IUT2 Université Grenoble Alpes
Département INFO

R3.02
Développement efficace

Cours 1 – listes chaînées & récursivité

Hervé Blanchon
Université Grenoble Alpes
IUT 2 – Département Informatique

Plan du cours

- Liste chaînée
  - type de données abstrait
  - définition inductive
- Illustration
- Algorithmes récursifs de parcours
  - deux familles de parcours
    - parcours de gauche à droite
    - parcours de droite à gauche
- Algorithmes de modification
  - insertion en tête
  - insertion à une position donnée
- Algorithmes itératifs de parcours
  - cas de la récursivité terminale
  - cas de la récursivité non terminale

Où je comprends la définition inductive d’un type de données !

LISTE CHAÎNÉE

Notion de Type Abstrait de Données

- Un TAD est une collection d'éléments munie d'opérations sur ces éléments
  - il ne définit pas la façon dont les données sont stockées
    - *i.e.* il est défini indépendamment de la manière dont on choisit de représenter les données en mémoire
  - il ne définit pas la façon dont les opérations sont implantées ; en effet, on ne sait pas comment les données sont stockées !

- Des TAD que vous connaissez sans le savoir
  - Collection séquentielle indexée (CSI)
    - une collection d'éléments tous du même type numérotés sur un intervalle [borne_inf, borne_sup]
      - implémentée par exemple avec un vecteur
  - Collection séquentielle indexée triée
    - une collection d'éléments comparables tous du même type numérotés en respectant leur ordre relatif sur un intervalle [borne_inf, borne_sup]
      - implémentée par exemple avec un vecteur trié

Notion de Type Abstrait de Données

- Un TAD est défini par :
  - *TA* : un nom du type abstrait
  - *Utilise* : les types abstraits utilisés par celui que l'on définit
  - *Opérations* : prototype de toutes les opérations que l'on peut faire sur le type abstrait
    - parmi ces opérations on trouve un **constructeur**, des **transformateurs** (*mutateurs*, *setters*) et des **observateurs** (*accesseurs*, *getters*)
    - exemples :
      - insérer un élément à la position n dans une coll. seq. indexée (CSI)
      - consulter le $n^{ième}$ dans une CSI
- Les opérations sont dotées de :
  - d'une *pré-condition* (qui doit être vérifiée pour que l'opération produise le résultat attendu)
  - d'une *post-condition* (ou *axiome*) qui décrit le comportement, l'effet, de chaque opération

Le TAD liste

- Une **liste L** est une suite finie d'éléments

$$l_1, l_2, \ldots, l_n \; ; n \geq 0$$

- On appelle :
  - **tête** de L : premier élément l_1 de L
  - **reste** de L : la **liste** $l_2, \ldots, l_n$ {le reste est une **liste**}
  - **longueur** de L : nombre d'éléments de L
  - **liste vide** : liste sans élément (de longueur 0), notée () ou NIL ou nullptr
  - **successeur** de l'élément l_i : l'élément l_{i+1} ($1 \leq i < n$)

Le TAD liste

- Les opérations associées :
  - estVide() : bool;
  - getLongueur() : int;
  - getInfoAtPosit(posit) : info; *(possible ?)*
  - *insertete(nouvelleInfo);* *// on pourrait s'en passer*
  - *supprimeTete();* *// on pourrait s'en passer*
  - insereAtPosit(position, nouvelleInfo) : bool;
  - supprimeAtPosit(position) : bool;
  - setInfoAtPosit(position, nouvelleInfo); *(possible ?)*
  - vide();

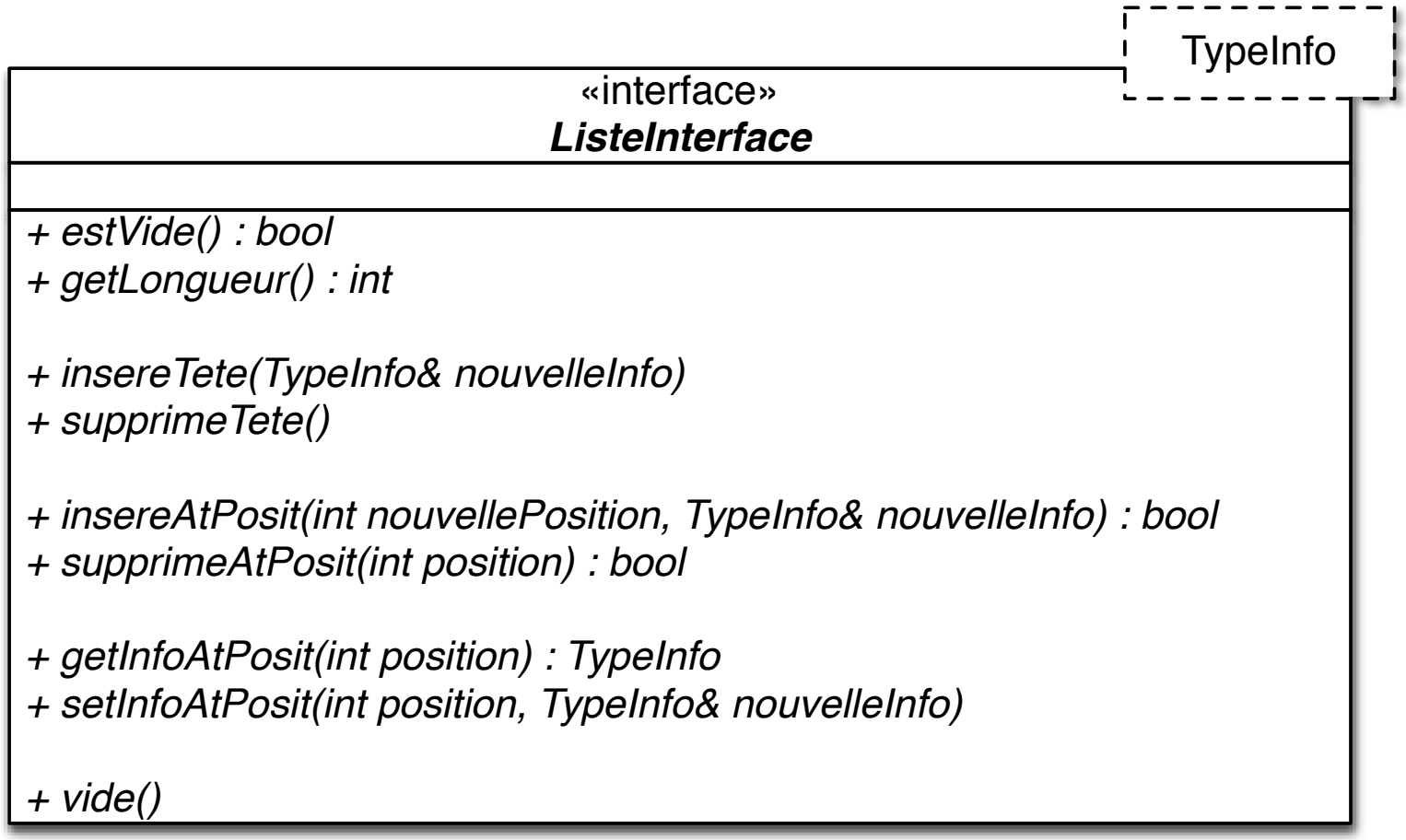
- *NOTE*
  - *pour des raisons pédagogiques certaines opérations seront implantées avec plusieurs algorithmes*

Le TAD liste

- Le TAD liste sera décrit (*implanté*) au moyen d'une classe abstraite qui …
  - définira uniquement les opérations comme des méthodes virtuelles pures
- Des classes implanteront effectivement une liste en héritant de cette classe abstraite, et …
  - … définiront la structure de données utilisée pour représenter la liste
    - ici nous utiliserons le **type liste chaînée**
  - … implanteront les opérations de liste sur la structure de données
    - ici nous implanterons des **méthodes sur une liste chaînée**

Le TAD liste : `ListeInterface`

Spécification UML



ListeInterface.h (classe abstraite associée au TAD) [début...]

```
template<class TypeInfo>
class ListeInterface {
public:
    /** cette (this) liste est-elle vide.
     @return True si la liste est vide ; False sinon. */
    virtual bool estVide() const = 0;

    /** nombre de cellules de cette (this) liste.
     @return nombre de cellules dans la liste. */
    virtual int getLongueur() const = 0;

    /** insertion d'une cellule contentant nouvelleInfo en tête de cette (this) liste.
     @pre  Aucune.
     @post une nouvelle cellule contenant nouvelleInfo est ajouté en tête de cette liste.
     @param nouvelleInfo  information portée par la nouvelle cellule de tête. */
    virtual void insereTete(const TypeInfo& nouvelleInfo) = 0;

    /** suppression de la première cellule de cette liste.
     @pre  Aucune.
     @post cette liste est privée de sa première cellule. */
    virtual void supprimeTete() = 0;

    /** insertion d'une cellule contentant nouvelleInfo à la position nouvellePosition ;
     @pre  Aucune.
     @post  si 1 <= position <= getLongueur() + 1 et l'insertion est réussie,
        nouvelleInfo est la nouvellePosition dans cette liste,
        et la valeur retournée est True ; sinon la valeur retournée est False
     @param nouvellePosition  position à laquelle insérer nouvelleInfo.
     @param nouvelleInfo  information à insérer dans la liste.
     @return  True si l'insertion est réussie, False sinon. */
    virtual bool insereAtPosit(int nouvellePosition, const TypeInfo& nouvelleInfo) = 0;

    …
```

ListeInterface.h (classe abstraite associée au TAD) [...fin]

```
    ...

    /** suppression de la cellule visée au rang position ;
     @pre  Aucune.
     @post  si 1 <= position <= getLongueur() la suppression est réussie,
        la cellule visée est supprimé de cette liste,
        and la valeur retournée est True ; sinon la valeur retournée est False.
     @param position  rang de la cellule visée à supprimer.
     @return  True si la suppression est réussie, False sinon. */
    virtual bool supprimeAtPosit(int position) = 0;

    /** suppression de toutes les cellules de la liste.
     @post  la liste est vide et le nombre de cellules vaut 0. */
    virtual void vide() = 0;

    /** valeur de l'information portée par la cellule visée de rang position.
     @pre  1 <= position <= getLongueur ().
     @post  l'information portée par la cellule visée est retournée.
     @param position  rang de la cellule visée.
     @return  information portée par la cellule visée. */
    virtual TypeInfo getInfoAtPosit(int position) const = 0;

    /** remplacement de l'information portée par la cellule visée au rang position.
     @pre  1 <= position <= getLongueur ().
     @post  l'information de la cellule visée vaut nouvelleInfo.
     @param position  rang de la cellule visée.
     @param nouvelleInfo  nouvelle information de la cellule visée. */
    virtual void setInfoAtPosit(int position, const TypeInfo& nouvelleInfo) = 0;

}; // end ListInterface
```

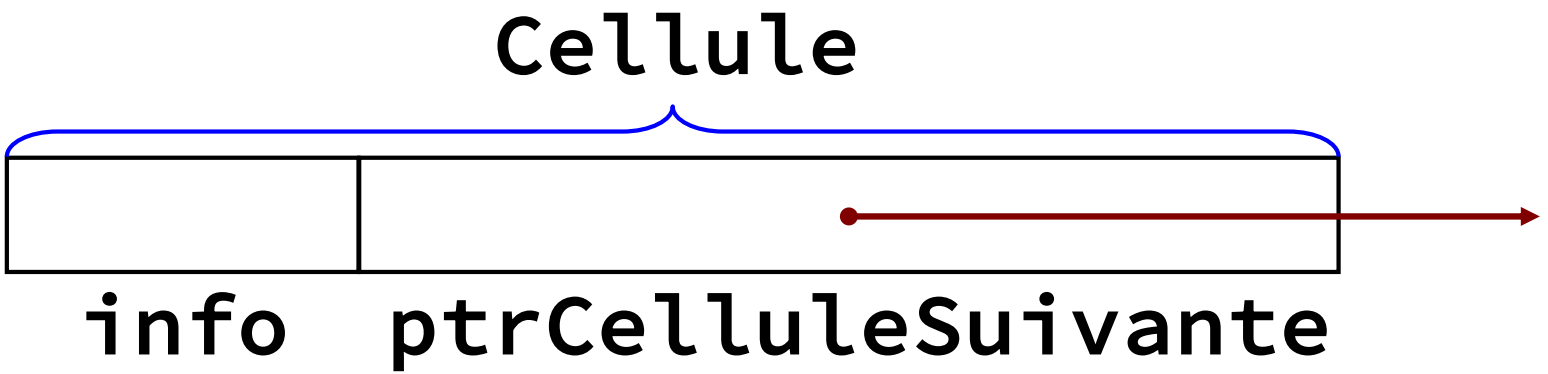
La classe `ListeInterface` en TD/TP

- On implantera quelques méthodes avec des algorithmes différents
  - **`insereAtPosit`**
    - `insereAtPositIter` `//itératif pur`
    - `insereAtPositRec` `//worker procédure récursive`
  - **`supprimeAtPosit`**
    - `supprimeAtPositIter` `//itératif pur`
    - `supprimeAtPositRec` `//worker procédure récursive`
  - **`vide`**
    - `videIter` `//itératif pur`
    - `videRec` `//worker procédure récursive`
  - **`getInfoAtPosit`**
    - `getInfoAtPositRec` `//worker fonction récursive`
  - **`setInfoAtPosit`**
    - `setInfoAtPositRec` `//worker procédure récursive`

La classe `Cellule`

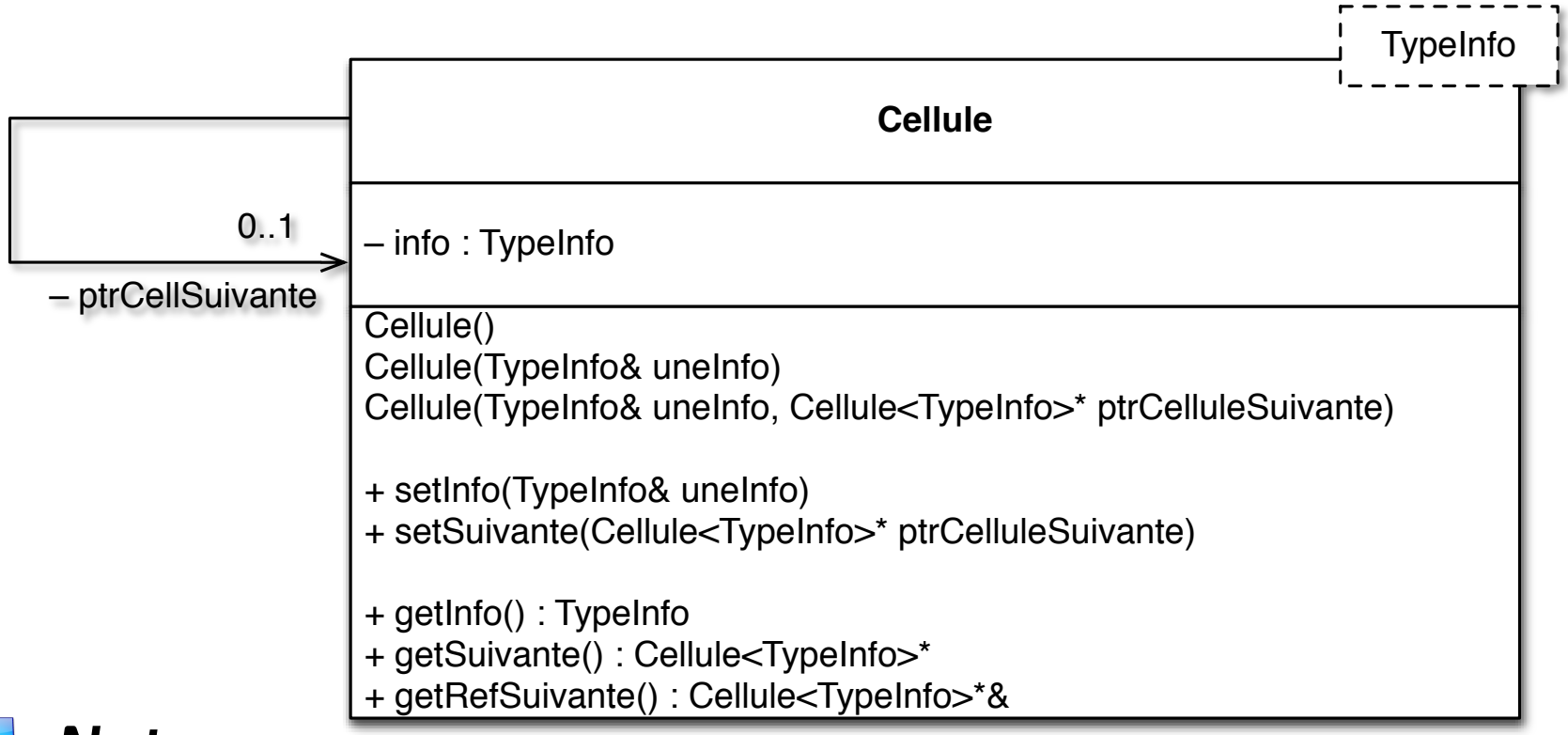
- On va définir la classe **`Cellule`** pour représenter un élément d'une liste chainée

- Une **`Cellule`** à deux attributs :



- On propose une classe modèle (template) pour manipuler des listes chaînées contenant des **`infos`** de n'importe quel type comparable

La classe `Cellule`



Notes

- une **`Cellule`** aura pour attribut un pointeur sur une **`Cellule`** suivante (**`ptrCellSuivante`**)
- une **`Cellule`** peut ne pas avoir de **`Cellule`** suivante
  - dans ce cas **`ptrCellSuivante`** vaut un pointeur nul (**`nullptr`** en C++11)

cellule.h [première version]

```
template<class TypeInfo>            // classe modèle (template) sur le type d'information
class Cellule {
private:
    TypeInfo info;                  // information portée par la cellule de type TypeInfo
    Cellule<TypeInfo>* ptrCellSuivante; // pointeur sur la cellule suivante

public:
    //Constructeur par défaut de cette cellule
    Cellule();

    //Constructeur de cette cellule avec une information uneInfo comme contenu
    Cellule(const TypeInfo& uneInfo);

    //Constructeur qui insère une cellule contenant uneInfo en tête de ptrCelluleSuivante
    Cellule(const TypeInfo& uneInfo, Cellule<TypeInfo>* ptrCelluleSuivante);

    //Setters pour info et ptrCellSuivante
    void setInfo(const TypeInfo& uneInfo);
    void setSuivante(Cellule<TypeInfo>* ptrCelluleSuivante);

    //Getter pour obtenir l'info de cette cellule
    TypeInfo getInfo() const;  //const car la méthode ne modifie pas cet objet

    //Getter d'un pointeur vers la cellule suivant cette cellule
    // @return pointeur vers une cellule
    Cellule<TypeInfo>* getSuivante() const;

    //Getter d'une référence de pointeur vers la cellule suivant cette cellule
    // @return référence sur un pointeur vers une cellule
    // /!\ cette méthode est nécessaire pour l'écriture de procédures récursives
    // dans le modèle "intuitif"
    // Ne peut être const car le pointeur retourné peut-être modifié
    Cellule<TypeInfo>*& getRefSuivante();
}; // end Cellule
```

Illustration

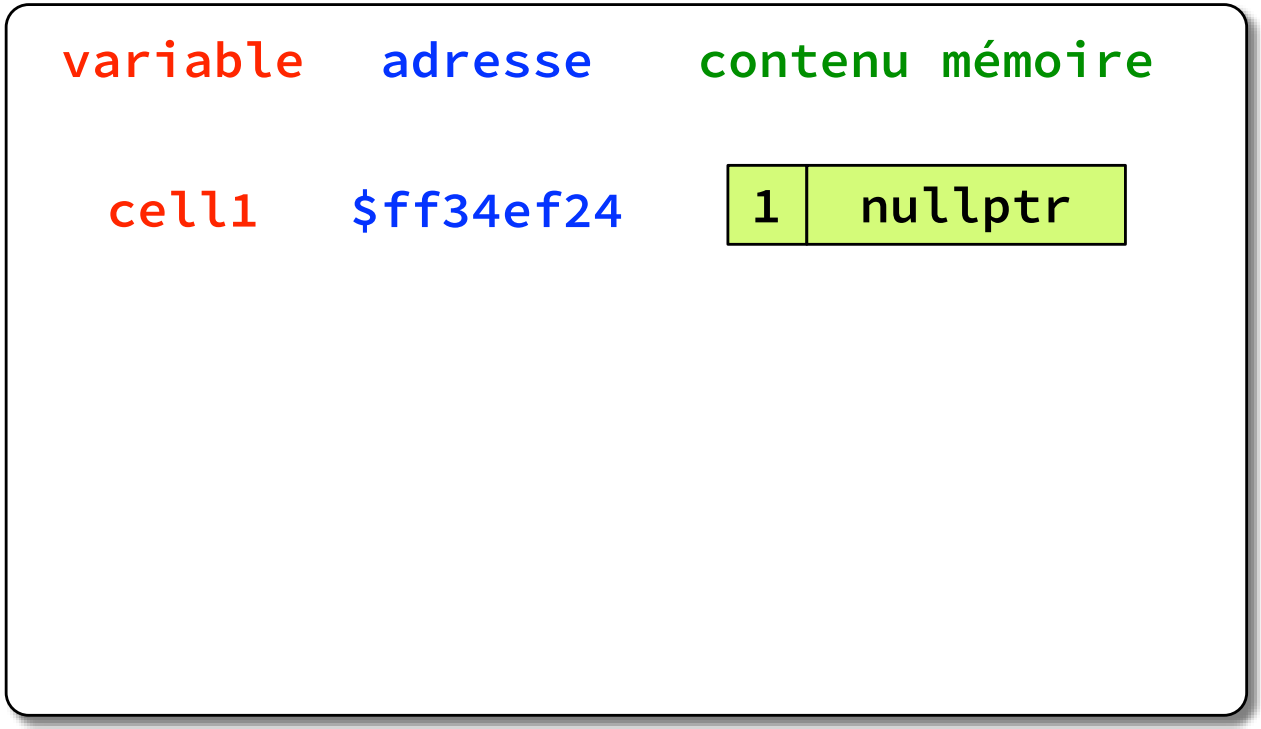
- Faisons la trace du programme suivant :

code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```

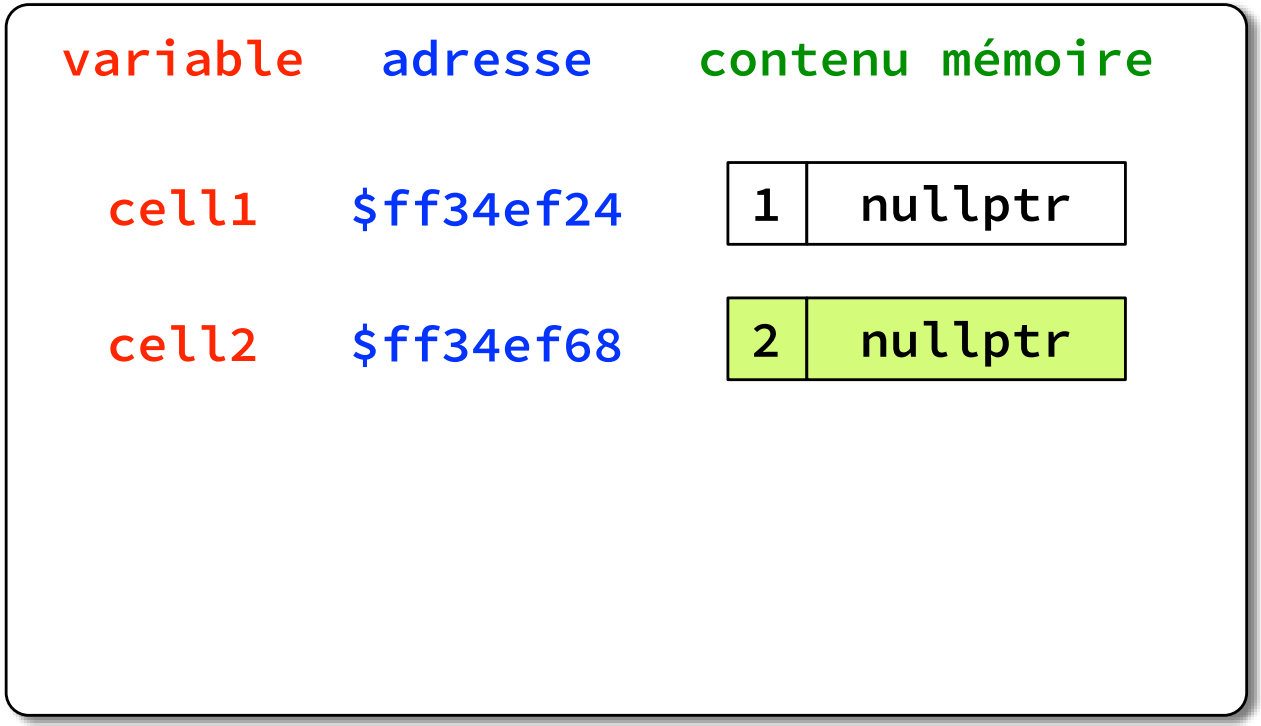
code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```



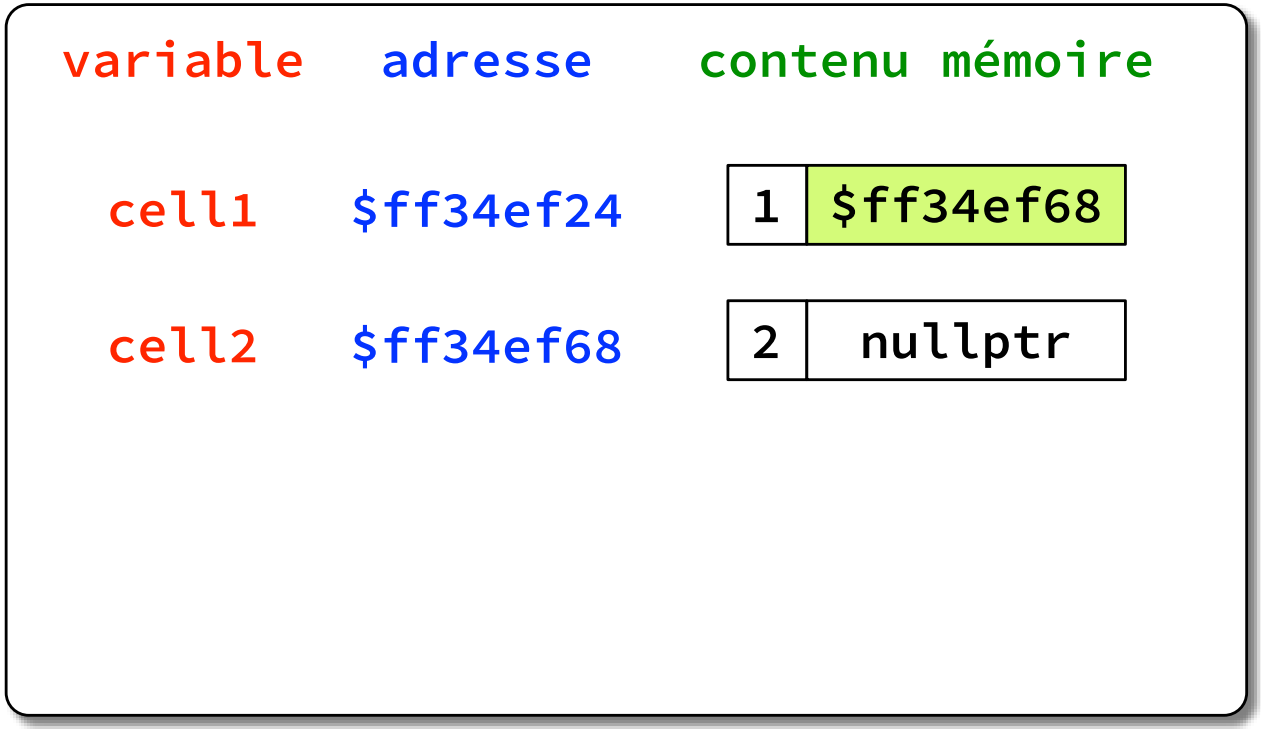
code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```



code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```



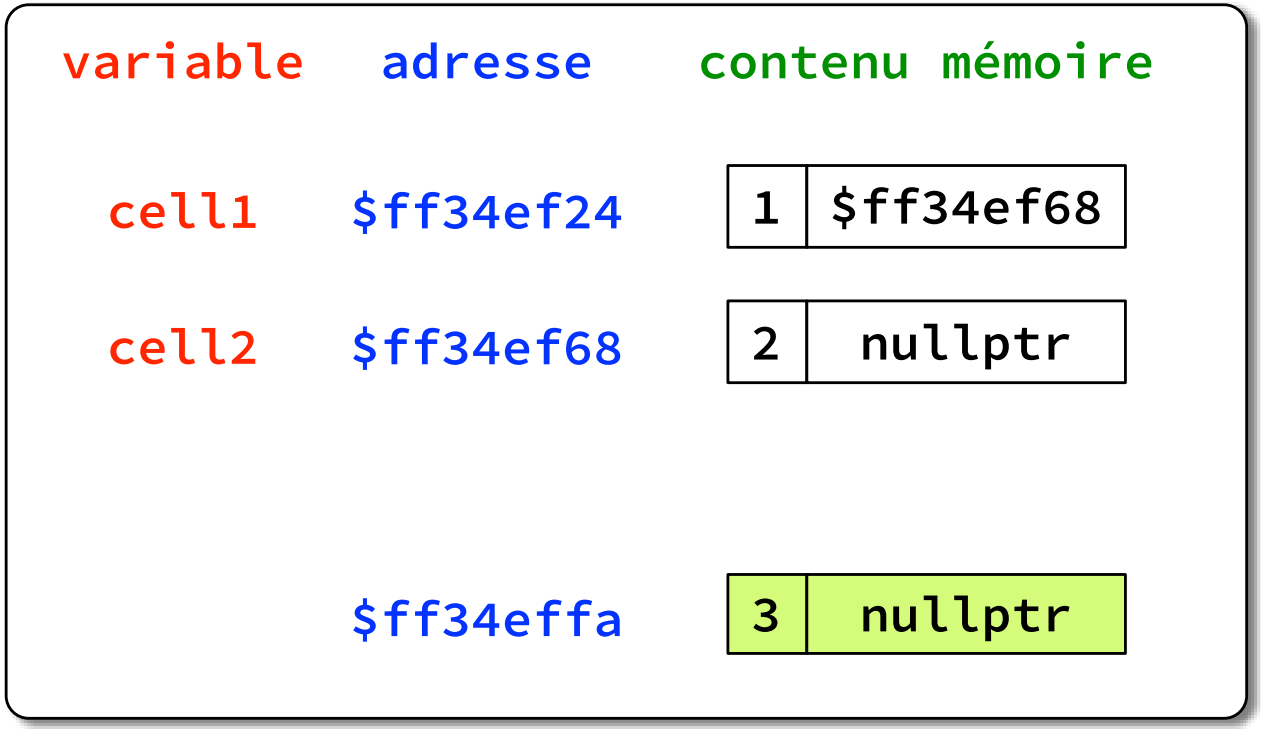
code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```

new Cellule<int>(3)

retourne

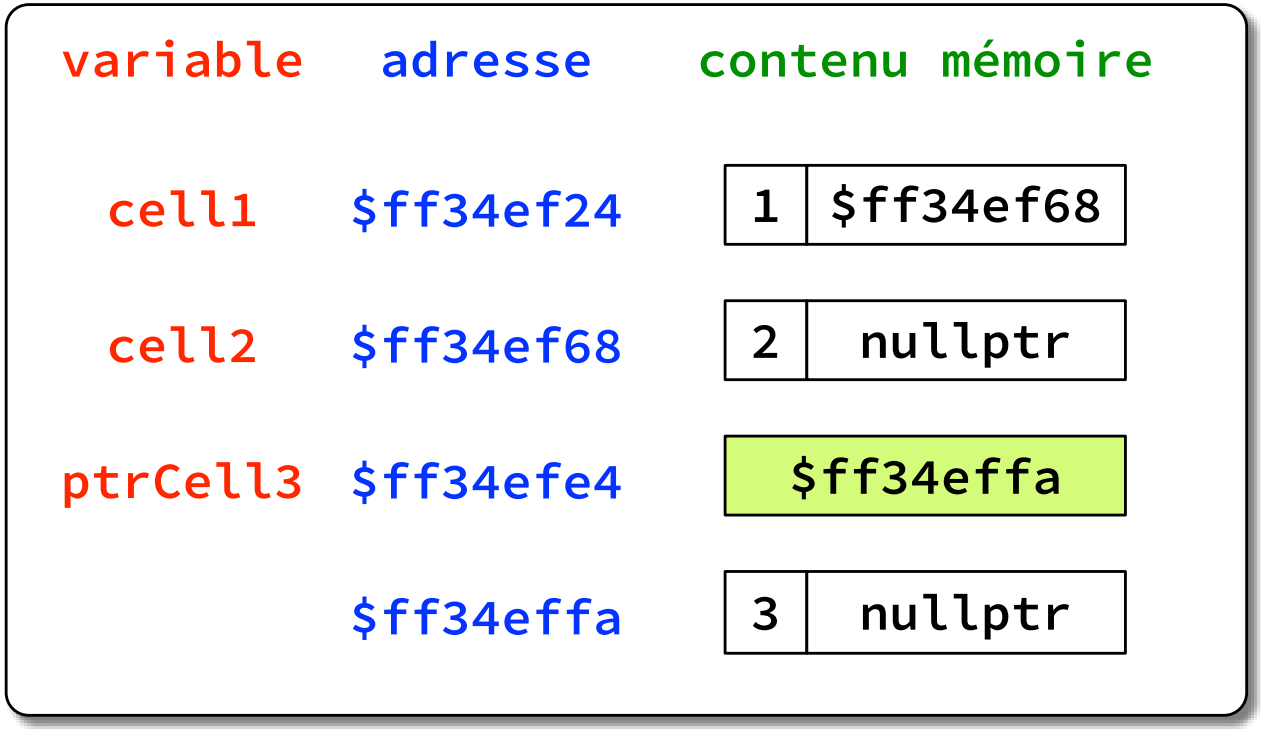
$ff34effa



code C++

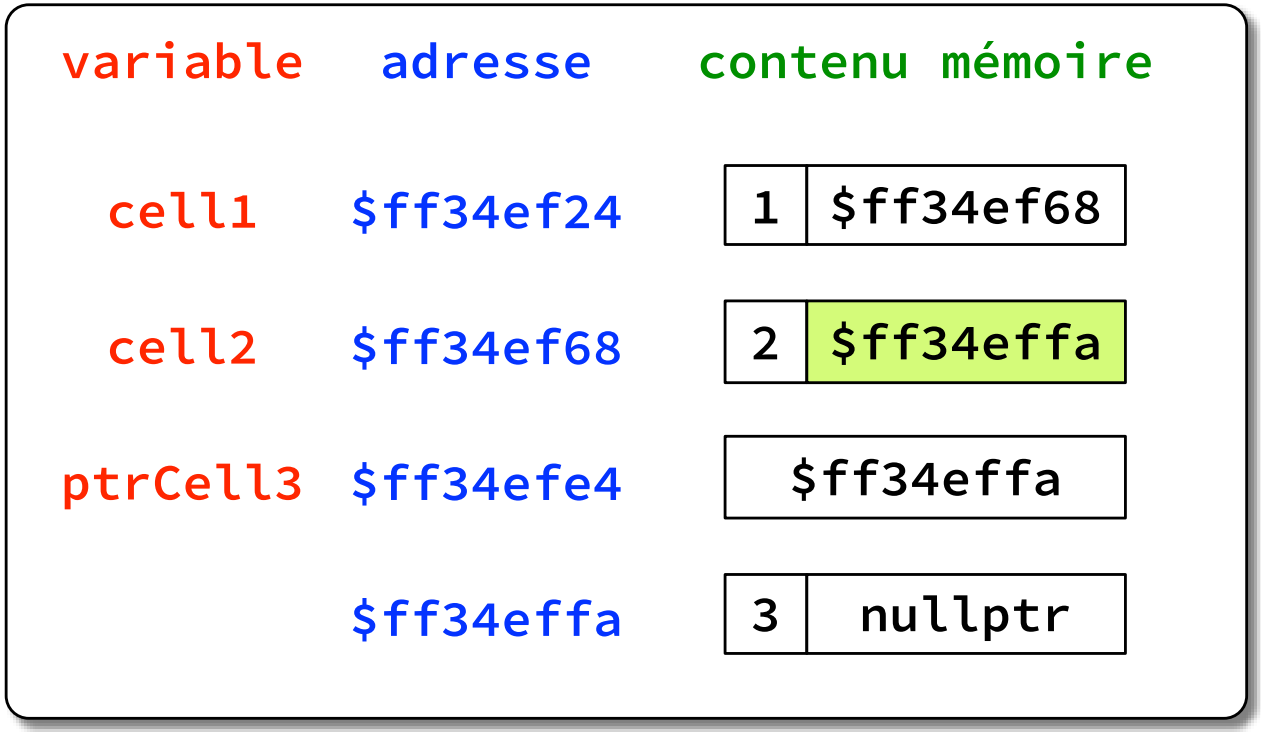
```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```

initialisation de **ptrCell3**



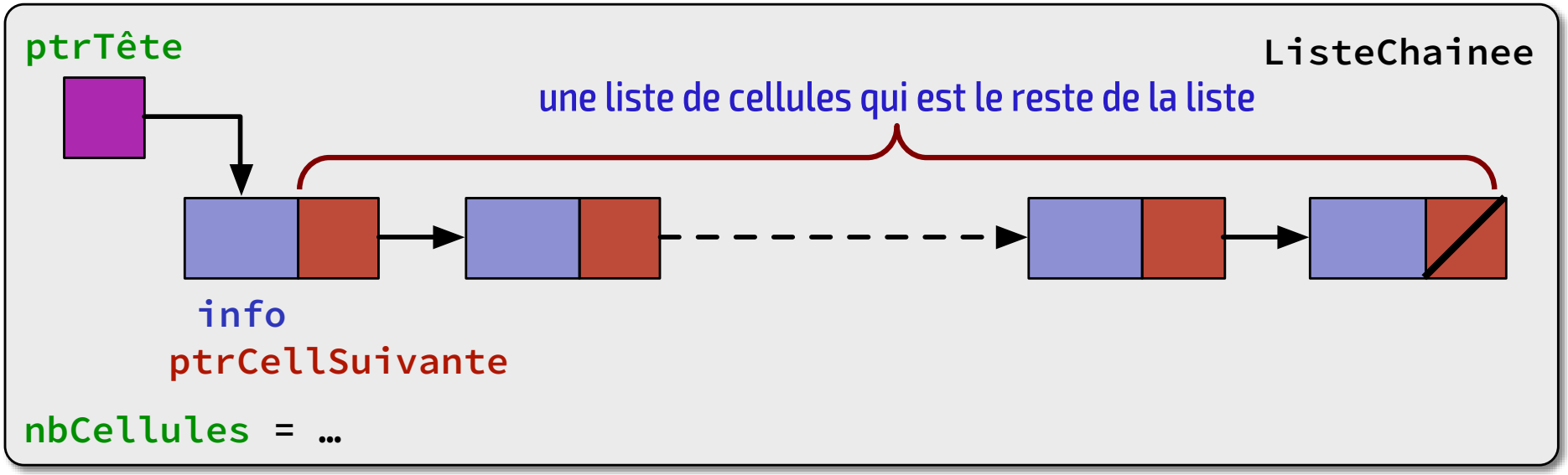
code C++

```
01  #include "Cellule.h »
02  using namespace std;
03  int main() {
04      Cellule<int> cell1(1);          // constructeur, 1 dans cell1
05      Cellule<int> cell2(2);          // constructeur, 2 dans cell2
06      // màj ptrCellSuivante de Cell1 avec l'adresse de cell2 (&Cell2)
07      maCell1.setSuivante(&Cell2);
08      // ptrCell3 est l'adresse d'une Cellule qui contient 3
09      Cellule<int>* ptrCell3 = new Cellule<int>(3);
10      // màj ptrCellSuivante de Cell1 avec l'adresse ptrCell3
11      maCell2.setSuivante(ptrCell3); // màj ptrCellSuivante de Cell2
12  }
```



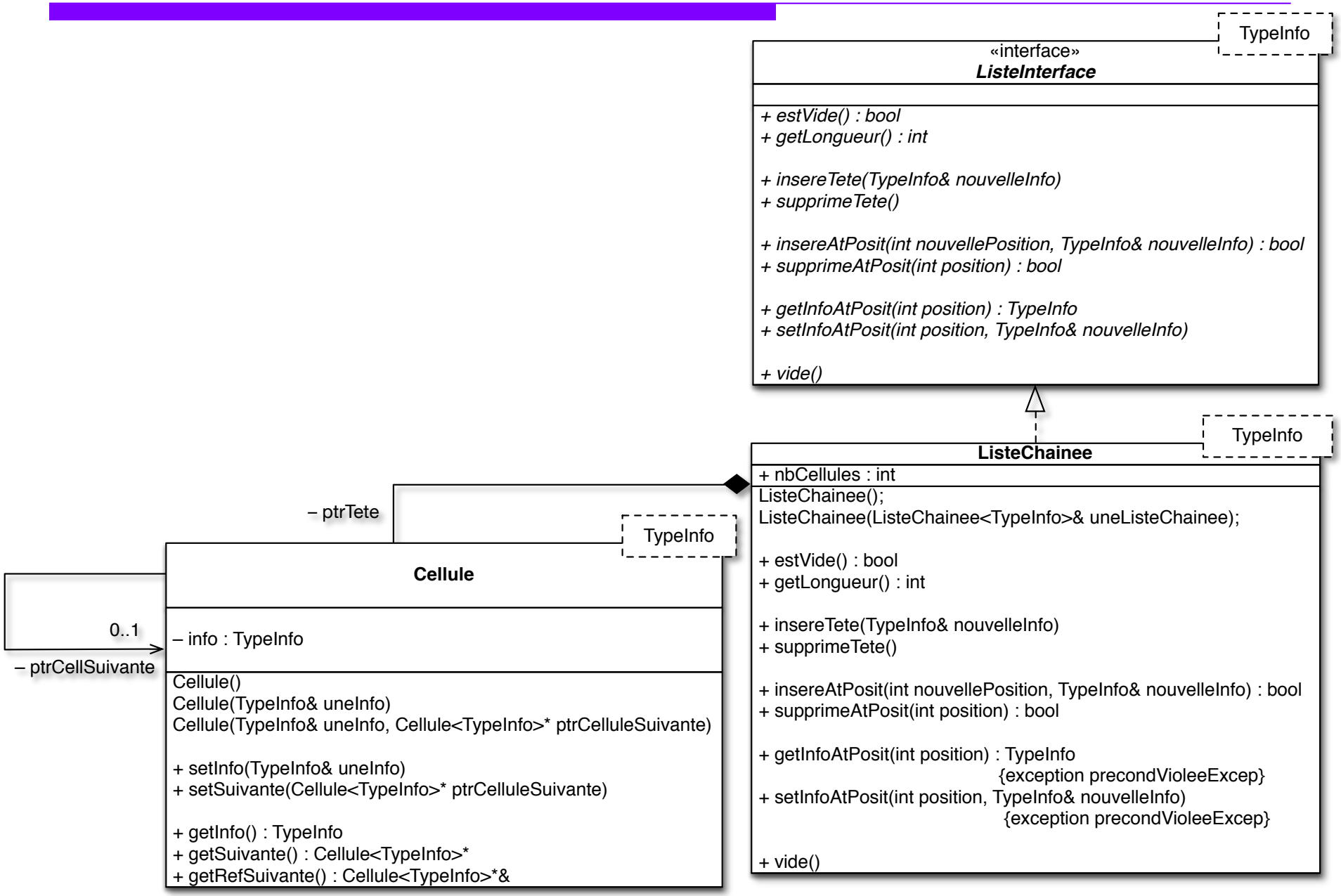
La classe `ListeChainee`

- Implantation du TAD liste au moyen d'une liste chaînée de **`Cellules`**



- **`ptrTête`** est un pointeur sur la première **`Cellule`**
  - chaque **`Cellule`** pointe sur son successeur
  - la dernière **`Cellule`** n'a pas de successeur (**`nullptr`**)
- **`nbCellules`** est le nombre de **`Cellules`** de la liste

La classe `ListeChainee`



ListeChainee.h

```
template<class TypeInfo>
class ListeChainee : public ListeInterface<TypeInfo> {
private:
    //structure de données pour implanter effectivement la liste
    Cellule<TypeInfo>* ptrTete; // pointeur sur la première Cellule de la liste;
    int nbCellules;             // nombre de cellules

    //méthodes utilitaires privées utilisées par les méthodes publiques
    ...

public:
    ListeChainee();
    ListeChainee(const ListeChainee<TypeInfo>& uneListeChainee);

    //-------------------------------------------------------------------------
    // Méthodes publiques : définition des méthodes virtuelles de ListeInterface
    //-------------------------------------------------------------------------
    bool estVide() const;

    int getLongueur() const;

    void insereTete(const TypeInfo& nouvelleInfo);
    void supprimeTete();

    bool insereAtPosit(int nouvellePosition, const TypeInfo& nouvelleInfo);
    bool supprimeAtPosit(int position);

    void vide();

    TypeInfo getInfoAtPosit(int position) const throw (PrecondVioleeExcep);
    void setInfoAtPosit(int position, const TypeInfo& nouvelleInfo)
                                                    throw (PrecondVioleeExcep);
};
```

Où j’ai une idée de comment ça marche !

ILLUSTRATION

code C++

```
01  #include <iostream>
02  #include <string>
03  #include "ListeChainee.h"
04
05  using namespace std;
06
07  int main() {
08      ListeChainee<string>* ptrMaListe = new ListeChainee<string>();
09      ptrMaListe->insereAtPosit(1, "un");
10      ptrMaListe->insereAtPosit(2, "deux");
11      ptrMaListe->insereAtPosit(2, "entreUnEtDeux");
12      ptrMaListe->afficheListe();
13      ptrMaliste->supprimeTete();
14      ptrMaListe->afficheListe();
15      ptrMaListe->insereTete("nouveauUn");
16      ptrMaListe->afficheListe();
17      cout << "Second : " << ptrMaListe->getInfoAtPosit(2) << endl;
18  }
19
```

trace

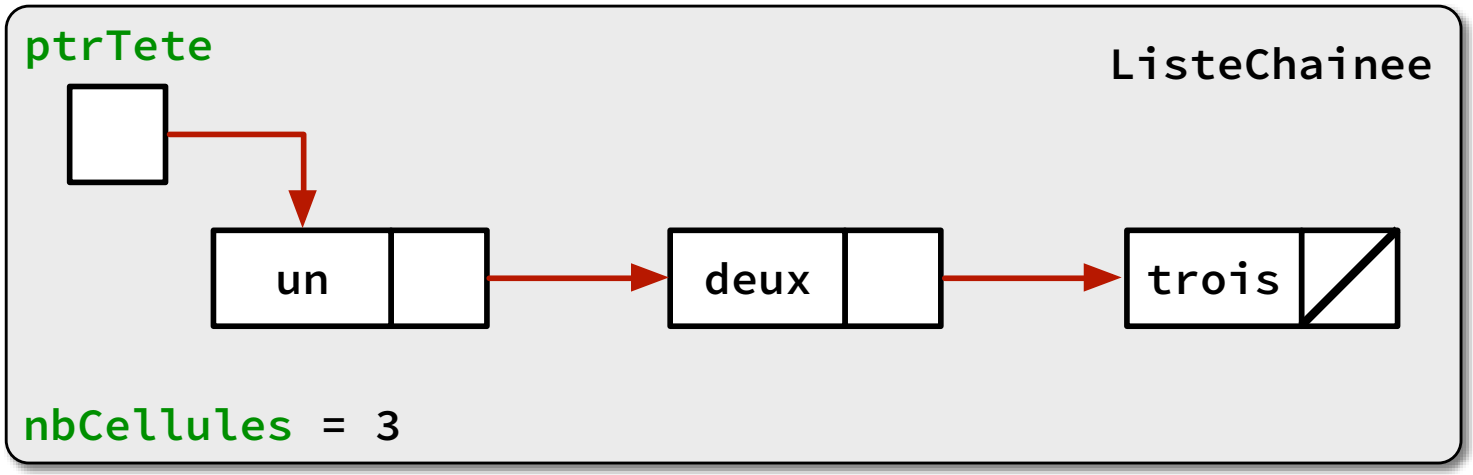
```
12  La liste contient -> un entreUnEtDeux deux
14  La liste contient -> entreUnEtDeux deux
16  La liste contient -> nouveauUn entreUnEtDeux deux
17  Second : entreUnEtDeux
```

Où je profite complètement de la définition inductive !

ALGORITHMES RÉCURSIFS

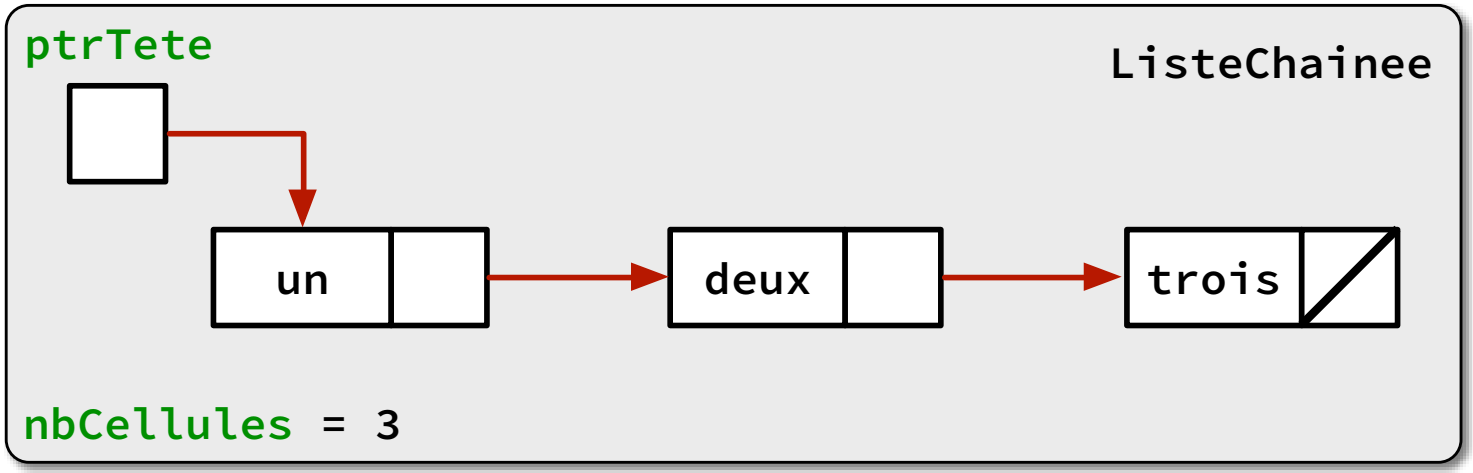
Affichage d'une `ListeChainee`

- Une **`ListeChainee`** comporte un pointeur sur sa première **`Cellule`** …
- … il faut donc parcourir toutes les **`Cellules`** de la **`ListeChainee`** à partir de la première
  - soit **`ptrTete`** de type **`Cellule<TypeInfo>*`**
- … en affichant l'information portée par chacune



Affichage d'une `ListeChainee`

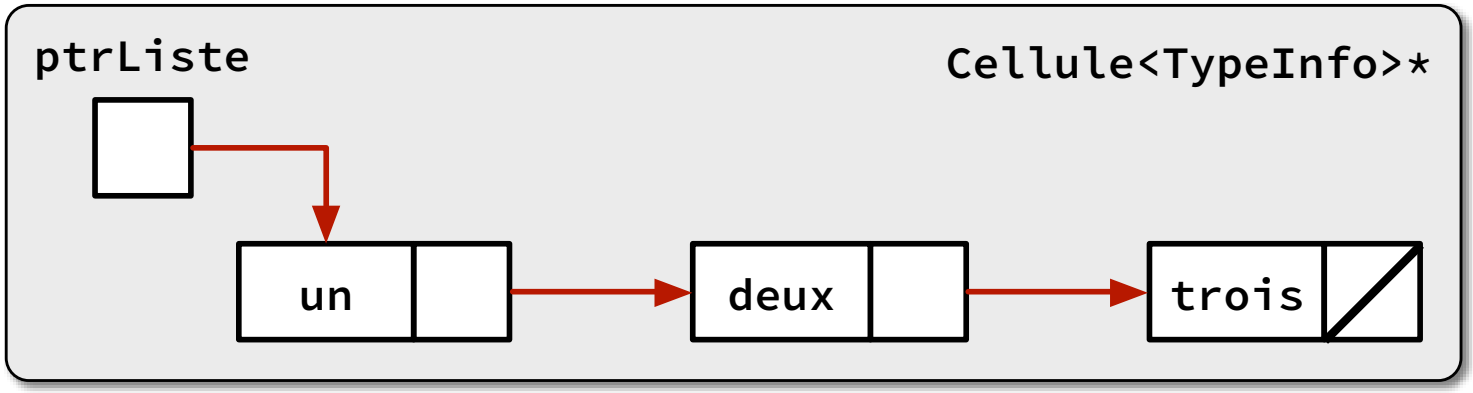
- Pour la **`ListeChainee`** suivante :



- On pourrait produire
  - **`un deux trois`**
    - parcours de gauche à droite
  - **`trois deux un`**
    - parcours de droite à gauche

Affichons une liste de `Cellules`

- On souhaite afficher :



- Imaginons que l'on dispose de 2 procédures

```
void afficheCellulesGD(const Cellule<TypeInfo>* ptrListe) const;
// affiche l'information des Cellules atteignables depuis
// ptrListe en faisant un parcours de gauche à droite
```

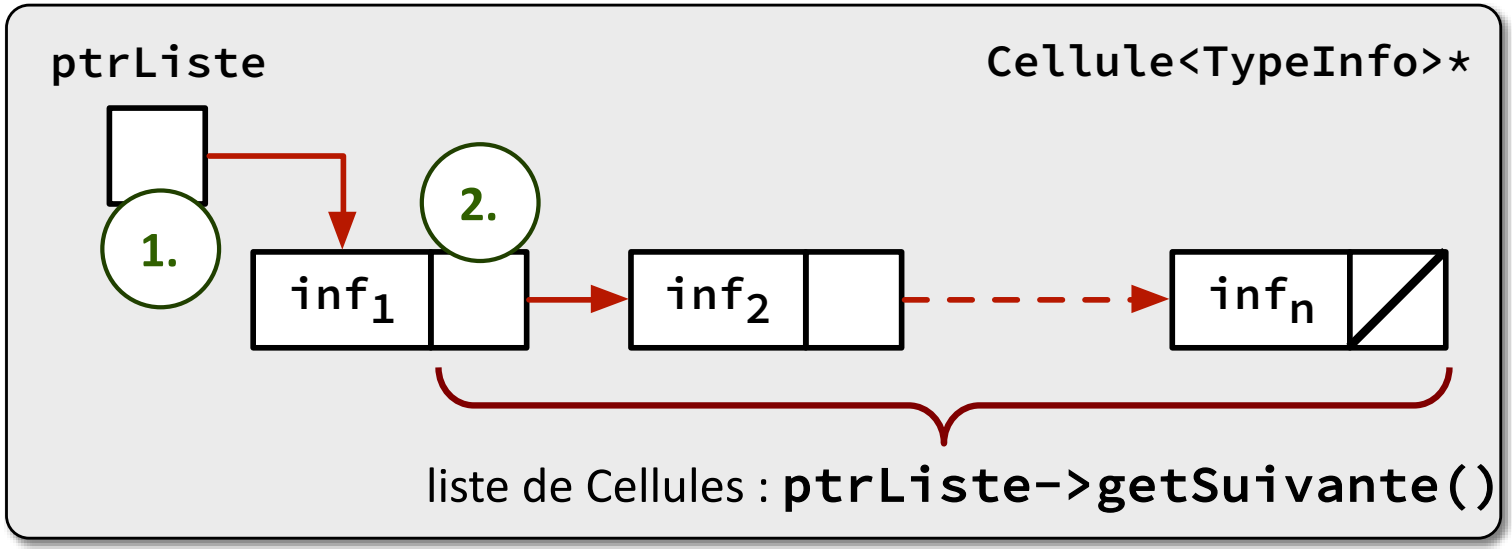
```
void afficheCellulesDG(const Cellule<TypeInfo>* ptrListe) const;
// affiche le l'information des Cellules atteignables depuis
// ptrListe en faisant un parcours de droite à gauche
```

Affichage d'une liste de `Cellules`

- **`ptrListe`**, une liste de **`Cellules`**, est ...
  - soit vide (elle ne contient aucune **`Cellule`**)
    - **`ptrListe == nullptr`**
  - soit composée d'une première **`Cellule`** chaînée à une liste de **`Cellules`** (celles qui suivent la première)
    - **`ptrListe != nullptr`**
    - **`ptrListe->getInfo()`**
      - est l'information portée par la première **`Cellule`**
    - **`ptrListe->getSuivante()`**
      - est un pointeur sur la première **`Cellule`** du reste de la liste

Affichage d'une liste de `Cellules` GD

- Si la liste de **`Cellules`** est vide :
  - rien à faire !
- Si la liste de **`Cellules`** n'est pas vide :



1. d'abord afficher le contenu de la première **`Cellule`**
2. puis afficher la liste de **`Cellules`** qui suivent la première

Affichage d'une liste de `Cellules` GD

- Réfléchissons à l'écriture de `afficheCellulesGD(Cellule<TypeInfo>* ptrListe)` en supposant qu'elle existe

- la liste de **`Cellules`** est vide
  - **`ptrListe == nullptr`** → il n'y a rien à faire ✳
- la liste de **`Cellules`** n'est pas vide
  - **`ptrListe != nullptr`** →
    1. afficher l'info de la 1ière **`Cellule`** : **`ptrListe->getInfo()`**
    2. afficher le reste de la liste **`ptrListe->getSuivante()`**
       - comment ??
       - en utilisant **`afficheCellulesGD(…)`** faite pour ça !!!

Affichage d'une liste de `Cellules` GD

- En résumé
  - `ptrListe == nullptr` → c'est terminé ✳
  - `ptrListe != nullptr` →

```
cout << this->getInfo() << " ";
afficheCellulesGD(ptrListe->getSuivante());
```

- Procédure

procédure récursive

```
void afficheCellulesGD(Cellule<TypeInfo>* ptrListe) const {
    if (ptrListe) {
        cout << ptrListe->getInfo() << " ";
        afficheCellulesGD(ptrListe->getSuivante());
    }
    // sinon rien à faire, donc on ne fait rien !!!
}
```

Une procédure c'est bien …

- En C++, mettre `afficheCellulesGD(…)` dans une classe !
- Laquelle ?
  - la classe `Cellule` : **NON**, une `Cellule` n'est pas une liste !
  - la classe `ListeChainee` : **OUI**, je n'ai pas le choix !
- Comment ?
  - le paramètre de `afficheCellulesDG(…)` est de type `Cellule<TypeInfo>*`
  - `afficheCellulesGD(…)`…
    - n'est **pas une méthode** publique,
    - c'est **une procédure ordinaire** de service

Une procédure c'est bien …

- Solution !
  - cacher **`afficheCellulesGD(…)`** dans la classe **`ListeChainee`**
  - ce sera une procédure privée de **ListeChainee**
- Comment je la nomme ?
  - ce n'est pas une méthode ordinaire
  - il faudrait que je puisse la reconnaître facilement
- Une convention de nommage !
  - `maMethodeWorker(…)` → une procédure récursive qui fait le travail de `maMethode()` qui ne peut le faire elle-même

Finalement ... GD

- La procédure ...

```
void afficheCellulesGD(Cellule<TypeInfo>* ptrListe) const {
    if (ptrListe) {
        cout << ptrListe->getInfo() << " ";
        afficheCellulesGD(ptrListe->getSuivante());
    }
    // sinon rien à faire, donc on ne fait rien !!!
} // end afficheCelluleGD
```

- ... devient une **procédure privée** de

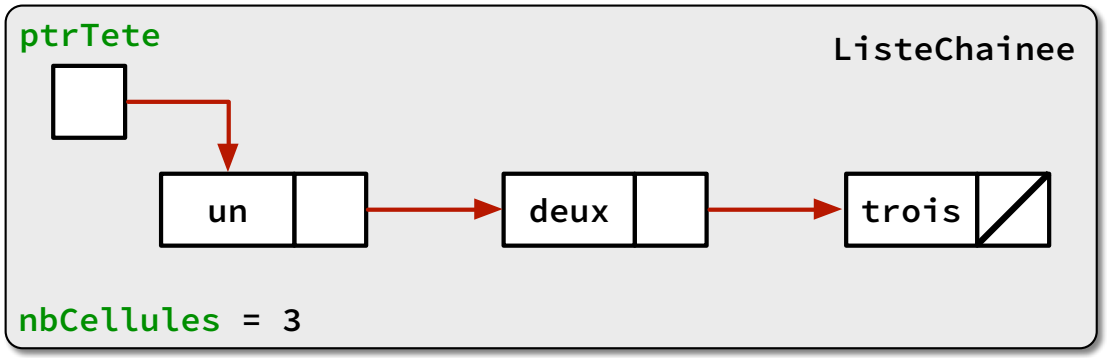
```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheGDWorker(Cellule<TypeInfo>* ptrListe) const {
    if (ptrListe) {
        cout << ptrListe->getInfo() << " ";
        afficheGDWorker(ptrListe->getSuivante());
    }
    // sinon rien à faire, donc on ne fait rien !!!
}
```

Mais ce n'est pas fini ... GD

afficheGD()

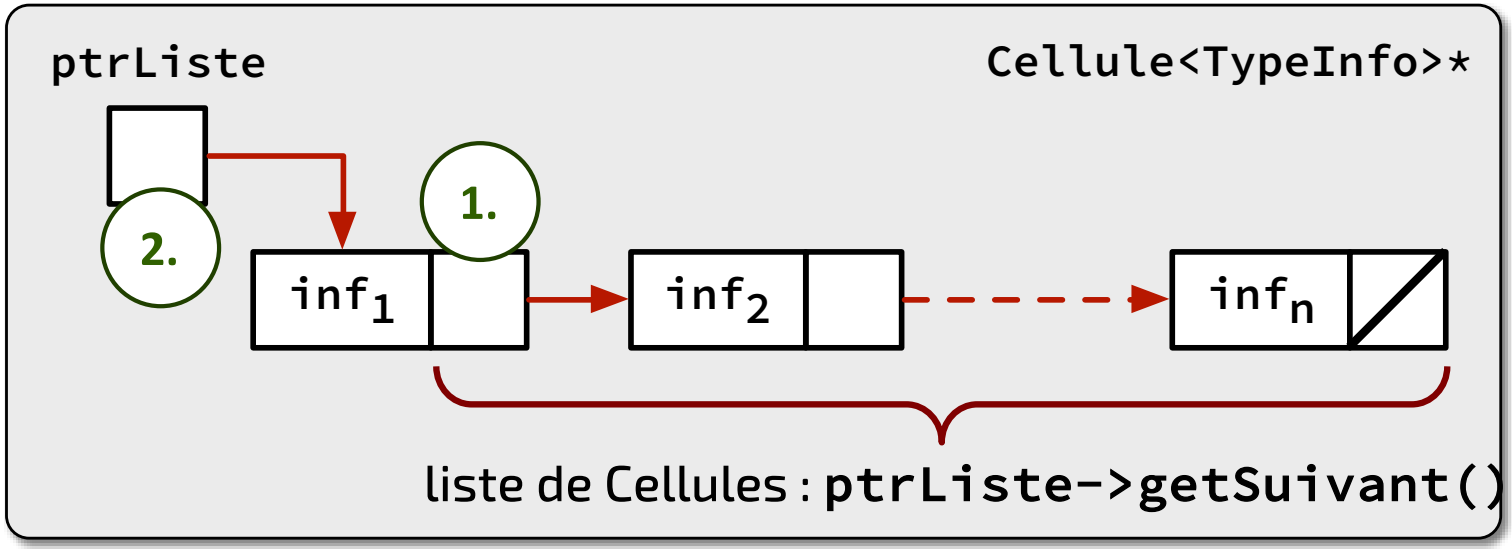
- **méthode publique** de la classe **ListeChainee**

```
template<typename TypeInfo>
void ListeChainee<TypeInfo>::afficheGD() {
  cout << "En parcours de gauche à droite, la liste contient : ";
  // afficheGDWorker privée, affiche une liste de Cellules
  afficheGDWorker(ptrTete);
  cout << endl;
} // end afficheGD
```



Affichage d'une liste de `Cellules` DG

- Si la liste de **`Cellules`** est vide :
  - rien à faire !
- Si la liste de **`Cellules`** n'est pas vide :



1. d'abord afficher la liste de **`Cellules`** qui suivent la première
2. puis afficher le contenu de la première **`Cellule`**

Affichage d'une liste de `Cellules` DG

- Réfléchissons à l'écriture de `afficheCellulesDG(Cellule<TypeInfo>* ptrListe)` en supposant qu'elle existe

- la liste de **`Cellules`** est vide
  - **`ptrListe == nullptr`** → il n'y a rien a faire ✳
- la liste de **`Cellules`** n'est pas vide
  - **`ptrListe != nullptr`** →
    1. afficher le reste de la liste **`ptrListe->getSuivante()`**
       - comment ??
       - en utilisant la méthode **`afficheCellulesDG()`** faite pour ça !!!
    2. afficher l'info de la 1ière **`Cellule`** : **`ptrListe->getInfo()`**

Affichage d'une liste de `Cellules` DG

- En résumé
  - `ptrListe == nullptr` → c'est terminé ✳
  - `ptrListe != nullptr` →

```
affichecellulesDG(ptrListe->getSuivante());
cout << ptrListe->getInfo() << " ";
```

- Procédure

procédure récursive

```
void affichecellulesDG(Cellule<TypeInfo>* ptrListe) const {
    if (ptrListe) {
        affichecellulesDG(ptrListe->getSuivante());
        cout << ptrListe->getInfo() << " ";
    }
    // sinon rien à faire, donc on ne fait rien !!!
}
```

Pour finir… DG

- **afficheDGWorker(…)** *(procédure privée)*

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheDGWorker(Cellule<TypeInfo>* ptrListe) const
{
    if (ptrListe) {
        afficheDGWorker(ptrListe->getSuivante());
        cout << ptrListe->getInfo() << " ";
    }
    // sinon rien à faire, donc on en fait rien !!!
}
```

- **afficheDG()** *(méthode publique)*

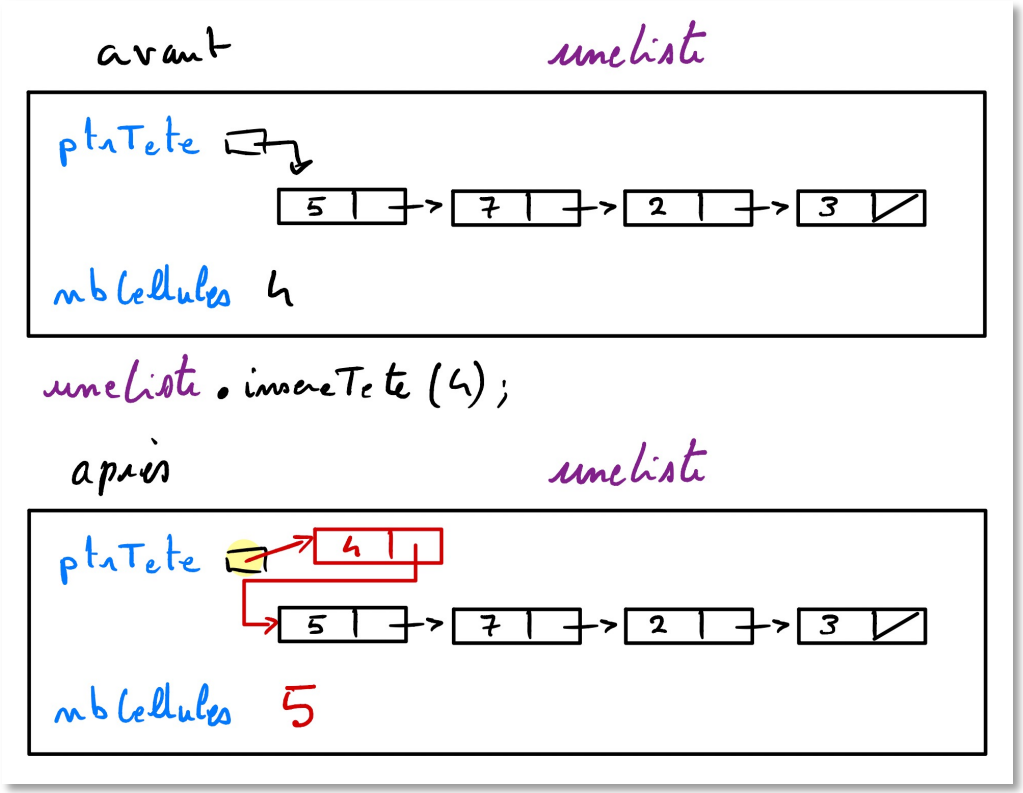
```
template<typename TypeInfo>
void ListeChainee<TypeInfo>::afficheDG() {
  cout << "En parcours de droite à gauche, la liste contient : ";
  // afficheDGWorker privée, affiche une liste de Cellules
  afficheDGWorker(ptrTete);
  cout << endl;
} // end afficheDG
```

De l’utilisation de la référence comme paramètre donnée

MODIFICATION D’UNE LISTE CHAÎNÉE

Insertion en tête

- Un dessin



- Une méthode publique

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::insereTete(const TypeInfo& nouvelleInfo) {
    // appel de l'insertion en tête dans la liste ptrTete
    insereTeteWorker(ptrTete, nouvelleInfo);
}
```

Insertion en tête

- Un worker privé
  - il n'est pas obligatoire mais va servir à d'autres

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::insereTeteWorker(
                Cellule<TypeInfo>*& ptrCetteListe,  // résultat
                const TypeInfo& nouvelleInfo) {
  // création d'une cellule contenant la nouvelleInfo
  Cellule<TypeInfo>* ptrNouvelleCellule =
                                new Cellule<TypeInfo>(nouvelleInfo);
  // la cellule suivante de cette cellule est la tête actuelle
  ptrNouvelleCellule->setSuivante(ptrCetteListe);
  // la nouvelle cellule de tête est la cellule contenant
nouvelleInfo
  // MODIFICATION du paramètre résultat
  ptrCetteListe = ptrNouvelleCellule;
  //une nouvelle cellule
  nbCellules++;
}
```

Insertion à position

- On numérote les positions à partir de 1
- On a la méthode publique suivante :

```
template<class TypeInfo>
bool ListeChainee<TypeInfo>::insereAtPositRec(
                int nouvellePosition,
                const TypeInfo& nouvelleInfo) {
  // si insertion possible appeller insereAtPositRecProcWorker()
  // qui met à jour cette liste
  bool insertionPossible = (nouvellePosition >= 1)
                        && (nouvellePosition <= nbCellules + 1);
  if (insertionPossible) {
    // mise à jour ce dette liste (ptrTete)
    insereAtPositRecWorker(ptrTete,
                           nouvellePosition,
                           nouvelleInfo);
  }
  return insertionPossible;
}
```

Le worker : la signature

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::insereAtPositRecProcWorker(
        Cellule<TypeInfo>*& ptrCetteListe,    // résultat
        const int nouvellePosition,
        const TypeInfo& nouvelleInfo) {
```

- Commentaire :
  - ptrCetteListe est un paramètre résultat
    - on peut devoir faire une insertion en tête

Le worker : les situations

- Base (situation triviale) : nouvellePosition = 1
  - c'est une insertion en tête sur ptrTete, utiliser `insereTeteWorker(ptrTete, nouvelleInfo)`
- Récurrence :
  - il faut insérer plus loin !
    - comment avancer ? deux méthodes
      - `Cellule<TypeInfo>* getSuivante() const;`
      - `Cellule<TypeInfo>*& getRefSuivante();`
    - on ne veux pas un pointeur sur une Cellule
    - on veut l'adresse d'un pointeur sur une Cellule : une référence !
  - à quelle position ?
    - nouvellePosition-1

Le worker : le code

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::insereAtPositRecProcWorker(
            Cellule<TypeInfo>*& ptrCetteListe,
            int nouvellePosition,
            const TypeInfo& nouvelleInfo) {

  if (nouvellePosition == 1) { //insertête avec worker

    insereTeteWorker(ptrCetteListe, nouvelleInfo);

  } else {

    insereAtPositRecWorker(ptrCetteListe->getRefSuivante(),
                           nouvellePosition - 1,
                           nouvelleInfo);

  }
}
```

Où je reviens à ce que je connais, l'itération !

ALGORITHMES DE PARCOURS ITÉRATIFS

Affichage d'une `ListeChainee` …

Itératif de gauche à droite

- Une méthode publique de **`ListeChainee`**
  - avec un **`Worker`** *(on pourrait s'en passer…)*

+.h

```
// Affichage itératif de cette liste
// parcours de gauche à droite
void afficheGDIter() const;
```

+.cpp

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheGDIter() const {
    cout << "En parcours de gauche à droite, la liste contient ->
";
    // afficheGDIterWorker privée, affiche une liste de Cellules
    afficheGDIterWorker(ptrTete);
    cout << endl;
}
```

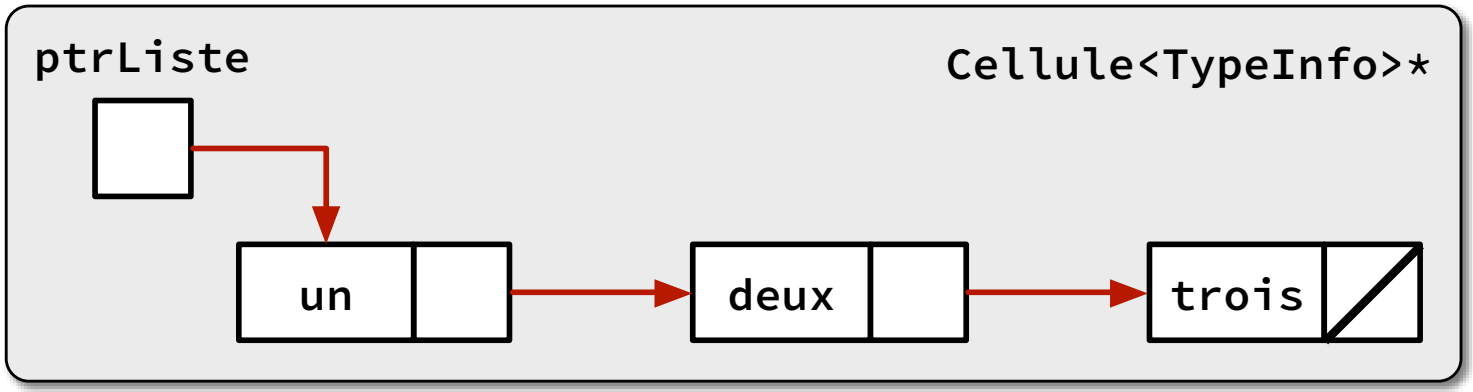
Affichage d'une liste de `Cellules` GD

Itératif

- Le prototype du **`Worker`** privé qui affiche de façon itérative une liste de **`Cellules`** est :

```
void afficheGDIterWorker(Cellule<TypeInfo>* ptrListe) const;
```

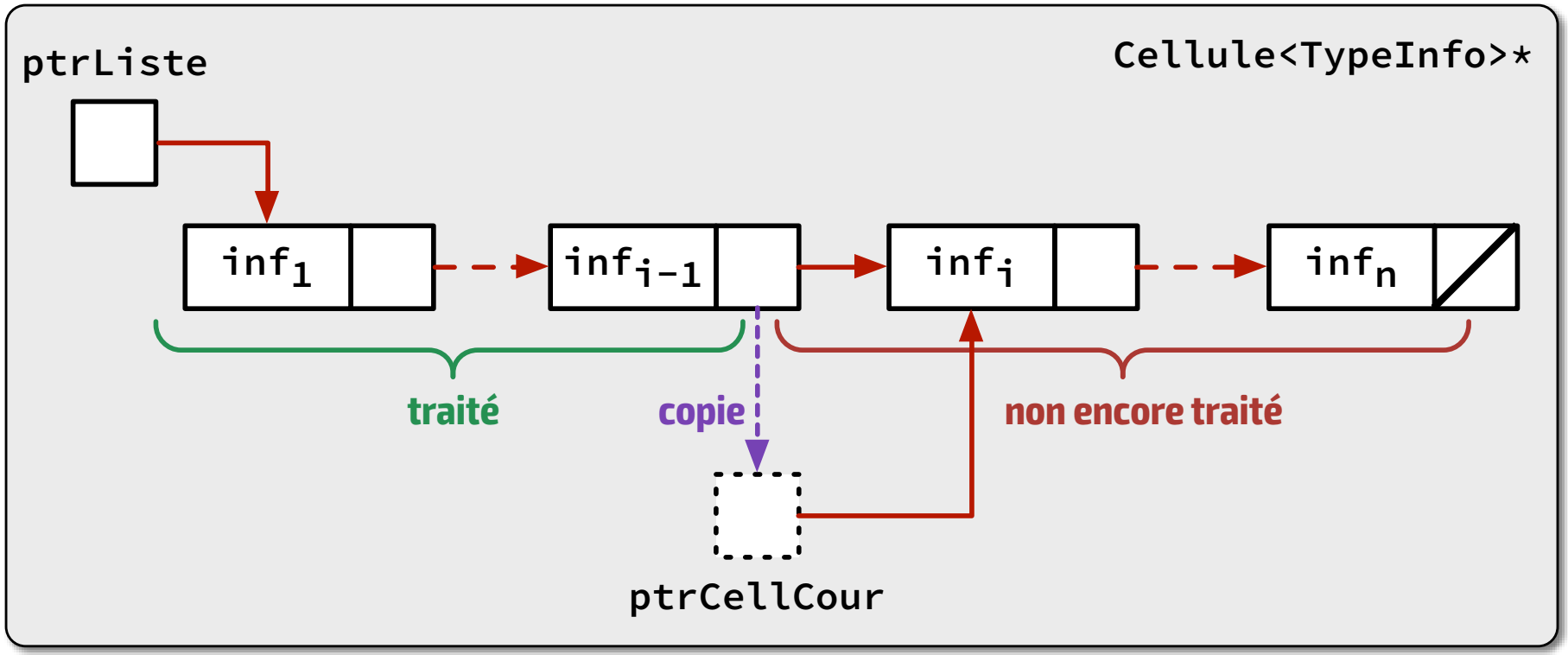
- Le paramètre effectif de la **procédure** sera le pointeur sur la première cellule : **`ptrListe`**



Affichage d'une liste de `Cellules` GD

Itératif

- Même approche qu'en M1102 & M1103
  - on a affiché les **`Cellules`** qui précèdent **`ptrCellCour`**
  - on n'a pas encore traité les **`Cellules`** qui suivent



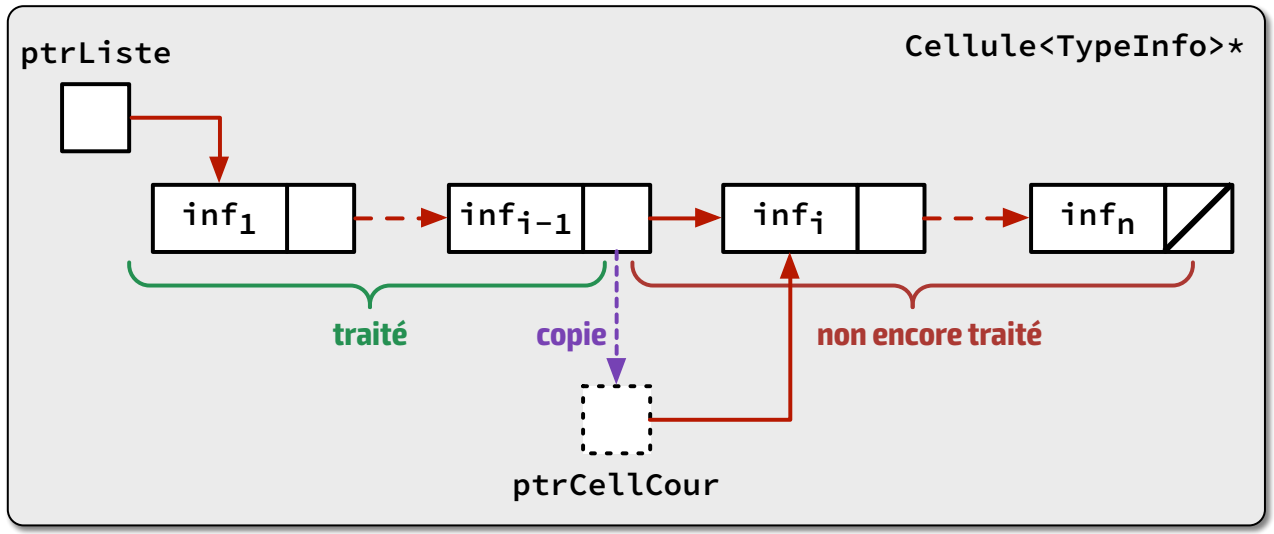
Affichage d'une liste de `Cellules` GD

Itératif

- Raisonnement par récurrence
  - **Hypothèse** : `ptrCellCour`$^+$ non traité
  - `ptrCellCour == nullptr` → {terminé} ✳
  - `ptrCellCour != nullptr` →

```
cout << ptrCellCour ->getInfo() << " ";
ptrCellCour = ptrCellCour->getSuivante(); ➡ H
```

  - **Itération** : `while (ptrCellCour) {…}`
  - **Initialisation** : `ptrCellCour = ptrListe;` ➡ H



Affichage d'une liste de `Cellules` GD Itératif

Procédure privée de `ListeChainee`

+.h

```
private:
 // Affichage itératif de cellules qui se suivent
 // parcours de gauche à droite
 void afficheGDIterWorker(Cellule<TypeInfo>* ptrListe) const;
```

+.cpp

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheGDIterWorker(Cellule<TypeInfo>* ptrListe)
const {
    Cellule<TypeInfo>* ptrCellCour = ptrListe;
    while (ptrCellCour) {
        cout << ptrCellCour->getInfo() << " ";
        ptrCellCour = ptrCellCour->getSuivante();
    }
}
```

Affichage d'une liste de Cellules GD Itératif

- L'écriture de la version itérative de l'affichage de gauche à droite ne présente pas de difficulté majeure
  - parce que dans la version récursive, on ne fait rien (aucune instruction) après l'appel récursif
  - on dit que c'est un **appel récursif terminal**

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheGDWorker(Cellule<TypeInfo>* ptrListe)
const {
    if (ptrListe) {
        cout << ptrListe->getInfo() << " ";
        afficheGDWorker(ptrListe->getSuivante());
    }
    // sinon rien à faire, donc on en fait rien !!!
} // end afficheCelluleGD
```

Affichage d'une liste de `Cellules` DG Itératif

- L'écriture de la version itérative de l'affichage de droite à gauche est plus difficile
  - parce que dans la version récursive, on travaille encore après l'appel récursif (**`cout...`**)
    - dans la version itérative, il faut se rappeler d'où l'on vient
      - il faut pour cela une structure de données auxiliaire (pile)
  - on dit que c'est un **appel récursif non terminal**

```
template<class TypeInfo>
void ListeChainee<TypeInfo>::afficheDGWorker(Cellule<TypeInfo>* ptrListe) const
{
    if (ptrListe) {
        afficheDGWorker(ptrListe->getSuivante());
        cout << ptrListe->getInfo() << " ";
    }
    // sinon rien à faire, donc on en fait rien !!!
} // end afficheCelluleDG
```