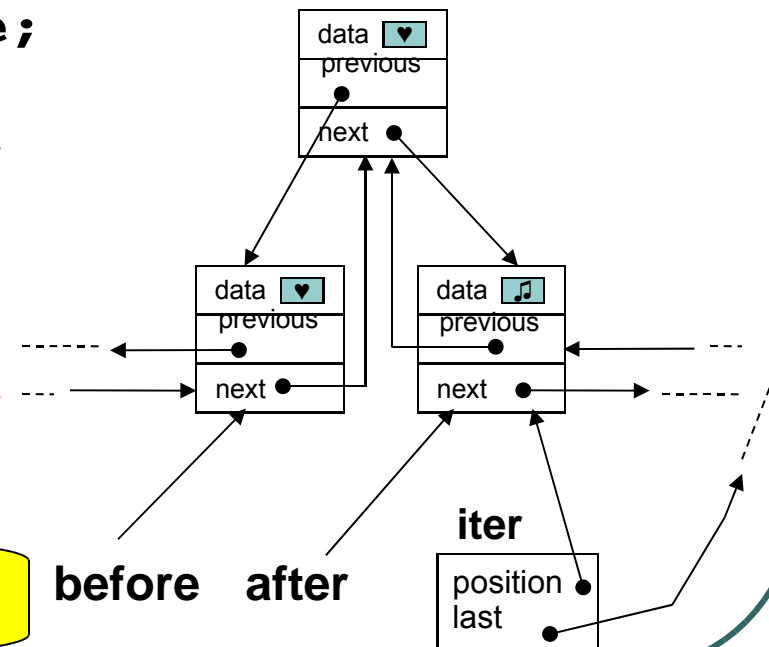


# Insertion à la position indiquée par l'itérateur (suite)

...

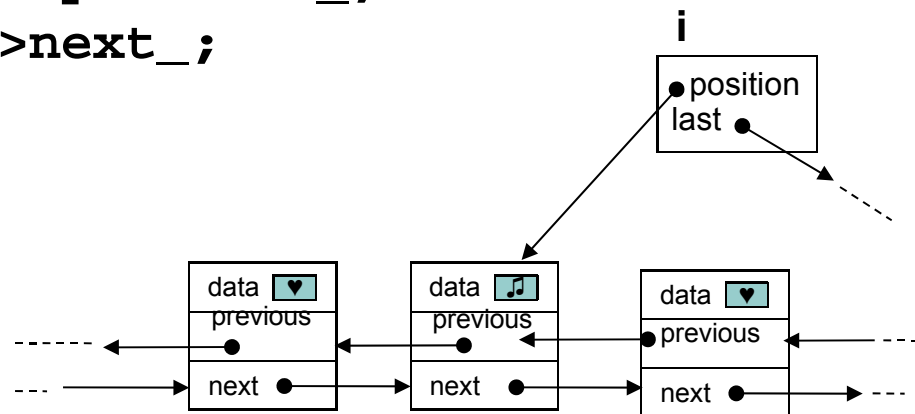
```
Node* after = iter.position_;  
Node* before = after->previous_;  
Node* newnode = new Node(s);  
newnode->previous_ = before;  
newnode->next_ = after;  
after->previous_ = newnode;  
if (before == 0)  
    first_ = newnode;  
else  
    before->next_ = newnode;  
}
```

Le noeud précédent, s'il existe, doit lui aussi pointer sur le nouveau noeud.



# Classe List – Retrait à la position indiquée par l'itérateur

```
Iterator List::erase(Iterator i)
{
    Iterator iter = i;
    assert(iter.position_ != 0);
    Node* remove = iter.position_;
    Node* before = remove->previous_;
    Node* after = remove->next_;
    ...
}
```

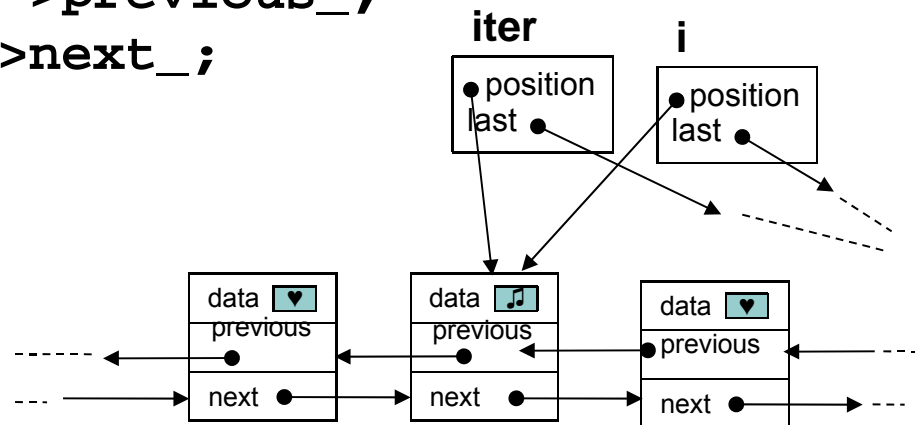




# Classe List – Retrait à la position indiquée par l'itérateur

```
Iterator List::erase(Iterator i)
{
    Iterator iter = i;
    assert(iter.position_ != 0);
    Node* remove = iter.position_;
    Node* before = remove->previous_;
    Node* after = remove->next_;
    ...
}
```

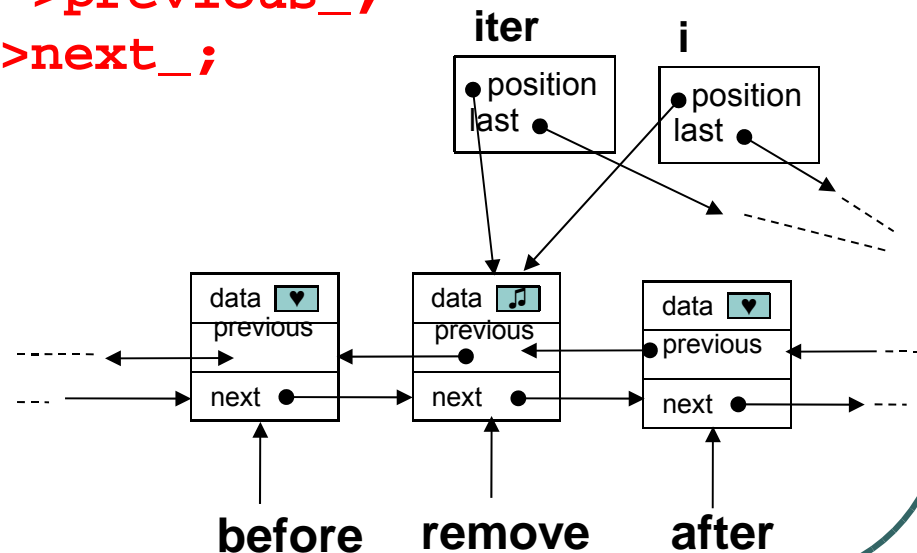
On copie l'itérateur.  
Si on essaie de faire un retrait  
après la fin de la liste, il s'agit  
évidemment d'une erreur.



# Classe List – Retrait à la position indiquée par l'itérateur

```
Iterator List::erase(Iterator i)
{
    Iterator iter = i;
    assert(iter.position_ != 0);
    Node* remove = iter.position_;
    Node* before = remove->previous_;
    Node* after = remove->next_;
    ...
}
```

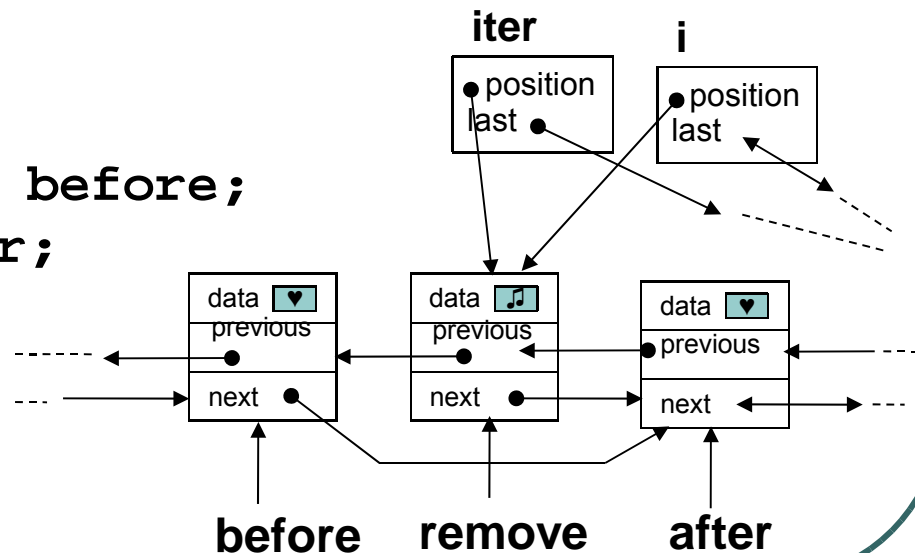
On pointe sur le noeud à effacer ainsi que son suivant et son précédent.



# Classe List – Retrait à la position indiquée par l'itérateur

```
...  
if (remove == first_)  
    first_ = after;  
else  
    before->next_ = after;  
if (remove == last_)  
    last_ = before;  
else  
    after->previous_ = before;  
iter.position_ = after;  
delete remove;  
return iter;  
}
```

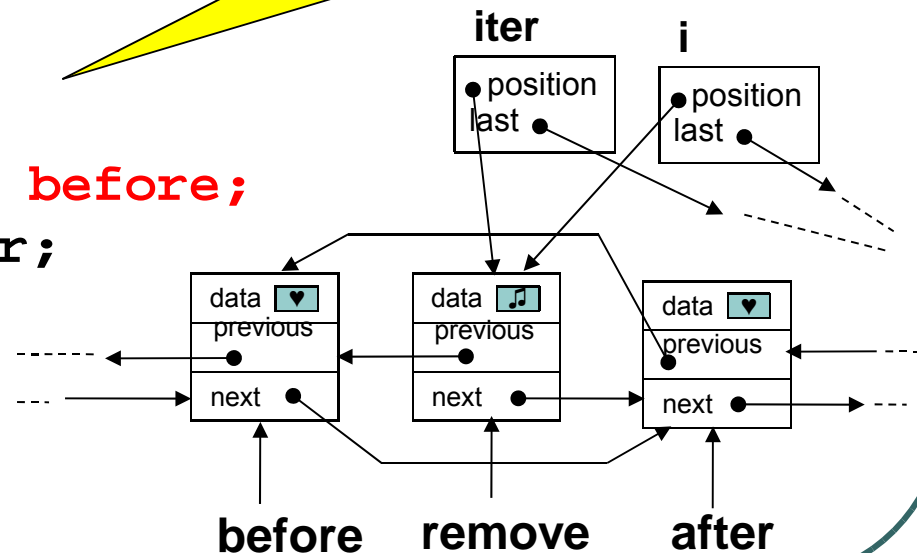
Si l'item retiré est le premier, on fait pointer le pointeur **first\_** de la liste sur l'item suivant. Sinon, on fait pointer le noeud précédent sur le noeud suivant.



# Classe List – Retrait à la position indiquée par l'itérateur

```
...  
if (remove == first_)  
    first_ = after;  
else  
    before->next_ = after;  
if (remove == last_)  
    last_ = before;  
else  
    after->previous_ = before;  
iter.position_ = after;  
delete remove;  
return iter;
```

Si l'item retiré est le dernier, on fait pointer le pointeur **last\_** de la liste sur l'item précédent. Sinon, on fait pointer le noeud suivant sur le noeud précédent.

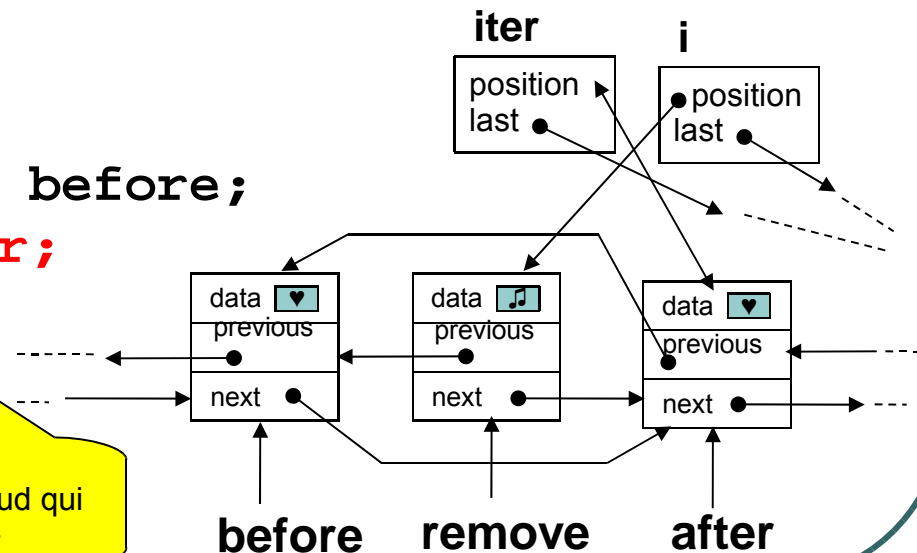


# Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
    first_ = after;
else
    before->next_ = after;
if (remove == last_)
    last_ = before;
else
    after->previous_ = before;
iter.position_ = after;
delete remove;
return iter;
```

```
}
```

Le nouvel itérateur  
pointera sur le noeud qui  
suit le noeud retiré.

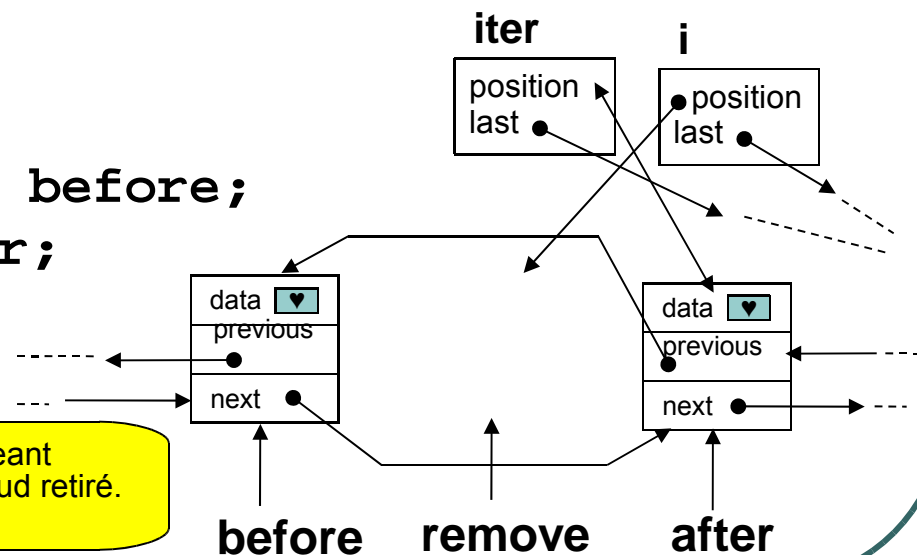


# Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
    first_ = after;
else
    before->next_ = after;
if (remove == last_)
    last_ = before;
else
    after->previous_ = before;
iter.position_ = after;
delete remove;
return iter;
```

```
}
```

On peut maintenant éliminer le noeud retiré.

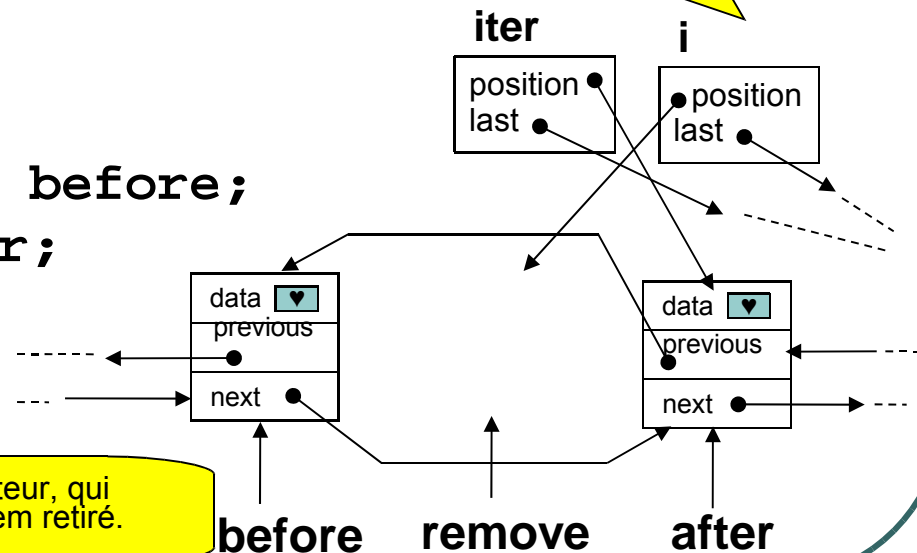


# Classe List – Retrait à la position indiquée par l'itérateur

```
...
if (remove == first_)
    first_ = after;
else
    before->next_ = after;
if (remove == last_)
    last_ = before;
else
    after->previous_ = before;
iter.position_ = after;
delete remove;
return iter;
}
```

Remarquez qu'à la sortie de la fonction, l'itérateur passé en paramètre se retrouve avec un pointeur invalide (n'oubliez pas que les pointeurs ont été copiés lors de l'appel de la fonction).

On retourne le nouvel itérateur, qui pointe sur l'item qui suit l'item retiré.



# Liste liée générique

---

```
template< typename T >
class Node
{
public:
    Node(T s);
private:
    T data;
    Node<T>* previous_;
    Node<T>* next_;
    ...
};
```



# Liste liée générique

---

```
template< typename T >
class Iterator
{
public:
    Iterator();
    T get() const;
    void next();
    bool equals(Iterator<T> iter) const;
private:
    Node<T>* position;
};
```

Remarquez qu'il faut accompagner le nom de la classe du type paramétrisé.

On ne peut plus utiliser la classe Node sans spécifier le type de donnée contenu dans un noeud.

# Liste liée générique

---

```
template< typename T >
class List
{
public:
    List();
    void push_back(T s);
    void insert(Iterator<T> pos, T s);
    Iterator<T> erase(Iterator<T> pos);
    Iterator<T> begin();
    Iterator<T> end();
private:
    Node<T>* first_;
    Node<T>* last_;
};
```

# Listes de la STL

---

- Pas besoin de définir une classe `List`, puisqu'il y en a déjà une fournie dans la bibliothèque STL:
- Il faut faire `#include <list>` pour inclure le fichier d'en-tête
- On utilise alors la class `list`, qui est une classe générique à un paramètre de type

# Listes de la STL – méthodes

---

- Principales méthodes définies pour cette classe:
  - `push_back()`, pour ajouter un item à la fin
  - `push_front()`, pour ajouter un item au début
  - `pop_back()`, pour retirer le dernier item de la liste
  - `pop_front()`, pour retirer le premier item de la liste
  - `front()`, qui retourne le premier élément de la liste
  - `back()`, qui retourne le dernier élément de la liste
  - `size()`, pour connaître le nombre d'items dans la liste
  - `empty()`, pour vérifier si la liste est vide
- Les itérateurs de liste seront vus plus tard lorsque nous présenterons la bibliothèque STL

# Listes de la STL – exemple

---

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    int valeur;
    list<int> liste;
    cin >> valeur;

    // On lit une liste d'entiers
    while (!cin.eof()) {
        liste.push_front(valeur);
        cin >> valeur;
    }
    // On les affiche en ordre inverse
    while (!liste.empty()) {
        cout << liste.front() << endl;
        liste.pop_front();
    }
    return 0;
}
```