Une version sensiblement plus explicite est disponible en page 48 de l'ouvrage *C++ par la pratique*.

Exercice 0 : listes chaînées (structures de données, pointeurs, niveau 0)

On veut écrire le programme `listeschainees.cc` qui implémente une version «à nous» des listes chaînées.
Cet exercice décrit pas à pas les différentes étapes pour y parvenir.

Pour une utilisation réelle concrète des listes chaînées, on emploierait les `forward_list` ou les `list` du standard, qui seront présentées en fin de second semestre. Le seul but du présent exercice est de prendre les listes chaînées comme thème pour prétexte à programmer avec des pointeurs...

Une liste chaînée est une structure de donnée où l'on peut accéder

- directement au premier élément ;
- à un élément quelconque (autre que le premier) *uniquement* au travers de l'élément précédent.

comme les images sur une bobine de film ou les catelles dans un corridor (de une catelle de large).

Une représentation d'une telle liste chaînée se fait à base d'éléments chaînés les uns aux autres ; un élément (= un chaînon) étant consitué de sa valeur et d'un *pointeur* sur l'élément suivant.

1. Ouvrez le fichier (vide) `listeschainees.cc` dans votre éditeur favori.
2. Préparez la «coquille vide» de base accueillant votre programme :

```
#include <iostream>
using namespace std;

int main()
{
    return 0;
}
```

3. Décidons d'implémenter une liste chaînée dynamique d'éléments contenant des valeurs de type `type_el`.

   Pour rendre la chose facilement modifiable, définissons ce type, par exemple comme type entier :

```
#include <iostream>
using namespace std;
typedef int type_el;
int main()
{
    return 0;
}
```

4. Définissons ensuite le type `ListeChainee`, représentant une « liste chaînée », en s'appuyant sur la constatation : une liste chaînée est une suite d'éléments contenant chacun une valeur et la liste chaînée des valeurs suivantes.

   Une liste chaînée sera donc naturellement codée par une **structure** :

```
struct ListeChainee {
   type_el valeur;
   ListeChainee suite;
};
```

   Mais cette façon de faire n'est pas correcte (essayez de compiler, vous verrez). En effet, la définition ci-dessus est **récursive** !

   Si la récursion ne pose aucun problème pour les fonctions (car dynamique), elle est impossible pour les définition des structures de données car conduit à une structure infinie (une chaîne contient une chaîne qui contient une chaîne qui...).

   Il faut donc raffiner la définition ci-dessus.

   Un bon moyen est de ne pas inclure la liste chaînée représentant la suite dans la définition d'une liste chaînée, mais simplement de pointer vers elle (la suite sera ainsi **extérieure** à la liste chaînée principale, évitant les

enchâssements infinis). On utilise pour cela un **pointeur**.

Mais quel pointeur ?
Avant C++11 la question ne se posait pas : on ne disposait que des «pointeurs à la C».
Cela reste certainement la meilleure solution à un niveau avancé, mais les nombreuses modifications apportées à la nouvelle norme 2011 de C++ allant dans le sens d'avoir moins besoin des «pointeurs à la C» au niveau programmeur, je propose d'essayer ici une autre solution et d'utiliser les «smart pointers» :

```
    #include <memory>
...
    struct ListeChainee {
       type_el valeur;
       unique_ptr<ListeChainee> suite;
    };
```

Mais en faisant cela, on voit bien qu'une liste chaînée est en fait un pointeur sur un tel élément et non pas cet élément lui-même (c'est un peu la même erreur que de confondre un ensemble réduit à un élément et cet élément unique, un élément de la structure *ListeChainee* est en fait un *maillon* de la chaîne, et pas la chaîne elle-même). Avec une telle définition, on ne pourrait d'ailleurs pas définir de `ListeChainee` vide !!

D'où la définition plus correcte :

```
typedef unique_ptr<Element> ListeChainee;
struct Element {
   type_el valeur;
   ListeChainee suite;
};
```

Cette syntaxe nous pose un dernier problème : dans la première ligne (`typedef unique_ptr<Element> ListeChainee;`) le type *Element* n'a pas encore été défini. Il faut le mettre **avant**.

Mais si on le met avant, la définition de `ListeChainee suite;` ne sera pas possible car, alors, ce sera `ListeChainee` qui n'aurait pas été défini à ce stade... Il faut donc mettre la définition de `ListeChainee` **avant** celle de `Element`....

Comment se sortir de ce dilemme ?

En indiquant au préalable que nous allons définir une structure `Element`, mais sans en donner le détail (ce qui nécessiterait la connaissance de `ListeChainee`).
Cela se fait de la façon suivante :

```
struct Element; // ici on dit au compilateur qu'Element existera par la suite.
typedef unique_ptr<Element> ListeChainee;
struct Element {
   type_el valeur;
   ListeChainee suite;
};
```

5. Notre structure de données étant définie, il ne nous reste plus qu'à (prototyper et) définir les fonctions de manipulation associées.

   Commençons par l'insertion d'un élément en tête de liste.

   Le prototype est assez naturel : il donner la liste où insérer et (la valeur de) l'élément à insérer.

   La liste en tête de laquelle on souhaite insérer allant être modifiée, il faut la passer par référence. L'élément à insérer n'est par contre pas modifié et peut être passé par valeur (même si pour une question de performance, une référence constante serait préférable).

   On a donc :

   ```
   void insere(ListeChainee& liste, type_el a_inserer);
   ```

   Pour ce qui est de la définition, nous allons effectuer les opération suivantes :

   1. créer le nouvel élément à insérer ;
   2. y mettre la valeur voulue ;
   3. faire pointer la suite de ce nouvel élément vers l'ancienne liste ;
   4. faire pointer la liste donnée sur ce nouvel élément.

Il ne reste plus qu'à l'écrire :

```
void insere(ListeChainee& existant, type_el a_inserer)
{
  // Crée le nouvel élément (en tant que liste chaînée ici)
  ListeChainee nouvelle_liste(new Element);

  // y met la valeur
  (*nouvelle_liste).valeur = a_inserer; // On peut aussi écrire nouvelle_liste->va

  // et connecte l'ancienne liste derrière
  nouvelle_liste->suite = existant;

  // puis remplace l'existant par cette nouvelle liste
  existant = nouvelle_liste;
}
```

ATTENTION ! Le code ci-dessus n'est pas encore totalement correct. Mais je voudrais ici faire déjà quelques remarques :

- On crée un nouvel élément. Mais il faut absolument que la portée ce nouvel élément soit plus grande que la fonction `insere`, puisque cet élément fera ensuite partie de la liste et pourra donc être utilisé ailleurs que dans la fonction `insere`.

  On **ne** peut donc **pas** utiliser une variable locale à la fonction `insere`. Il est **impératif** d'utiliser ici l'*allocation dynamique*, et donc un pointeur, et de créer cette place en appelant la fonction `new` (cas d'utilisation numéro 3 des pointeurs comme vu en cours).

- Remarquez la notation `->` pratique pour accéder aux champs d'une structure donnée par un pointeur : `X->Y` est exactement la même chose que `(*X).Y`.

Pour résumer, à ce stade vous devriez avoir un code qui ressemble à ceci :

```
#include <iostream>
#include <memory>  // pour unique_prt
using namespace std;

// ===== DÉFINITION DES TYPES =====
typedef int type_el; // type des éléments de la liste

struct Element; // un "maillon" de la liste chaînée
typedef unique_ptr<Element> ListeChainee;

struct Element {
  type_el valeur;     // contenu de l'élément courant
  ListeChainee suite; // la suite de la liste chaînée
};

// ====== PROTOYTPES =====
// insère un nouvel élément dans la liste
void insere(Element& existant, type_el a_inserer);

// ----------------------------------------------------------------------
int main()
{
  return 0;
}

/* ********************************************************************* *
 * Insère un nouvel élément en tête d'une liste.                         *
 * Entrée : l'élément de la liste où insérer et l'élément à insérer      *
 * ********************************************************************* */
void insere(ListeChainee& existant, type_el a_inserer)
{
  ListeChainee nouvelle_liste(new Element);
  nouvelle_liste->valeur = a_inserer;
  nouvelle_liste->suite = existant;
  existant = nouvelle_liste;
}
```

Si vous essayez de compiler (faites le), vous aurez des messages d'erreur. Il y en effet un petit problème ici, avancé et très spécifique (il n'est donc pas fondamental de le comprendre en profondeur) :

- Ce problème vient de ce que `ListeChainee` est un **`UNIQUE`**`_ptr`, c'est-à-dire qu'il ne peut y avoir qu'**UN SEUL** pointeur pointant sur le même endroit.

  Or, lorsque l'on fait

  ```
  nouvelle_liste->suite = existant;
  ```

  on a *deux* pointeurs qui pointent au même endroit : `existant` et `nouvelle_liste->suite`. C'est interdit par les *unique*_ptr. En clair, on ne peut pas faire de *copie* d'`unique_ptr`.

  Que faire ?

  Il existe en C++11 une solution alternative à la copie : le déplacement (*move*). C'est ce qu'il faut utiliser ici (et c'est pour cela que les `unique_ptr` ne pouvaient pas exister avant C++11 !).

  On corrige donc :

  ```
  nouvelle_liste->suite = move(existant);
  ```

  Il faut bien sûr faire de même à la fin lorsque l'on met `nouvelle_liste` dans `existant` :

  ```
  existant = move(nouvelle_liste);
  ```

Le code de la fonction `insere` est donc au final :

```
void insere(ListeChainee& existant, type_el a_inserer)
{
  ListeChainee nouvelle_liste(new Element);
  nouvelle_liste->valeur = a_inserer;
  nouvelle_liste->suite = move(existant);
  existant = move(nouvelle_liste);
}
```

Compilez. Il ne devrait cette fois plus y avoir d'erreur.

6. Passons à l'insertion d'un élément *après* un élément. Ceci est très similaire à l'insertion en tête d'une liste : il suffit d'insérer en tête de la liste de ses suivants :

```
/* ********************************************************************* *
 * Insere un nouvel élément après un élément donné.                      *
 * Entrée : La liste où insérer et l'élément à insérer.                  *
 * ********************************************************************* */
void insere(Element& existant, type_el a_inserer)
{
    insere(existant.suite, a_inserer);
}
```

7. Passons maintenant à la suppression d'un élément de la liste.
   Il suffit de savoir supprimer le premier élément d'une liste pour pouvoir supprimer n'importe quel élément.

   La suppression du premier élément d'une liste ne pose aucun problème majeur. Il faut juste ne pas supprimer un élément qui n'existe pas, c.-à-d. qu'il ne faut rien faire sur une liste vide...

```
void supprime(ListeChainee& l)
{
  if (not est_vide(l)) {
    l = move(l->suite);
  }
}
```

   Le `move` étant encore une fois lié aux spécificités des `unique_ptr`.

8. Le calcul de la longueur présenté en cours ne devrait pas présenter de difficulté (ne pas oublier la condition d'arrêt). Notez simplement qu'on pourrait aussi imaginer le coder avec une boucle `for` si l'on savait itérer sur des `unique_ptr`, mais ceci est vraiment au delà de ce cours.

```
size_t taille(const ListeChainee& l)
{
    if (est_vide(l)) return 0;
    else             return 1 + taille(l->suite);
```

```
}
```

9. Pour terminer, il nous faut coder le test si une liste est vide :

```
/* ********************************************************************** *
 * Test si la liste chaînée est vide ou non                               *
 * Entrée : La liste                                                      *
 * ********************************************************************** */
bool est_vide(const ListeChainee& l)
{
  return (l == nullptr);
}
```

Voici pour finir un programme complet illustrant les listes chaînées. Pour information, je donne aussi **ci-dessous une version utilisant les `forward_list` du standard (C++11)** , comme vous devriez les utiliser si vous avez besoin de listes chaînées.

```
#include <iostream>
#include <memory> // pour unique_prt
#include <utility> // pour move
using namespace std;

// ===== DÉFINITION DES TYPES =====
typedef int type_el; // type des éléments de la liste

struct Element; // un "maillon" de la liste chainée
typedef unique_ptr<Element> ListeChainee;

struct Element {
  type_el valeur;     // contenu de l'élement courant
  ListeChainee suite; // la suite de la liste chaînée
};

// ====== PROTOYTPES =====

// insere un nouvel élément dans la liste
void insere(ListeChainee& l, type_el e);

// insere un nouvel élément après un élément d'une liste
void insere(Element& existant, type_el a_inserer);

// supprime le premier élément de la liste
void supprime(ListeChainee& l);

// supprime l'élément suivant l'élément donné
void supprime(Element& e) { supprime(e.suite); }

// détermine si la liste est vide
bool est_vide(const ListeChainee& l) { return (l == nullptr); }

// calcule la longueur de la liste
unsigned int taille(const ListeChainee& l);

// affiche le contenu d'une liste
void affiche(const ListeChainee& l);

//-----------------------------------------------------------------------
int main()
{
  ListeChainee maliste;
  type_el une_valeur = 3;

  insere(maliste, une_valeur);
  insere(maliste, 2);
  insere(maliste, 1);
  insere(maliste, -1);
  insere(maliste, 0);

  affiche(maliste);
```

```
  supprime(maliste);
  affiche(maliste);

  supprime(maliste->suite->suite);
  affiche(maliste);

  return 0;
}

/* ********************************************************************** *
 * Insere un nouvel élément en tête de liste.                             *
 * Entrée : La liste où insérer et l'élément à insérer.                   *
 * ********************************************************************** */
void insere(ListeChainee& existant, type_el a_inserer)
{
  ListeChainee nouvelle_liste(new Element);
  nouvelle_liste->valeur = a_inserer;
  nouvelle_liste->suite = move(existant);
  existant = move(nouvelle_liste);
}

/* ********************************************************************** *
 * Insère un nouvel élément après un élément donné.                       *
 * Entrée : l'élément de la liste où insérer et l'élément à insérer       *
 * ********************************************************************** */
void insere(Element& existant, type_el a_inserer)
{
    insere(existant.suite, a_inserer);
}

/* ********************************************************************** *
 * Supprime le premier élément de la liste                                *
 * Entrée : la liste à modifier                                           *
 * ********************************************************************** */
void supprime(ListeChainee& l)
{
  if (not est_vide(l)) {
    l = move(l->suite);
  }
}

/* ********************************************************************** *
 * Affiche le contenu d'une liste chaînée                                 *
 * Entrée : La liste à afficher                                           *
 * ********************************************************************** */
void affiche_R(const ListeChainee& liste); // partie récursive de l'affichage

void affiche(const ListeChainee& liste) // partie principale de l'affichage
{
  cout << "taille : " << taille(liste) << endl;
  affiche_R(liste);
  cout << endl;
}
void affiche_R(const ListeChainee& l)
{
  if (not est_vide(l)) {
    cout << " -> " << l->valeur;
    affiche_R(l->suite);
  }
}

/* ********************************************************************** *
 * Calcule la taille d'une liste chaînée                                  *
 * Entrée : La liste                                                      *
 * ********************************************************************** */
size_t taille(const ListeChainee& l)
{
  if (est_vide(l)) return 0;
```

```
  else return 1 + taille(l->suite);
}
```

Voici une version avec des `forward_list`s. Ceci fait appel à des notions qui seront présentées au second semestre (iterator). Je ne donne le code ici que pour illustration/information. Il est encore hors programme pour le moment.

```
#include <iostream>
#include <forward_list>
using namespace std;

// ===== DÉFINITION DES TYPES =====
typedef int type_el; // type des éléments de la liste

typedef forward_list<type_el> ListeChainee;
typedef ListeChainee::const_iterator Element;

// ====== PROTOYTPES =====

// insere un nouvel élément dans la liste
void insere(ListeChainee& l, type_el e)
{ l.push_front(e); }

// insere un nouvel élément après un élément d'une liste
void insere(Element& existant, type_el a_inserer)
{ ListeChainee().insert_after(existant, a_inserer); }
/** jcc: je ne comprends pas pourquoi ils ont fait de insert_after
    une non static member function!! Ca n'a pas de sens (quid
    si incompatibilité entre l et e ??) !!
    Avec g++-4.6 en tout cas il n'y a pas de vérification de compatibilité et
    le code ci-dessus fonctionne...

    Sinon : il faudrait changer la définition de Element :
    struct Element {
      ListeChainee::const_iterator position;
      ListeChainee chaine_mere;
    };
    et faire ici :
     existant.chaine_mere.insert_after(existant.position, a_inserer);
 **/

// supprime le premier élément de la liste
void supprime(ListeChainee& l)
{ l.pop_front(); }

// supprime l'élément suivant l'élément donné
void supprime(Element& e)
{ ListeChainee().erase_after(e); }
/** jcc: même remarque que pour insert_after
 **/

// détermine si la liste est vide
bool est_vide(const ListeChainee& l)
{ return l.empty(); }

// calcule la longueur de la liste
size_t taille(const ListeChainee& l);

// supprime de la liste toutes les valeurs égale à celle indiquée
void supprime(ListeChainee& l, type_el e)
{ l.remove(e); }

// supprime de la liste toutes les occurences de l'élément
void supprime(ListeChainee& l, const Element& e)
{ l.remove(*e); }

// vide la liste
void vide(ListeChainee& l)
{ l.clear(); }
```

```
// affiche le contenu d'une liste
void affiche(const ListeChainee& l);

//--------------------------------------------------------------------------
int main()
{
  ListeChainee maliste({12,-34});
  type_el une_valeur = 3;

  insere(maliste, une_valeur);
  insere(maliste, -1);
  insere(maliste, 2);
  insere(maliste, 1);
  insere(maliste, 0);

  affiche(maliste);

  supprime(maliste); // supprime la tête
  affiche(maliste);

  supprime(maliste, 2);  // supprime les valeurs 2
  affiche(maliste);

  return 0;
}

/* ************************************************************************ *
 * Affiche le contenu d'une liste chaînée                                   *
 * Entrée : La liste à afficher                                             *
 * ************************************************************************ */
void affiche(const ListeChainee& liste)
{
  cout << "taille : " << taille(liste) << endl;
  for (auto el : liste) {
    cout  << el << " ";
  }
  cout << endl;
}

/* ************************************************************************ *
 * Calcule la taille d'une liste chaînée                                    *
 * Entrée : La liste                                                        *
 * ************************************************************************ */
size_t taille(const ListeChainee& liste)
{
  size_t t(0);
  for (auto el : liste) { ++t; }
  return t;
}
```

Dernière mise à jour le 25 novembre 2021
Last modified: Thu Nov 25, 2021