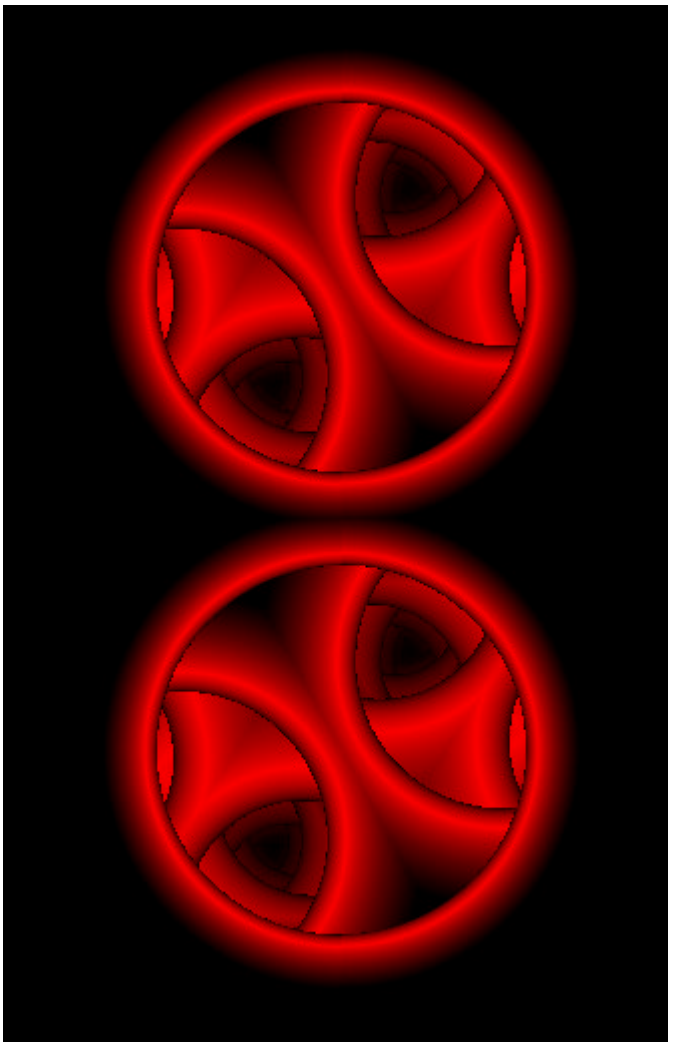
CHAPITRE 17

Classes génériques



17.1 *Motivation*

Les classes génériques ne font pas partie du langage C++, mais se révèlent très pratiques pour implémenter des mécanismes de base réutilisables par un grand nombre. Utilisé en conjonction avec la construction template de C++, les classes génériques permettent d'allier un code optimal du point de vue de la taille et de la réutilisation à une écriture claire et propre du code.

L'utilisation de templates permet en soi d'écrire des classes génériques, mais présente un inconvénient : pour chaque instanciation de template, le code implémentant le template doit être entièrement régénéré. Si dans notre exemple de tableau de longueur variable, cette régénération ne conduit pas à une multiplication excessive de la quantité de code, ceci peut s'avérer gênant dans le cas de template complexes. Or, nous l'avons vu dans l'exemple précédent, le code généré par les différentes instantiations de template est fondamentalement identique, à quelques rares exceptions près.

L'idée de classes génériques est d'écrire une classe indépendante du type d'objet manipulé. Pour ce faire, on dispose d'un outil en C++, qui sont les pointeurs sur des objets de type void (`void*`) ou pointeurs génériques. Nous avons déjà parlé de ce type de pointeur particulier; rappelons que le pointeur `void*` peut pointer sur un objet de n'importe quel type, mais qu'en revanche, il n'est pas possible de déréférencer directement un pointeur de type `void*`, ni d'assigner la valeur d'un pointeur de type `void*` sur un pointeur sur un autre type (une conversion explicite est nécessaire).

17.2 *Pile générique (GenericStack)*

Si nous examinons notre exemple de pile du chapitre précédent, définissons une pile générique, appelée GenericStack au moyen de pointeurs génériques (`void*`).

```
class     GenericStack
     {
     private :
          class     StackNode
               {
               public :
                    void          *data;
                    StackNode     *next;
                    StackNode(void *newData, StackNode *nextNode)
                         : data(newData), next(nextNode) {}
               };
     public :
          GenericStack();
          ~GenericStack();
          void push(const void *object);
          void * pop();
     };
```

Cette implémentation est dangereuse, car elle n'exprime pas le fait que tous les objets de la pile doivent être du même type. Un utilisateur pourrait utiliser cette pile pour mettre un mélange de réels, d'entiers et de classes arbitraires sans que le compilateur n'y trouve quoi que ce soit à redire. L'implémentateur de la classe doit interdire un tel abus, volontaire ou accidentel (à moins qu'il ne soit prêt à payer le prix de l'implémentation d'une pile (stack) d'objets de type quelconque). La solution est de générer des classes d'interface à GenericStack, de la manière suivante :

```
class     IntStack
     {
     private :
          GenericStack       s;
     public :
          void push(const int* anInt) { s.push(anInt);}
          int* pop() { return (int*) s.pop(); }
     };

class     FloatStack
     {
     private :
          GenericStack  s;
     public :
          void push(const float* aFloat) { s.push(aFloat);}
          float* pop() { return (float*) s.pop(); }
     };
```

Il subsiste néanmoins encore un inconvénient : il est toujours possible à un utilisateur d'instancier GenericStack. Or GenericStack n'est pas destiné à être exporté, mais uniquement à économiser du code. C'est là typiquement une relation de type "est implémenté en termes

de": IntStack sera implémenté comme un GenericStack. Ceci nous conduit à utiliser l'héritage privé :

```
class     GenericStack
     {
     private :
          struct    StackNode
               {
               void      *data;
               StackNode *next;
               StackNode(void *newData, StackNode *nextNode)
                    : data(newData), next(nextNode) {}
               };
     protected :
          GenericStack();
          ~GenericStack();
          void push(const void *object);
          void * pop();
     };

class     IntStack : private GenericStack
     {
     public :
          void      push(const int* anInt)
               { GenericStack::push(anInt);}
          int* pop() { return (int*) GenericStack::pop(); }
     };
```

Le problème de cette implémentation est que nous devons définir une nouvelle classe pour chaque type de données que nous voulons mettre dans la pile. Non seulement il s'agit là d'un travail fastidieux, mais il est impossible de prévoir à l'avance tous les types de données que de futurs utilisateurs pourraient avoir envie de mettre dans la pile, à moins de laisser à ces derniers le soin de créer eux-mêmes les classes dérivées. Notre implémentation originale par template évitait cet inconvénient. Mais rien n'empêche de réutiliser les template à ce niveau du design :

```
template      <class    T>
class     Stack : private GenericStack
     {
     public :
          void      push(const T* anObjectPtr)
               { GenericStack::push(anObjectPtr);}
          T* pop() { return (T*) GenericStack::pop(); }
     };
typedef Stack<int> IntStack;
```

Ce design réunit les avantages des templates aux avantages de la réutilisation de code par héritage. En résumé, on retiendra les remarques suivantes :

- On utilisera des modèles lorsqu'il s'agit de générer des collections de classes pour lesquelles le type des objets n'affecte pas le comportement des méthodes de la classe.
- L'héritage doit être utilisé lorsque le type des objets affecte le comportement de la classe.

- Les modèles peuvent être utilisés avantageusement en conjonction avec des classes génériques. On combine ainsi la réutilisation du code avec l'automatisation de génération de nouvelles classes offerte par les templates. Mais vu le caractère forcément très flexible de la classe générique, il faut s'assurer qu'un utilisateur ne puisse pas l'instancier en tant que telle.

17.3 *Un exemple complet (FileOfRecord)*

Nous nous proposons ici d'implémenter, à titre d'exemple, une classe permettant de construire des fichiers composés d'enregistrements de longueur fixe, ayant à peu près les mêmes caractéristiques que la construction FILE OF du langage PASCAL. Cette construction est intéressante par le fait qu'elle permet d'effectuer des entrées-sortie sur fichier de manière structurée, et qu'elle apporte une certaine abstraction de l'implémentation du fichier vis-à-vis du type de données que l'on manipule.

En C++, on pourrait implémenter cette classe un peu de la manière suivante:

```
//
//   Definition : TypedFile.h
//

#include    <fstream.h>
#include    <iostream.h>
include    <stdlib.h>
#include    <unistd.h>
#include    <stdio.h>

class   FileOfRecord
    {
    private :
        int     recordSize;     // Size of one data record
        fstream f;              // Physical file
        bool fileOpen;       // Early catch of programming errors

                                // Make the copy constructor illegal !
                                // (Do not implement it)
        FileOfRecord(FileOfRecord&);
                                // Same for the assignment operator
        FileOfRecord& operator=(FileOfRecord&);

            // This class can be instantiated only by subclasses!
    protected :
        FileOfRecord(int recSize)
            {
            fileOpen = FALSE;
            recordSize = recSize;
            }
            //
            // These methods can only be used by an inheriting class
            //
        bool putRecord(const void *recToPut);
        bool getRecord(void *whereToPutIt);

            // These methods may be used by an application
    public :
            // This is much like fstream:open().
        bool openFile(char *filename,
                        int openMode = ios::in | ios::out,
                        int protection = (int) filebuf::openprot);
```

```
            // This is much like fstream:close().
        void    closeFile();
            // Seek to absolute position related to Begin Of File :
        bool seekTo(long recordNumber);
            // Jump to begin of file :
        bool gotoBOF();
            // Jump to end of file :
        bool gotoEOF();
            // relative moves :
        bool seekRelative(long offset);
            // simply return last system error
        int     getErrorNumber() { return errno; }
            // same, but with the system error string
        const   char    *getSysErrorMessage()
              { return strerror(errno); }
            // File size in records
        const   int     fileSize();
    };


//
// Implementation : TypedFile.C
//
// This implementation is system-specific. Shown here is an
//   implementation for an HP-UX V9.X file system on a PA-RISC
//   UNIX machine. However, adapting the implementation for other
//   machines should be quite easy.

#include    <sys/stat.h>
#include    "TypedFile.h"

boolean FileOfRecord::openFile(char *filename, int openMode,
                        int protection)

    {
    f.open(filename, openMode, protection);
    if (!f.fail()) fileOpen = true;
    return fileOpen;
    }

void    FileOfRecord::closeFile()

    {
    if (!fileOpen) return;
    f.close();
    fileOpen = false;
    }

const   int   FileOfRecord::fileSize()

    {
    if (!fileOpen) return 0;

    struct stat buf;
    int fildes = f.rdbuf()->fd();
    if (fstat(fildes, &buf) != 0) return 0;
    return(buf.st_size / recordSize);
```

```
    }

bool FileOfRecord::seekTo(long recordNumber)

    {
    if (!fileOpen) return false;
    f.seekg(recordNumber * recordSize);
    if (f.fail())
        {
        f.clear(0);
        return false;
        }
    return true;
    }

bool FileOfRecord::seekRelative(long offset)

    {
    if (!fileOpen) return false;
    long    actpos = f.tellg();
    if (f.fail())
        {
        f.clear(0);
        return false;
        }
    actpos = actpos / recordSize + offset;
    if ((actpos >= 0) && (actpos <= fileSize()))
        return seekTo(actpos);
    else return false;
    }

bool FileOfRecord::gotoBOF()

    {
    if (!fileOpen) return false;
    return seekTo(0L);
    }

bool FileOfRecord::gotoEOF()

    {
    if (!fileOpen) return false;
    f.seekg(0L, ios::end);
    if (f.fail())
        {
        f.clear(0);
        return false;
        }
    return true;
    }

bool FileOfRecord::putRecord(const void *recToPut)

    {
    f.write((const char*) recToPut, recordSize);
    if (f.fail())
        {
```

```
        f.clear(0);
        return false;
        }
    return true;
    }

bool FileOfRecord::getRecord(void *whereToPutIt)

    {
    f.read((char*) whereToPutIt, recordSize);
    if (f.fail())
        {
        f.clear(0);
        return false;
        }
    return true;
    }
```

On remarque que le type d'objets contenus dans le fichier n'est mentionné nulle part. Les méthodes d'accès aux données que sont `getRecord()` et `putRecord()` utilisent des pointeurs génériques. Cette classe ne peut pas être implémentée par l'application, puisqu'elle ne possède aucun constructeur public utilisable. Ce n'est pas par erreur, c'est parceque cette classe ne correspond pas à ce que nous désirons faire, qui est de déclarer un fichier comme en PASCAL, soit

```
VAR  f : FILE OF INTEGER;
```

par exemple. Dans notre implémentation, nous devrions (à supposer que le constructeur existe, et soit déclaré public, ainsi que la méthode `getRecord()`) déclarer :

```
FileOfRecord  f(sizeof(int));

int           anInt;

if (f.getRecord((const void*) &anInt)) { ... }
```

Non seulement ce n'est pas très joli, mais cela ne réalise pas vraiment l'abstraction que nous souhaitions obtenir, à savoir ce que l'on pouvait attendre de PASCAL :

```
VAR  f : FILE OF INTEGER;
VAR  anInt : INTEGER;

read(f, anInt);
```

Il est possible d'utiliser les possibilités de macros en C pour résoudre ce problème. Ainsi, on pourrait définir :

```
#define   TYPEDFILE(TYPE) \
class     TYPEDFILE : public FileOfRecord \
    { \
    public : \
```

```
        TYPEDFILE() : FileOfRecord(sizeof(TYPE)) { ; } \
        ~TYPEDFILE() { closeFile(); } \
        bool put(TYPE& recToPut) \
            { return putRecord((const void*) &recToPut); } \
        bool get(TYPE& whereToPutIt) \
            { return getRecord((const void*) &whereToPutIt); } \
    };
```

Cette méthode fonctionne très bien, et remplit parfaitement les desiderata que nous avions fixé. Elle présente néanmoins les inconvénients inhérents aux macros. De plus, la génération d'un listing par le compilateur étend la macro çi-dessus sur une seule ligne; il y a de ce fait peu de chances que sa définition tienne sur le papier.

La construction template permet une meilleure abstraction, et évite le défaut de la simple substitution textuelle du préprocesseur. La syntaxe en est la suivante :

```
template    <class TYPE>
class FileOf : public FileOfRecord
    {
    public :
        FileOf() : FileOfRecord(sizeof(TYPE)) { ; }
        ~FileOf() { closeFile(); }
        bool put(TYPE& recToPut)
            { return putRecord((const void*) &recToPut); }
        bool get(TYPE& whereToPutIt)
            { return getRecord((const void*) &whereToPutIt); }
    };
```

Déclarer un fichier d'entiers peut dès lors se faire de la manière suivante :

```
FileOf<int>   f;
int           anInt;

f.openFile("filename.tmp");
while (f.get(anInt)){ ... } // do something
f.closeFile();
```

On a réalisé une assez bonne approximation de la construction désirée, et le code spécifique à l'implémentation d'un fichier d'entiers est réduit à un minimum.