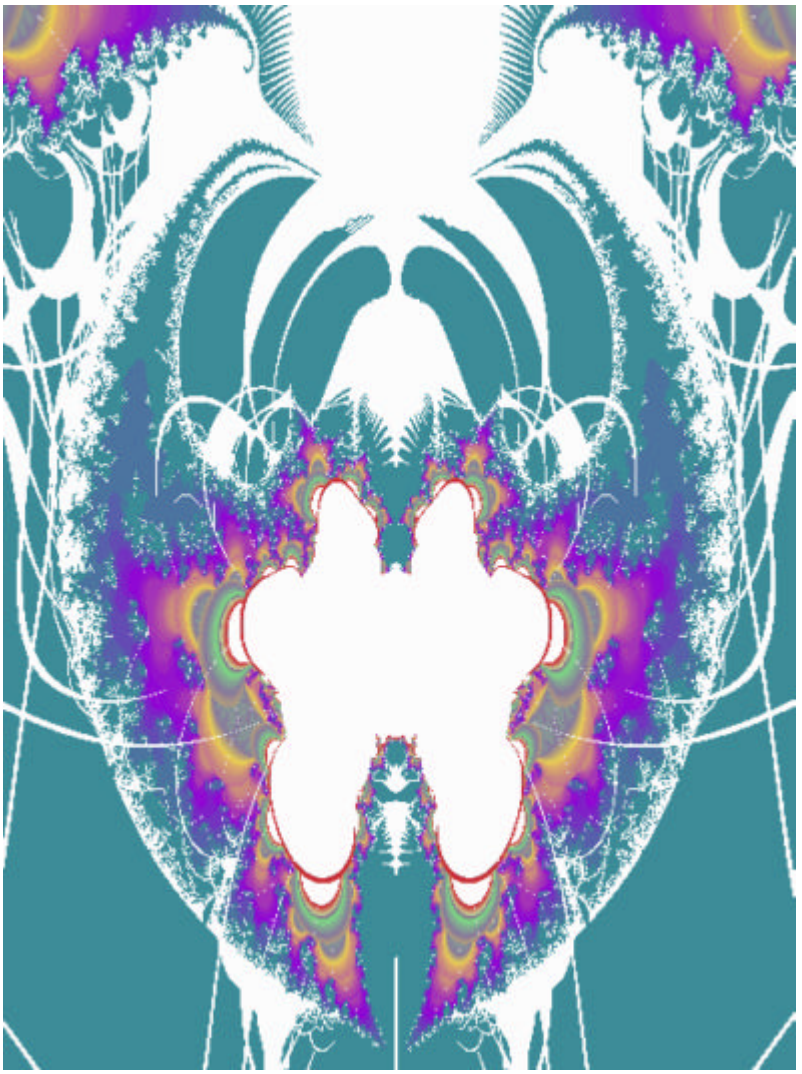
CHAPITRE 13

Classes imbriquées



13.1 *Motivation*

A partir de la version 3.0 de la spécification du langage C++, il est possible d'utiliser des types imbriqués. Cette fonctionnalité permet de cacher des détails d'implémentation en les rendant purement locaux à une classe donnée, comme dans l'exemple suivant :

```
class     LinkedList
     {
     private :
          struct    ListElement
               {
               void*     data;
               ListElement    *next, *previous;
               };
          LinkedList&operator=(LinkedList& l);
          LinkedList(LinkedList&);
          ListElement    *head, *tail;
     public :
          LinkedList();
          ~LinkedList();
          void      addToList(const void *newData);
          ....
     };
```

L'élément de la liste n'est pas accessible en dehors des méthodes de la classe `LinkedList`, puisque la définition du type se fait dans le contexte de la classe uniquement. L'élément englobé, lui, n'a qu'une visibilité très limitée:

```
int  x, y;
class     outer
     {
     public :
          int x;
          static    int  staticVar;
     class     inner
          {
          void      someFunc(int anInt)
               {
               x = anInt;     // Erreur, outer::x n'est pas accessible.
                              // outer::x ne correspond pas à une instance
               staticVar = anInt; // staticVar est statique, ok
               ::x = anInt;       // Ok, variable globale
               y = anInt;         // Ok, variable globale
               }
          void      anotherFunc(outer *outerPtr, int anInt)

               {
               outerPtr->x = anInt;    // Ok, accès à un membre de la
                                       // classe englobante par un
                                       // pointeur.
               }
          }
     };
```

Un autre avantage de la définition de classes imbriquées réside dans le fait que l'identi-

ficateur de classe, étant local au contexte d'une classe englobante, n'apparaît pas à l'extérieur du contexte de la classe. Il n'apparaît de ce fait pas non plus dans la table de symboles globaux. Un programme C++, comme tout programme orienté objets, essaie de réutiliser un maximum de code, quitte à importer également des fonctionnalités inutiles, à seule fin de profiter de certaines fonctionnalités déjà développées dans le cadre d'un autre sous-système. Cette manière de faire tend à remplir la table de symboles globaux d'identificateurs inutiles, qui peuvent éventuellement créer des conflits lors de la réutilisation de code provenant de diverses sources indépendantes. On parle dans le jargon technique de C++ de "*global name space pollution*". Toute technique permettant d'éviter d'exporter des identificateurs inutiles dans la table de symboles globale est de ce point de vue, utile. Utiliser des classes imbriquées est une manière de faire tout à fait efficace de ce point de vue.

13.2 *Alternative*

A vrai dire, il est possible de cacher encore beaucoup plus efficacement une implémentation à l'aide d'un pointeur sur une classe non exportée, comme nous l'avons déjà vu lors de notre discussion sur les membres privés, et la manière de se protéger contre des accès inautorisés :

Fichier llist.h :

```
class     ListImplem;// Classe simplement déclarée par son identificateur,
                     // mais pas par son interface.
class     LinkedList
     {
     private :
          ListImplem    *_impl;
          LinkedList&   operator=(LinkedList& l);
          LinkedList(LinkedList&);
     public :
          LinkedList();
          ~LinkedList();
          void      addToList(const void *newData);
          ....
     };
Fichier llist.C : // non exporté, en principe inaccessible

class     ListImplem
     {
     public :
          struct   ListElement
               {
               void*    data;
               ListElement   *next, *previous;
               };
          ListElement   *head, *tail;
          ListImplem();
          ~ListImplem();
     ...
     };
```

Dans la mesure du possible, plus l'interface à une classe est concis, plus il est simple à comprendre. Une bonne manière de le rendre concis est donc de le débarasser de tout ce qui ne concerne pas directement les utilisateurs, soit les membres privés, essentiellement, même s'il faut pour cela faire quelques concessions sur les fonctions `inline`.

Il serait faux, en revanche, de limiter l'utilisabilité de la classe de manière à rendre cet interface concis. Une classe trop limitée ne sera pas utilisée. Si un interface semble devenir trop large et trop complexe, il vaut la peine de se demander si l'on n'a pas fait une erreur lors de la définition de la classe, et si un redesign ne serait pas approprié.