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 facilitaient et aggrémentaient l'exercice de la programmation procédurale :

- pas de références en C pur,
- pas de surcharge de fonctions (possibilité de donner le même nom à des fonctions différentes),
- pas d'entrées/sorties utilisant les classes `cin` et `cout` (utilisées en LIFAP1, LIFAP3 et LIFAP4),
- possibilité de commenter une ligne par
  \\
- nécessité en C de faire précéder le nom d'un type défini par une structure du mot clé `struct` (mais on peut faire un `typedef`),
- pas d'allocation dynamique de mémoire par les opérateurs `new` et `delete`, mais par les fonctions `malloc` et `free`.

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 références du langage `C++` pour mettre en œuvre les passages de paramètres **données-résultat** et **résultat** (cf. annexe).
La mise en œuvre en langage `C` de paramètres formels **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 l'utilisateur de la procédure, qui devra 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 formel **données-résultat** ou **résultat** de type `T` : `void h(T *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 d'interface de la bibliothèque standard sont suffixés par `.h`.

Ainsi, on fera appel à `printf` et non plus à `std::printf`, et on inclura <`stdio.h`> et non plus <`cstdio`>.

Commentaires

En C les commentaires se font ainsi (si vous utilisez une version antérieure à 1999) :

```
/* Exemple d'un commentaire ...
sur deux lignes! */
```

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 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 (fichiers .h) du module pour chacune des 2 versions sont disponibles sur la page web du cours, ainsi que des exemples de fichiers utilisateurs.

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 déallocation 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.

```
\\ 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 *suivant;
};

class Liste
{
    public :
    //Constructeurs-----------------------------------------------------------
    Liste();
    //Postcondition : la liste initialisee est vide
    Liste(const Liste & l);
    //Postcondition : la liste initialisee et l correspondent a des listes identiques
    //                (mais elles sont totalement independantes l'une de l'autre)

    //Destructeur-------------------------------------------------------------
    ~Liste();

    //Affectation-------------------------------------------------------------
    Liste & operator = (const Liste & l);
    //Precondition : aucune
    //               (la liste a affecter et l initialisees et manipulees uniquement
    //               a travers les operations du module)
    //Postcondition : la liste affectee et l correspondent a des listes identiques
    //                (mais elles sont totalement independantes l'une de l'autre)

    bool testVide() const;
    //Precondition : aucune
    //               (*this initialisee et manipulee uniquement a travers les
    //                operations du module)
    //Resultat : true si *this est vide, false sinon

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

    Cellule * premiereCellule() const;
    //Precondition : aucune
    //               (*this initialisee et manipulee uniquement a travers les
    //                operations du module)
    //Resultat : adresse de la premiere cellule de *this si this->testVide()==false
    //           0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiee a travers
    //           la connaissance de l'adresse de sa premiere cellule

    Cellule * celluleSuivante(const Cellule *c) const;
    //Precondition : c adresse valide d'une Cellule de la Liste *this
    //Resultat : adresse de la cellule suivante si elle existe
    //           0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiee a travers
    //           la connaissance de l'adresse d'une de ses cellules

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

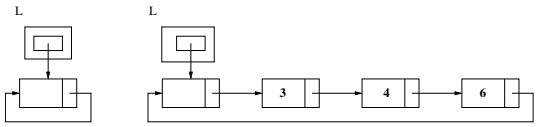
```
    //Resultat : valeur de l'Elem contenu dans la Cellule

    void affichage() const;
    //Precondition : aucune
    //               (*this initialisee et manipulee uniquement a travers les
    //                operations du module)
    //Postcondition : Affichage exhaustif de tous les elements de *this

    void ajoutEnTete(const Elem & e);
    //Precondition : aucune
    //               (*this et e initialises et manipules uniquement a travers les
    //                operations de leurs modules respectifs)
    //Postcondition : L'Elem e est ajoute en tete de *this

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

    void vide();
    //Precondition : aucune
    //               (*this initialisee et manipulee uniquement a travers les
    //                operations du module)
    //Postcondition : this->testVide()==true

    void ajoutEnQueue(const Elem & e);
    //Precondition : aucune
    //               (*this et e initialises et manipules uniquement a travers les
    //                operations de leurs modules respectifs)
    //Precondition : L'Elem e est ajoute en fin de la liste *this

    //OPERATIONS QUI POURRAIENT ETRE AJOUTEES AU MODULE

    Cellule * rechercheElement(const Elem & e) const;
    //Precondition : aucune
    //               (*this initialisee et manipulee uniquement a travers les
    //                operations du module)
    //Resultat : Adresse de la premiere Cellule de *this contenant e, 0 sinon
    //           Attention : la liste *this pourrait ensuite etre modifiee a travers
    //           la connaissance de l'adresse d'une de ses cellules

    void insereElementApresCellule(const Elem & e,Cellule *c);
    //Precondition : c adresse valide d'une Cellule de la Liste *this
    //               ou 0 si this->testVide()==true
    //Postcondition : l'element e est insere apres la Cellule pointee par c

    void modifieInfoCellule(const Elem & e,Cellule *c);
    //Precondition : *this non vide et c adresse valide d'une Cellule de *this
    //Postcondition : l'info contenue dans *c a pour valeur e
    private :
        void ajoutEnQueueConnaissantUneCellule(const Elem & e, Cellule *c);
    //Donnees membres----------------------------------------------------------
        Cellule *ad;
        int taille;
};
```

```
#endif


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

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

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

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


void initialiseListeDefaut(Liste *pl);
/* Precondition : *pl non prealablement initialisee */
/* Postcondition : la liste *pl initialisee est vide */

void initialiseListeCopie(Liste *pl1, Liste l2);
/* Precondition : *pl1 non prealablement initialisee,  */
/*                l2 prealablement initialisee et manipulee uniquement */
/*                a travers les operations du module  */
/* Postcondition : la liste *pl1 initialisee correspond a une copie de l2 */
/*  (mais les 2 listes sont totalement independantes l'une de l'autre) */

void testamentListe(Liste *pl);
/* Precondition : *pl prealablement initialisee et manipulee uniquement */
/*                a travers les operations du module */
/* Postcondition : *pl prete a etre detruite */

void affectationListe(Liste *pl1, Liste l2);
/*Precondition : *pl1 et l2 prealablement initialisees et manipulees */
/*               uniquement a travers les operations du module */
/* Postcondition : la liste *pl1 correspond a une copie de l2 */
/*  (mais les 2 listes sont totalement independantes l'une de l'autre) */

int testListeVide(Liste l);
/* Precondition : l prealablement initialisee et manipulee uniquement */
/*                a travers les operations du module */
/* Resultat : 1 si l est vide, 0 sinon */

Elem premierElementListe(Liste l);
/* Precondition : testListeVide(l)==false */
```

```
/* Resultat : valeur de l'Elem contenu dans la 1ere Cellule */

Cellule * premiereCelluleListe(Liste l);
/* Precondition : l initialisee et manipulee uniquement a travers les  */
/*                 operations du module */
/* Resultat : adresse de la premiere cellule de l si testListeVide(l)==false */
/*            NULL sinon */
/*            Attention : la liste l pourra ensuite etre modifiee a travers  */
/*            la connaissance de l'adresse de sa premiere cellule */

Cellule * celluleSuivanteListe(const Cellule *pc, Liste l);
/* Precondition : pc adresse valide d'une Cellule de la Liste l */
/* Resultat : adresse de la cellule suivante si elle existe */
/*            NULL sinon */
/*            Attention : la liste l pourra ensuite etre modifiee a travers  */
/*            la connaissance de l'adresse d'une de ses cellules */

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

void affichageListe(Liste l);
/* Precondition : l initialisee et manipulee uniquement a travers les  */
/*                 operations du module */
/* Postcondition : Affichage exhaustif de tous les elements de l */

void ajoutEnTeteListe(Elem e,Liste *pl);
/* Precondition : *pl et e initialises et manipules uniquement a travers les  */
/*                 operations de leurs modules respectifs) */
/* Postcondition : L'Elem e est ajoute en tete de *pl */

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

void videListe(Liste *pl);
/* Precondition : aucune  */
/*                (*pl initialisee et manipulee uniquement a travers les  */
/*                 operations du module) */
/* Postcondition : testlisteVide(pl)==true */

void ajoutEnQueueListe(Elem e,Liste *pl);
/* Precondition : *pl et e initialises et manipules uniquement a travers les  */
/*                 operations de leurs modules respectifs */
/* Precondition : L'Elem e est ajoute en fin de la liste *l */

#endif
```

1.2 Liste chaînée circulaire

De manière à enrichir votre bibliothèque de modules, 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 formel **données-résultat** ou **résultat** `x` de type `T` :
`void f(T & x)`

— Procédure `g` ayant un paramètre formel **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.