



1 - Notion de liste chaînée .....	2
2 - Création d'une liste chaînée .....	2
Définition du type élémentaire .....	3
Création d'une liste vide .....	3
3 - Ajout d'éléments à une liste .....	4
Ajout d'un élément .....	4
Insertion d'une liste dans une autre liste .....	6
4 - Parcours de listes .....	6
5 - Suppression d'éléments d'une liste .....	7
6 - Cas particuliers remarquables .....	8
Listes ordonnées .....	8
Listes circulaires .....	9
Listes superposées .....	10
7 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ? .....	10
8 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ? .....	10
9 - Pré-requis de la Leçon 11 .....	10

L'usage de données organisées en tableaux (cf. Leçon 9) présente deux caractéristiques principales : l'accès aux éléments est direct, et cet avantage se paye en figeant, à un moment donné, le nombre d'éléments du tableau.

On parle d'accès direct pour souligner le fait que la simple connaissance de l'index d'un élément permet d'accéder à cet élément, sans avoir à parcourir le tableau. De ce point de vue, un tableau ressemble plus à un CD audio (où l'on peut accéder directement à n'importe quelle plage dont on connaît la position sur le disque) qu'à une cassette (qu'il faut faire défiler intégralement – éventuellement en vitesse rapide – si l'on souhaite écouter le dernier titre).

La nécessité de figer la taille d'un tableau se manifeste de deux façons différentes, selon s'il s'agit d'un vrai ou d'un faux tableau. Dans le premier cas, la taille doit être choisie au moment de l'écriture du programme, alors que dans le second, cette décision peut être différée jusqu'à l'exécution du programme, ce qui permet d'adapter la taille du tableau aux circonstances rencontrées lors d'une exécution donnée. Il n'en reste pas moins que, une fois le tableau créé, il n'est pas réellement possible d'en modifier la taille<sup>1</sup>.

Fondamentalement, les tableaux sont donc une organisation efficace lorsque le nombre de données varie peu pendant une même exécution du programme, et lorsque le programme est susceptible de devoir accéder à n'importe laquelle de ces données à n'importe quel moment. Lorsque les conditions inverses sont remplies (données en nombre très variable, et accès en ordre prévisible), le choix d'un tableau serait donc maladroit, et l'objet de cette Leçon est précisément de décrire une structure de données qui serait alors plus judicieuse.

## 1 - Notion de liste chaînée

Une liste chaînée est une structure de données dont chaque élément fournit un moyen d'accéder à l'élément suivant.

Cette définition fait immédiatement apparaître l'importance de la notion d'ordre dans une liste chaînée puisque, par sa nature même, le chaînage ne permet de parcourir intégralement une liste qu'en partant du premier élément et en visitant les éléments suivants l'un après l'autre, dans l'ordre où ils s'enchaînent.

La possibilité d'accès direct aux éléments, que nous venons de souligner dans le cas des tableaux, est donc une caractéristique dont les listes chaînées sont dépourvues : elles ne permettent qu'un accès habituellement qualifié de *séquentiel*.

En contrepartie de ces possibilités relativement limitées d'accès aux éléments, l'absence de tout principe organisationnel externe aux éléments permet une grande souplesse lorsqu'il s'agit de modifier la liste. Des opérations telles que

- la modification de l'ordre des éléments ;
  - l'ajout d'un élément à une position quelconque dans la liste ;
  - l'insertion d'une seconde liste à une position quelconque dans la liste ;
  - la suppression d'un élément quelconque ;
  - la suppression d'un nombre quelconque d'éléments consécutifs ;
- soient, nous allons le voir, aussi faciles à programmer que rapidement exécutées.

La modification de l'ordre des éléments d'un tableau (qui se traduit par une succession d'échanges de valeurs entre paires d'éléments), ainsi que les insertions ou suppressions (qui impliquent le déplacement de toutes les valeurs situées après la position où elles interviennent), sont des opérations très coûteuses lorsqu'elles sont appliquées sur des tableaux de grande taille.

## 2 - Création d'une liste chaînée

En C++, le moyen le plus naturel pour rendre un élément capable de fournir un accès à l'élément suivant est certainement de doter chaque élément d'un pointeur sur l'élément suivant. Il faut donc que chaque élément soit capable de conserver non seulement les données qui sont sa raison d'être, mais également une adresse, qui crée la structure de liste.

<sup>1</sup> On peut simuler une telle modification en créant un nouveau tableau ayant la nouvelle taille souhaitée, et en y recopiant les éléments du premier tableau, mais cette opération risque d'être fort longue dans le cas d'un tableau de grande taille.

La première étape est donc la création d'un type de données correspondant à ce cahier des charges, c'est à dire d'une classe comportant au minimum deux variables membres : l'une destinée à stocker les données proprement dites, l'autre permettant d'assurer le chaînage entre les éléments.

#### Définition du type élémentaire

Pour prendre un exemple simple, considérons que les données devant être stockées sont des valeurs numériques<sup>2</sup>. Dans ce cas, le type élémentaire peut initialement être défini ainsi :

```

1 class CElement
2 {
3 public:
4     double valeur;
5     CElement * suivant;
6 };

```

Attention, la variable membre suivant est de type "pointeur sur CElement" et non "CElement". Il n'est en effet pas possible qu'une variable membre ait pour type la classe à la définition de laquelle sa déclaration contribue, et un simple raisonnement sur la place occupée en mémoire par une instance le montre bien : si la classe CElement avait un membre de type int et un membre de type CElement, une variable de type CElement occuperait en mémoire la place occupée par un int, plus la place occupée par un... CElement. L'inanité d'une telle définition est donc patente.

#### Création d'une liste vide

La plus grande des difficultés conceptuelles liées à l'utilisation d'une liste chaînée est peut être due au fait que l'objet "liste" n'a pas d'existence indépendante des éléments qui le composent.

Dans le cas d'un tableau, par exemple, la structure est d'abord créée, puis elle reçoit les valeurs qu'elle est supposée contenir<sup>3</sup>. Une liste, en revanche, naît et grandit (ou diminue) au gré des ajouts (et des suppressions) de données qu'elle subit.

L'expression "liste vide" est donc un peu abusive, puisqu'il ne s'agit pas vraiment d'une liste, mais simplement d'un moyen qui servira à accéder au premier élément lorsque la liste ne sera plus vide. Deux options sont possibles : une variable de type "pointeur sur élément" et de valeur NULL, ou un premier élément "factice", qui ne stocke aucune donnée mais sert de "point d'ancrage" à la liste. Ces deux approches sont à peu près équivalentes, mais comme la première oblige à considérer l'ajout du premier élément comme un cas particulier, nous allons adopter ici la seconde, même si elle est, d'un certain point de vue, un peu moins élégante.

On convient de signaler qu'un élément est le dernier de la liste en donnant une valeur NULL au pointeur qui devrait désigner l'élément suivant. Dans le contexte de notre exemple, nous pourrions donc créer une liste vide en écrivant

```

1 CElement laListe;
2 laListe.suivant = NULL;

```

Une représentation imagée de cette "liste vide" pourrait être :

laListe	
valeur	< sans importance >
suivant	NULL

Bien qu'il s'agisse d'un CElement, la variable laListe ne doit pas être considérée comme un véritable élément de la liste. Le contenu de son membre valeur est donc sans intérêt, alors que son membre suivant peut adopter deux états : il est NULL lorsque la liste est vide, et il contient l'adresse du premier élément dans le cas contraire.

<sup>2</sup> Le nombre et la nature des variables membre qui représentent les données ne changent évidemment rien à la logique et à la nature des opérations de gestion de la structure de liste.

<sup>3</sup> Même si on utilise une liste d'initialisation pour combiner création d'un vrai tableau et attribution des valeurs aux éléments en une même instruction, il reste clair que le tableau est d'abord créé (ce qui implique une attribution de mémoire), et que c'est seulement ensuite que les éléments reçoivent leur valeur.

### 3 - Ajout d'éléments à une liste

La facilité avec laquelle on peut ajouter un élément à une liste est l'une des caractéristiques principales de cette structure de données. Il est même possible d'insérer une autre liste, à une position quelconque dans une liste donnée. Etant donnée la conception de la "liste vide" que nous avons adopté, l'ajout d'un élément (ou d'une liste) se fera toujours après un élément existant déjà : dans le cas du premier "véritable" élément, l'insertion aura simplement lieu après l'élément "factice" qui représente la liste vide. C'est là l'avantage de cette façon de représenter la liste vide : l'ajout du premier élément n'a, techniquement, rien de particulier.

Cet ajout d'un premier élément sur une liste vide reste toutefois *psychologiquement* un peu particulier, en raison du fait que nous connaissons la valeur du membre suivant de l'élément après lequel nous procédons à un ajout : ce pointeur est, par définition, NULL. Plutôt que de raisonner sur ce cas, il est donc préférable d'imaginer que nous opérons sur une liste comportant déjà quelques éléments. L'exemple suivant représente une liste comportant deux éléments, dont le premier contient la valeur 1 et le second la valeur 2 :

laListe	
valeur	< sans importance >
suitant	A1

CElement résidant à l'adresse A1	
valeur	1
suitant	A2

CElement résidant à l'adresse A2	
valeur	2
suitant	NULL

#### Ajout d'un élément

Si, dans cette situation, nous cherchons à ajouter après le premier un nouvel élément contenant la valeur 1.5, nous devons effectuer plusieurs opérations.

1) Il nous faut tout d'abord créer le nouvel élément et stocker dans son membre valeur la donnée qui doit être insérée dans la liste. Nous obtenons alors la situation suivante :

laListe	
valeur	< sans importance >
suitant	A1

CElement résidant à l'adresse A1	
valeur	1
suitant	A2

CElement résidant à l'adresse A2	
valeur	2
suitant	NULL

nouveau CElement	
valeur	1.5
suitant	< inconnu >

Remarquez que, bien que le nouveau CElement existe, il ne fait pour l'instant pas partie de la liste, puisque son adresse n'est indiquée par aucun des éléments de celle-ci.

2) Pour insérer ce nouvel élément dans la liste, il faut lui donner comme successeur l'élément qui était jusqu'à présent le successeur de celui après lequel il va prendre place. En l'occurrence, si le nouvel élément s'insère entre les deux éléments actuels de la liste, son successeur sera l'actuel second élément, ce qui signifie que le membre suivant du nouvel élément doit contenir l'adresse A2.

laListe	
valeur	< sans importance >
suitant	A1

CElement résidant à l'adresse A1	
valeur	1
suitant	A2

CElement résidant à l'adresse A2	
valeur	2
suitant	NULL

nouveau CElement	
valeur	1.5
suitant	A2

La structure de données se trouve à ce moment là dans un état qui ne correspond pas à la définition d'une liste chaînée, puisque deux éléments pointent sur le dernier, alors qu'aucun ne pointe encore sur le nouveau.

3) La cohérence de la liste doit finalement être rétablie en indiquant que le nouvel élément est désormais le successeur de celui après lequel il prend place.

laListe	
valeur	< sans importance >
suitant	A1

CElement résidant à l'adresse A1	
valeur	1
suitant	adresse du nouveau

CElement résidant à l'adresse A2	
valeur	2
suitant	NULL

nouveau CElement	
valeur	1.5
suitant	A2

Une fois la logique des opérations comprise, l'écriture d'une fonction les effectuant ne pose pas de réel problème. Le seul point important à souligner est que cette fonction doit être en mesure d'accéder à *l'élément après lequel* l'insertion va avoir lieu. En effet, le contenu du membre suivant de cet élément doit pouvoir être utilisé lors de la deuxième étape et modifié lors de la troisième.

On voit que la chronologie des opérations est ici cruciale : si le membre suivant de l'élément précédant l'insertion recevait d'abord l'adresse du nouveau (opération 3), il ne serait ensuite plus possible de retrouver l'adresse du dernier pour la stocker dans le membre suivant du nouveau (opération 2).

Le meilleur moyen pour permettre à la fonction d'accéder à l'élément après lequel l'insertion doit avoir lieu est de faire de cette fonction un **membre** de la classe `CElement`, et de l'appeler au titre de cet élément. Cette fonction ne reçoit donc qu'un seul paramètre : un double qui recevra la valeur devant être stockée dans l'élément créé.

```

1 void CElement::ajouteElement(double valeurAStocker)
2 {
3   CElement * leNouveau = new CElement;    // 1 - on crée le nouvel élément
4   if (leNouveau == NULL)
5       //prendre les mesures qui s'imposent
6       leNouveau->valeur = valeurAStocker; //1bis - on stocke la valeur
7       leNouveau->suivant = suivant;      //2 - on désigne son successeur
8       suivant = leNouveau;              //3 - leNouveau successeur
9   }

```

Si `laListe` est une variable, il faut souligner que c'est la seule qui soit impliquée dans la définition de la liste. Les véritables éléments sont en effet créés par allocation dynamique (ce sont donc des objets anonymes et non des variables) et ils sont utilisés pour stocker les uns les adresses des autres, ce qui rend inutile le recours permanent à des variables de types "pointeur sur `CElement`".

Lorsque l'instance utilisée pour invoquer `ajouteElement()` se trouve être le pseudo-élément figurant une liste vide, la valeur qui est transférée (7) dans le membre suivant du nouvel élément est celle contenue dans le membre suivant du pseudo élément, c'est à dire `NULL`. Aucune opération spécifique n'est donc nécessaire pour indiquer que l'élément nouvellement créé se trouve en fin de liste, et nous constatons, comme annoncé, que la création du premier élément d'une liste n'exige pas un traitement particulier.

La fonction `ajouteElement()` étant ainsi définie, la situation qui nous a servi d'exemple pourrait être recréée par exécution du fragment de code suivant :

```

// création de la liste vide
1 CElement laListe;
2 laListe.suivant = NULL;

//création du premier élément
3 laListe.ajouteElement(1);

//création du dernier élément
4 CElement *lePremier = laListe.suivant;
5 lePremier->ajouteElement(2);

//insertion d'un nouvel élément entre les deux autres
6 lePremier->ajouteElement(1.5);

```

L'utilisation (4) d'un "pointeur sur `CElement`" n'est nécessaire que pour respecter les ordres de création et d'enchaînement des éléments correspondant à notre exemple. Si ces ordres sont sans importance, l'ajout d'un élément à la liste se fera très facilement "par la tête", en écrivant simplement quelque chose comme :

```
laListe.ajouteElement(36);
```

Les listes ainsi créées ont la particularité d'être organisées dans l'ordre chronologique inverse : le premier élément de la liste est celui qui a été créé en dernier.

### Insertion d'une liste dans une autre liste

Cette opération revient à fusionner les deux listes concernées et se résume, quel que soit le nombre d'éléments impliqués, à la modification des valeurs contenues dans deux variables membre. Les deux listes disjointes peuvent en effet être représentées ainsi :

laListeA	CElement en A1	CElement en A2	
valeur	valeur 1	valeur 2	
suivant	suivant A2	suivant NULL	

  

laListeB	CElement en B1	CElement en B2	CElement en B3
valeur	valeur 1	valeur 2	valeur 3
suivant	suivant B2	suivant B3	suivant NULL

et l'insertion de la seconde liste entre les deux éléments de la première se traduira par :

laListeA	CElement en A1	CElement en A2	
valeur	valeur 1	valeur 2	
suivant	suivant B1	suivant NULL	

  

laListeB	CElement en B1	CElement en B2	CElement en B3
valeur	valeur 1	valeur 2	valeur 3
suivant	suivant B2	suivant B3	suivant A2

L'insertion d'une liste après un élément donné implique donc trois pointeurs :

- l'adresse (A1, dans cet exemple) de l'élément après lequel l'insertion doit avoir lieu, (parce que le membre suivant de cet élément doit être modifié)
- l'adresse (B1, dans cet exemple) du premier élément de la liste à insérer, (parce que c'est cette adresse qui doit être rangée dans le membre suivant de l'élément après lequel l'insertion doit avoir lieu)
- l'adresse (B3, dans cet exemple) du dernier élément de la liste à insérer. (parce que le membre suivant de cet élément doit recevoir l'adresse du successeur de l'élément après lequel l'insertion doit avoir lieu)

Si elle est invoquée au titre de l'élément après lequel la liste doit être insérée, une fonction pratiquant cette opération peut donc être dotée de deux arguments de type "pointeur sur CElement" :

```

1 void CElement::insereListe (CElement *debutAjout, CElement *finAjout)
2 {
3   finAjout->suivant = suivant;
4   suivant = debutAjout;
5 }

```

Selon les circonstances, il peut également être nécessaire de signaler que la liste dont les éléments ont été "injectés" dans l'autre liste est désormais vide.

Remarquons que la fonction `insereListe()` pourrait se contenter d'un seul argument. En effet, le dernier élément de la liste à ajouter peut très bien être retrouvé, si l'on dispose de l'adresse du premier : il suffit de parcourir la liste.

## 4 - Parcours de listes

L'accès aux éléments d'une liste repose sur un principe simple : lorsqu'un pointeur contient l'adresse d'un élément, il peut très facilement prendre pour valeur l'adresse de l'élément suivant. Ainsi, si `ptr` contient l'adresse d'un élément d'une liste, après exécution de

```
ptr = ptr->suivant;
```

deux cas peuvent se produire :

- soit l'élément précédemment désigné par `ptr` était le dernier de la liste, et `ptr` contient donc maintenant la valeur NULL,
- soit `ptr` contient maintenant l'adresse du successeur dans la liste de l'élément qu'il désignait précédemment.

La mise en œuvre de ce principe est assez simple, à condition de bien comprendre qu'il exige l'utilisation d'une variable auxiliaire de type "pointeur sur CElement", qu'il convient d'initialiser avec l'adresse de l'élément à partir duquel doit s'effectuer le parcours de la liste. Le fragment de

code suivant crée une liste de 500 éléments, pour le simple plaisir de la parcourir ensuite exhaustivement (en annulant au passage les valeurs contenues dans les éléments) :

```

1 //création d'une liste vide
  CElement laListe;
2 laListe.suivant = NULL;

  //ajout de 500 éléments
3 int i;
4 for (i = 0 ; i < 500 ; i = i + 1);
5   laListe.ajouteElement(i); //laListe contient les nombres de 499 à 0

  //remise à zéro de tous les éléments
6 CElement * ptr = laListe.suivant;
7 while(ptr != NULL)
8   {
9     ptr->valeur = 0;
10    ptr = ptr->suivant; //la "formule magique" du parcours de liste...
11  }

```

Cette technique de parcours constitue la méthode normale pour accéder aux éléments d'une liste. Dans certains cas, il peut être envisageable de "chaîner" les opérateurs de sélection pour simuler un accès direct à un élément de la liste. On pourrait, par exemple, modifier la valeur contenue dans les trois premiers éléments d'une liste en écrivant :

```

laListe.suivant->valeur = 1; //modifie le contenu du 1er élément
laListe.suivant->suivant->valeur = 2; //modifie le contenu du 2o élément
laListe.suivant->suivant->suivant->valeur = 3; //modifie le contenu du 3o élément

```

Les inconvénients de cette approche sautent toutefois aux yeux : elle devient vite encombrante et peu lisible, elle est très rigide (une ligne de code donnée ne peut qu'accéder toujours au même rang dans la liste) et elle ne peut être employée que dans des contextes où l'on est certain que la longueur de la liste est suffisante pour que l'évaluation de l'expression ne conduise pas à déréférencer le pointeur NULL final.

## 5 - Suppression d'éléments d'une liste

La création d'éléments fait appel à une allocation dynamique de mémoire. Lorsque certains de ces éléments (ou même tous ceux présents dans la liste) ne sont plus utiles, il est donc nécessaire de restituer cette mémoire au système, sous peine de créer une "fuite de mémoire".

La logique des opérations est assez simple :

- l'élément précédant celui qui doit être supprimé doit adopter comme successeur le **successeur de l'élément qui doit être supprimé** (ce qui retire l'élément à supprimer de la liste),
- l'élément visé peut ensuite être **effectivement supprimé**.

Etant donné que l'élément qui précède celui qui va être supprimé doit être modifié, c'est au titre de celui-ci qu'il convient d'appeler la fonction chargée d'exécuter cette basse besogne. Avant d'entreprendre la suppression de l'élément suivant, la fonction devra donc vérifier que celui-ci existe réellement (ce qui n'est pas le cas si l'adresse reçue se trouve, par erreur, être celle du dernier élément de la liste). Nous écrirons donc :

```

1 void CElement::supprimeSuivant()
2 {
3   CElement * laVictime = suivant;
4   if(laVictime == NULL)
5     return;
6   suivant = laVictime->suivant;
7   delete laVictime;
8 }

```

Une erreur fréquemment commise dans ce genre de situation est de céder trop rapidement à la tentation de se passer de la variable locale baptisée `laVictime` dans la fonction ci-dessus. La fonction suivante est, certes, un peu plus brève que la précédente, mais elle est *fausse*.

```

1 void CElement::supprimeSuivant() //FONCTION FAUSSE ET DANGEREUSE
2 {
3   if(suivant == NULL)
4     return;
5   delete suivant;
6   suivant = suivant->suivant; //HORREUR !
7 }

```

En effet, après l'application de l'opérateur `delete` au pointeur `suivant`, celui-ci n'est plus valide. La dernière instruction de la fonction commet donc une faute grave en le déréférençant. Cette erreur est d'autant plus dangereuse que, dans bien des cas, le code semble fonctionner sans problème. En effet, ce n'est pas parce que la zone de mémoire n'est plus réservée au stockage d'un `CElement` qu'elle est forcément utilisée tout de suite pour stocker autre chose. Le déréférencement fautif `suivant` immédiatement l'utilisation de `delete`, il risque fort d'utiliser une information correcte *par hasard*. Ceci revient à dire que le programme fonctionne *par hasard*, avec une probabilité élevée de succès. En d'autres termes, tout ira bien jusqu'à ce que le calcul effectué soit *vraiment* important. Ce jour là, tout ira mal.

La destruction de tous les éléments d'une liste peut donc être obtenue en écrivant simplement :

```

1 while(laListe.suivant != NULL) //tant que la liste n'est pas vide...
2   laListe.supprimeSuivant(); //...supprime son premier élément

```

## 6 - Cas particuliers remarquables

Du fait de leur grande souplesse d'utilisation, les listes chaînées peuvent répondre à des besoins très divers, dans de nombreuses situations de programmation. Trois types particuliers de listes chaînées me semblent mériter d'être signalés ici.

### Listes ordonnées

Face à un ensemble important de données, une question qui se pose très souvent est de déterminer si une valeur particulière est présente. Pour éviter d'avoir à examiner toutes les données avant de pouvoir répondre par la négative, on peut procéder à un tri, ce qui permet ensuite d'exclure rapidement de nombreuses données de l'ensemble à explorer.

Imaginons un ensemble de valeurs numériques triées en ordre croissant. Choisissons un des éléments de cet ensemble et observons sa valeur. Si la valeur recherchée est plus petite que celle de notre élément, nous sommes certains qu'elle ne figure dans aucun des éléments placés après l'élément observé. Inversement, si la valeur recherchée est plus grande que celle de l'élément observé, nous sommes certains qu'elle ne figure pas avant l'élément observé. Chaque observation permet donc d'éliminer une des deux hypothèses (avant ou après), et on peut montrer que cette méthode connaît une efficacité maximum lorsque l'élément observé se situe au centre de l'ensemble des éléments susceptibles de contenir la valeur recherchée. Chaque observation élimine alors l'une des deux moitiés de cet ensemble, ce qui explique que cette méthode est connue sous le nom de recherche dichotomique. Les tableaux se prêtent assez bien à la mise en œuvre de cette technique.

L'obligation de parcourir les éléments d'une liste dans l'ordre où ils se désignent les uns les autres s'oppose à l'utilisation d'algorithmes aussi efficaces que la recherche dichotomique, mais la recherche d'une valeur dans une liste est quand même plus rapide si celle-ci est triée : la recherche peut être abandonnée dès que l'on rencontre un élément excédant la valeur recherchée (si l'ordre est croissant) ou plus petit que celle-ci (si l'ordre est décroissant).

Si la recherche dans une liste n'est pas des plus efficace, cette structure de données se prête en revanche très bien au "maintien de l'ordre" lorsque les valeurs qu'elle contient sont modifiées. En effet, déplacer un élément dans une liste ne nécessite la modification que de trois pointeurs (un pour "ressouder" la liste à l'endroit abandonné par l'élément déplacé, et deux pour effectuer la "greffe" à la nouvelle position<sup>4</sup>). Par contraste, modifier une seule valeur dans un tableau peut nécessiter des milliers de modifications pour conserver l'ordre, et ces modifications risquent, en plus, d'impliquer des objets considérablement plus volumineux que des pointeurs... Selon les fréquences relatives des opérations de recherche et des modifications de valeurs, une approche basée sur une liste peut donc s'avérer finalement plus efficace qu'une approche basée sur un tableau, même si celle-ci peut exploiter la recherche dichotomique.

<sup>4</sup> Ces deux opérations correspondent respectivement à la suppression d'un élément et à l'insertion d'un élément dans une liste, opérations dont nous avons déjà étudié la logique.

## Listes circulaires

La structure de liste présente une particularité originale : si l'un de ses éléments désigne comme successeur un élément qui figure déjà plus haut dans la liste, la liste est dépourvue de fin. Cette possibilité peut être exploitée pour obtenir facilement un fonctionnement cyclique. Une liste circulaire de sept éléments dont chacun contiendrait le nom d'un des jours de la semaine, par exemple, permet d'obtenir toujours de la même façon le nom du jour suivant un jour donné : si aujourd'hui est un pointeur contenant l'adresse de l'élément qui contient le nom du jour courant, l'instruction suivante pourra être exécutée à minuit :

```
aujourd'hui = aujourd'hui->suivant;
```

L'avantage de cette technique par rapport à l'usage d'un tableau est qu'elle permet une meilleure lisibilité du code : le tableau exige un index dont le calcul doit assurer le retour au début du tableau en fin de semaine. La liste permet la circularité sans index ni calcul, et se prête en outre fort bien aux cas où le nombre de valeurs figurant dans le cycle est susceptible de varier (le calcul de l'index désignant le bon élément d'un tableau devient vite assez pénible dans ce genre de situation).

Si l'on dispose d'un type élémentaire capable de stocker des chaînes de caractères, on peut créer ainsi la liste circulaire nécessaire :

```
// création de la liste vide
1 CElement lesJours;
2 lesJours.suivant = NULL;

//création des éléments
3 lesJours.ajouteElement("Dimanche");
4 CElement * dimanche = lesJours.suivant; //on aura besoin de cette adresse
5 lesJours.ajouteElement("Samedi");
6 lesJours.ajouteElement("Vendredi");
7 lesJours.ajouteElement("Jeudi");
8 lesJours.ajouteElement("Mercredi");
9 lesJours.ajouteElement("Mardi");
10 lesJours.ajouteElement("Lundi");
11 CElement * lundi = lesJours.suivant;

//on rend la liste circulaire
12 dimanche->suivant = lundi;
```

Une fois rendue circulaire, la liste ne peut évidemment plus être parcourue par une boucle dont l'arrêt repose sur la rencontre d'un membre suivant de valeur nulle. A ce test, on peut substituer une comparaison avec l'adresse du premier élément de la liste, et le fragment de code suivant compte ainsi le nombre d'éléments présents dans une liste circulaire :

```
//dénombrer des éléments d'une liste circulaire
1 CElement * ptr = laListe.suivant;
2 int nbElements = 0;
3 if (ptr != NULL)
4     do {
5         nbElements = nbElements + 1;
6         ptr = ptr->suivant;
7     } while (ptr != laListe.suivant);
```

Si `laListe` n'est pas circulaire, ce fragment de code provoquera une erreur d'exécution en déréférençant (Ⓞ) le membre suivant du dernier élément. La valeur de `nbElements` sera bien, à cet instant, le nombre recherché, mais c'est là une bien maigre consolation...

La destruction d'une liste circulaire exige également des précautions particulières qui constituent sans doute l'essentiel du coût impliqué par l'utilisation d'une telle structure :

```
//destruction des éléments d'une liste circulaire
1 CElement *premier = laListe.suivant;
2 if (premier != NULL)
3     do{
4         premier->supprimeSuivant();
5     } while (premier->suivant != premier);
6 laListe.supprimeSuivant();
```

**Listes superposées**

Rien n'oblige le type élémentaire à ne comporter qu'un seul pointeur. Si deux pointeurs sont disponibles, chaque élément peut désigner non seulement l'élément suivant, mais aussi l'élément précédent. La liste est alors dite "doublement chaînée", et elle se trouve simultanément triée dans l'ordre croissant et dans l'ordre décroissant, selon la chaîne de pointeurs choisie pour la parcourir.

Cette caractéristique n'est pas très spectaculaire, puisque n'importe quel tableau trié en ordre croissant est également trié en ordre décroissant, si on le parcourt en faisant décroître l'index au lieu de le faire croître.

Bien entendu, les divers pointeurs figurant dans le type élémentaire d'une liste (et il peut y en avoir plus de deux) ne sont nullement tenus de correspondre à un ordonnancement faisant intervenir la même dimension. S'il est organisé en liste, l'inventaire d'une quincaillerie peut très bien, par exemple, être à la fois trié dans l'ordre alphabétique des noms des articles (100 arrosoirs, 25 bouchons, 30 clous, 25 décapsuleurs...) et dans l'ordre croissant des nombres d'exemplaires (25 décapsuleurs, 25 bouchons, 30 clous, 100 arrosoirs). Il faut dans ce cas que le type élémentaire comporte deux pointeurs, nommés par exemple `nomSuivant` et `effectifSuivant`, et bien entendu, que les adresses correctes aient été placées dans les membres de chacun des éléments.

Comme l'ordre des éléments d'un tableau est déterminé par l'emplacement qu'ils occupent en mémoire, il est clair qu'un tableau ne peut pas être à la fois dans deux ordres différents. Si plusieurs ordres doivent être utilisés, il est nécessaire de retrier fréquemment le tableau (ce qui peut être très long) ou d'utiliser plusieurs tableaux, ce qui complique considérablement les opérations de mises à jour lorsqu'une donnée doit être modifiée (parce qu'on a vendu un arrosoir, par exemple). Dans le cas d'une liste, on peut maintenir simultanément autant d'ordonnements que nécessaire, sans avoir à dupliquer la description des éléments, ce qui diminue à la fois l'espace mémoire nécessaire et les risques d'erreurs de programmation.

**7 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ?**

- 1) Une liste chaînée est obtenue en rendant chaque élément capable de fournir un moyen d'accéder à l'élément suivant.
- 2) Un moyen simple pour créer une liste chaînée est de définir une classe comportant un membre de type "pointeur sur instance de la classe". Cette classe sert ensuite de type élémentaire à la liste.
- 3) On peut créer une "liste vide" en instanciant le type élémentaire.
- 4) Les opérations d'ajout et de suppression dans une liste sont simples et rapides, même lorsqu'elles doivent être effectuées en respectant un tri préalable de la liste.
- 5) L'accès à un élément d'une liste implique le parcours de la liste élément par élément, ce qui peut être long.
- 6) Le ou les pointeur(s) figurant dans le type élémentaire permet(tent) un grand nombre de fantaisies dont les plus classiques sont la circularité et la superposition de listes.

**8 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?**

Tous les bons manuels de programmation traitent cette question. Le choix est donc vaste.

**9 - Pré-requis de la Leçon 11**

La Leçon 11 fait partie du module INF Z23. Il vous faudra donc satisfaire aux modalités d'examen d'INF Z13 avant d'y avoir droit !