TP 1

1 Et si on changeait de compilateur

Si on ne dispose plus d'un compilateur `C++`, mais seulement d'un compilateur `C`, on perd la possibilité d'utiliser quelques éléments qui facilitent et agrémentent l'exercice de la programmation procédurale et fonctionnelle :

- pas de références en C,
- pas de surcharge de fonctions (possibilité de donner le même nom à des fonctions agissant sur des paramètres en nombre ou de type différent) en C,
- pas d'entrées/sorties utilisant les classes `cin` et `cout` (utilisées en LIFAP1, LIFAP3 et LIFAP4) en C,
- nécessité en C de faire précéder le nom d'un type défini par une structure du mot clé `struct` (mais on peut ensuite faire un `typedef` pour se débarrasser de ce mot clé),
- absence du mécanisme de constructeurs et de destructeur en C,
- allocation dynamique de mémoire dans le tas par les fonctions `malloc` et `free`, mais impossibilité en C d'utiliser les opérateurs `new` et `delete`, qui couplent l'allocation dynamique de mémoire avec un appel automatique du constructeur et du destructeur sur la zone allouée.

On récapitule ici les modifications qui vont en découler dans votre manière de programmer :

Passage de paramètres

Jusqu'à présent, nous avons bénéficié des passages par référence du langage `C++` pour mettre en œuvre les passages de paramètres MODIFIABLES (**données-résultat** et **résultat**) (voir annexe).
La mise en œuvre en langage `C` de paramètres MODIFIABLES (**résultat** ou **données-résultat**) est plus délicate. Elle doit être gérée via l'utilisation de pointeurs, d'une part par le programmeur de la procédure, mais aussi par les utilisateurs, qui devront s'adapter au mécanisme d'indirection ainsi introduit. C'est ce qu'on appelle un passage par adresse.

Exemple d'une procédure `h` ayant un paramètre MODIFIABLE (**données-résultat** ou **résultat**) de type `T` :

```
void h(T *px){Code de la fonction agissant sur *px}
```

L'appel de `h` sur une variable `a` de type `T` s'écrit donc `h(&a)`.

Allocation mémoire dans le tas

Le langage `C` n'offre pas les opérateurs `new` et `delete`, il convient donc d'utiliser les fonctions `malloc` et `free` (bien inclure <`stdlib.h`> en début de fichier).

Import de fonctions de la bibliothèque standard

Le langage `C` ne plonge pas les fonctions de la bibliothèque standard dans l'espace de nom `std`. Par ailleurs, les fichiers de la bibliothèque standard à inclure sont suffixés par `.h`.

Ainsi, on fera appel à `printf` en `C` et à `std::printf` en `C++`, et on inclura <`stdio.h`> en `C` et <`cstdio`> en `C++`.

Structure

Étant donné une structure C :

```
struct S
{
  int s1;
  int s2;
};
```

Voici un exemple de définition sur la pile d'une variable de type `struct S` :

```
struct S vs;
```

1.1 A vous de jouer !

Sur la page web du cours, vous trouverez la mise en œuvre en `C++` (compilable par le compilateur `g++`) du module `Nombre Complexe` vu en cours, ainsi qu'une seconde mise en œuvre en `C` (compilable par le compilateur `gcc`). Ces fichiers pourront vous servir d'exemple.

On se propose à présent de réaliser l'implantation du type abstrait `Liste`, respectivement en `C++` et en `C` pur. Les interfaces du module (c'est à dire les fichiers .h) sont disponibles sur la page web du cours pour chacune des 2 versions ainsi que des exemples de fichiers utilisateurs.

Commencez par étudier les fichiers fournis, et répondez aux questions suivantes :
- Dessiner une `Liste` vide.
- Dessiner une `Liste` dans laquelle on a mis 3 éléments.
- Qu'est-ce que la copie profonde d'une `Liste` ?
- Quelle est la nuance entre une `Liste` vide et une `Liste` sur laquelle le destructeur a été appelé ? Vous pouvez notamment commencer à réfléchir au cas de la liste chainée circulaire (voir exercice suivant).

Commencez par écrire le code de la procédure d'initialisation d'une `Liste` en une liste vide (constructeur en `C++`), la procédure d'ajout en tête, la procédure d'affichage, puis la procédure testament (destructeur en `C++`). Le reste des autres procédures n'est à écrire que si vous avez fini le reste du TP. Testez bien votre code au fur et à mesure en utilisant le programme de test. Nous vous recommandons d'écrire tour à tour chaque procédure ou fonction en `C++` et en `C`, de manière à maîtriser les deux langages.

Remarque : La classe `Cellule` aurait pu être dotée d'un constructeur et d'un destructeur. Nous ne souhaitons néanmoins pas mettre de désallocation dynamique dans le destructeur de `Cellule` car nous voulons que le prototype des opérations sur les `Cellule`s et sur les `Listes` ne change pas lorsqu'on passera à une implantation différente (voir exercice suivant sur les listes chaînées circulaires).

```
\\ liste.h (version C++)
#ifndef _LISTE
#define _LISTE
#include "element.h" //offrant le type Elem

class Liste; // declaration

class Cellule
```

```
{
    friend class Liste;
    private :
        Elem info;
        Cellule *psuivant;
};

class Liste
{
    public :
    //Constructeurs = Initialiseurs ----------------------------------------
    Liste();
    //Postcondition : la liste *this est  initialisée comme étant vide
    Liste(const Liste & l);
    //Postcondition : la liste *this est initialisée en copie profonde de l
    //         (mais elles sont totalement independantes l'une de l'autre)

    //Destructeur-----------------------------------------------------------
    ~Liste();
    //Postcondition : l'espace occupé par *this  peut-être restitué

    //Affectation-----------------------------------------------------------
    Liste & operator = (const Liste & l);
    //Précondition : aucune
    //       (la liste *this à affecter et l doivent être initialisées)
    //Postcondition : la liste *this correspond à une copie profonde de l
    //           (mais elles sont totalement independantes l'une de l'autre),

    bool testVide() const;
    //Précondition : aucune
    //               (*this initialisee)
    //Résultat : true si *this est vide, false sinon

    Elem premierElement() const;
    //Précondition : testListeVide(l)==false
    //Résultat : valeur de l'Elem contenu dans la 1ere Cellule

    Cellule * premiereCellule() const;
    //Précondition : aucune
    //               (*this initialisée)
    //Résultat : adresse de la premiere cellule de *this si this->testVide()==false
    //           0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiée à travers
    //           la connaissance de l'adresse de sa première cellule

    Cellule * celluleSuivante(const Cellule *pc) const;
    //Précondition : pc adresse valide d'une Cellule de la Liste *this
    //Résultat : adresse de la cellule suivante si elle existe
    //           0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiée à travers
    //           la connaissance de l'adresse d'une de ses cellules

    Elem elementCellule(const Cellule * pc) const;
    //Précondition : pc adresse valide d'une Cellule de la Liste *this
```

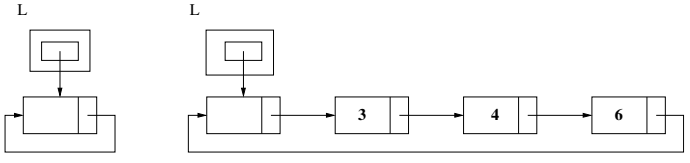
```
    //Résultat : valeur de l'Elem contenu dans la Cellule

    void affichage() const;
    //Précondition : aucune
    //               (*this initialisée)
    //Postcondition : Affichage exhaustif de tous les éléments de *this

    void ajoutEnTete(const Elem & e);
    //Précondition : aucune
    //               (*this et e initialisés)
    //Postcondition : L'Elem e est ajouté en tête de *this

    void suppressionEnTete();
    //Précondition : this->testVide()==false
    //Postcondition : la liste *this perd son premier élément

    void vide();
    //Précondition : aucune
    //               (*this initialisée)
    //Postcondition : this->testVide()==true (tous les éléments sont retirés)

    void ajoutEnQueue(const Elem & e);
    //Précondition : aucune
    //               (*this et e initialisés)
    //Postcondition : L'Elem e est ajouté en fin de la liste *this

    //OPERATIONS QUI POURRAIENT ETRE AJOUTEES AU MODULE

    Cellule * rechercheElement(const Elem & e) const;
    //Précondition : aucune
    //               (*this initialisée)
    //Résultat : Adresse de la première Cellule de *this contenant e, 0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiée à travers
    //           la connaissance de l'adresse d'une de ses cellules

    void insereElementApresCellule(const Elem & e,Cellule *pc);
    //Précondition : pc adresse valide d'une Cellule de la Liste *this
    //               ou 0 si this->testVide()==true
    //Postcondition : l'element e est inseré après la Cellule pointée par pc

    void modifieInfoCellule(const Elem & e, Cellule *pc);
    //Precondition : *this non vide et c adresse valide d'une Cellule de *this
    //Postcondition : l'info contenue dans *pc a pour valeur e

private :
        void ajoutEnQueueConnaissantUneCellule(const Elem & e, Cellule *pc);
    //Données membres----------------------------------------------------------
        Cellule *ad;
        int taille;
};
#endif


#ifndef _LISTE
```

```
#define _LISTE
\\ liste.h (version C)
/* Inclusion des modules utilisés par le module Liste */
/* (A l'exclusion de ceux qui sont seulement utilisés dans l'implantation : */
/* ceux là sont inclus dans le fichier d'implantation) */

#include "element.h" //offrant le type Elem

typedef struct _Cellule
{
  Elem info;
  struct _Cellule *psuivant;
} Cellule;

typedef struct _Liste
{
  Cellule *ad;
  int taille;
} Liste;


void initialiseListeDefaut(Liste *pl);
/* Précondition : *pl non préalablement initialisée */
/* Postcondition : la liste *pl initialisée est vide */

void initialiseListeCopie(Liste *pl1, Liste l2);
/* Précondition : *pl1 non préalablement initialisée,  */
/*                        l2 préalablement initialisée */
/* Postcondition : la liste *pl1 est initialisée avec une copie
/*                         profonde de l2 */


void testamentListe(Liste *pl);
/* Précondition : *pl préalablement initialisée */
/* Postcondition : *pl prête à être détruite */

void affectationListe(Liste *pl1, Liste l2);
/*Précondition : *pl1 et l2 préalablement initialisées  */
/* Postcondition : la liste *pl1 correspond à une copie */
/*                         profonde de l2  après affectation*/

int testListeVide(Liste l);
/* Précondition : l préalablement initialisée */
/* Résultat : 1 si l est vide, 0 sinon */

Elem premierElementListe(Liste l);
/* Précondition : testListeVide(l)==false */
/* Résultat : valeur de l'Elem contenu dans la 1ere Cellule */

Cellule * premiereCelluleListe(Liste l);
/* Précondition : l initialisée  */
/* Resultat : adresse de la premiere cellule de l si testListeVide(l)==false */
/*            NULL sinon */
/*            Attention : la liste l pourra ensuite être modifiée à travers  */
```

```
/*            la connaissance de l'adresse de sa première cellule */

Cellule * celluleSuivanteListe(const Cellule *pc, Liste l);
/* Précondition : pc adresse valide d'une Cellule de la Liste l */
/* Résultat : adresse de la cellule suivante si elle existe */
/*            NULL sinon */
/*            Attention : la liste l pourra ensuite être modifiée à travers  */
/*            la connaissance de l'adresse d'une de ses cellules */

Elem elementCelluleListe(const Cellule * pc);
/* Précondition : pc adresse valide d'une Cellule d'une Liste */
/* Résultat : valeur de l'Elem contenu dans la Cellule */

void affichageListe(Liste l);
/* Précondition : l initialisée  */
/* Postcondition : Affichage exhaustif de tous les elements de l */

void ajoutEnTeteListe(Elem e,Liste *pl);
/* Précondition : *pl et e initialisés  */
/* Postcondition : L'Elem e est ajouté en tête de *pl */

void suppressionEnTeteListe(Liste *pl);
/* Précondition : testListeVide(*pl)==false */
/* Postcondition : la liste *pl perd son premier élément */

void videListe(Liste *pl);
/* Précondition : aucune  */
/*                (*pl initialisée)  */
/* Postcondition : testlisteVide(*pl)==true */
/*                 tous les éléments ont été retirés */

void ajoutEnQueueListe(Elem e,Liste *pl);
/* Précondition : *pl et e initialisés  */
/* Postcondition : L'Elem e est ajoute en fin de la liste *pl */

#endif
```

1.2 Liste chaînée circulaire

De manière à enrichir votre bibliothèque de modules algorithmiques, reprenez à présent votre module `Liste` avec une implantation différente, chaînée circulaire et utilisant une cellule bidon (sentinelle). Vous souvenez-vous de l'intérêt d'une implantation circulaire en terme de complexité ?



L : Liste vide

L : Liste de trois éléments : 3, 4, 6

Annexe

Passage de paramètres en C++

— Procédure `f` ayant un paramètre MODIFIABLE (**données-résultat** ou **résultat**) `x` de type `T` :
```
void f(T & x)
```

— Procédure `g` ayant un paramètre NON MODIFIABLE (**données**) `x` de type `T` :
On a le choix entre 2 mises en œuvre possibles.
```
void g(T x)
void g(const T & x)
```

L'appel à `f` ou `g` sur une variable `a` de type `T` s'écrit directement `f(a)` ou `g(a)`, sans se soucier du mode de passage.