Les types abstraits de données (ADT)

par Emmanuel Delahaye (Espace personnel d'Emmanuel Delahaye)

Date de publication : 27 janvier 2008

Dernière mise à jour : 26 septembre 2010

Cet article présente la possibilité de programmer en C en utilisant une approche plus 'objet' que 'fonction'.

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Introduction

Le langage C est orienté 'fonction'. Cela signifie que la fonction est l'entité de base de la construction d'un programme écrit en C. L'idée directrice est qu'un programme n'est qu'une suite d'instructions organisées en fonctions. Cependant, force est de constater qu'un travail sans données n'a pas grand intérêt.

D'autres langages sont orientés 'objet'. Cela signifie que la construction du code est organisée autour des données. Les fonctions (méthodes) n'étant que des outils au service du traitement des données.

Le langage C ne dispose pas naturellement des 'classes' qui permettent de construire un objet auquel sont directement rattachées des 'méthodes'. Cependant, il est possible de programmer en C en utilisant une approche plus 'objet' que ne le laissait supposer le langage au départ. Un bon exemple est donné par l'implémentation de la gestion des flux en C standard, qui est organisée autour de l'objet de type FILE.

II - Analyse de l'objet standard FILE

Le langage C définit FILE comme un type dit 'opaque'. C'est à dire que le comportement est décrit, mais pas les détails internes de l'objet. Un certain nombre de fonctions associées définissent le comportement de l'objet. Il existe une paire de fonctions permettant de créer et détruire l'objet :

```
/* C90 */
 FILE *fopen (char const *filename, char const *mode);
 int fclose (FILE *stream);
```

on a aussi un certain nombre de fonctions permettant la lecture et l'ecriture par byte, par bloc, par ligne de texte etc.

Toutes ces fonctions ont en commun un paramètre de type FILE *. Ce paramètre contient l'adresse de l'objet qui a été créé lors de l'instanciation de celui-ci par fopen(). C'est en quelque sorte la 'clé' qui permet d'accéder à la fonction avec le bon contexte. Le principal intérêt de la programmation par contexte est que le code peut être unique et les instanciations infinies, ce qui va dans le sens de la capitalisation du code. De nombreux exemples sont donnés dans ma **bibliothèque de fonctions** qui est le plus souvent organisée en modules regroupés autour de la définition d'un certain objet.

III - Définition du type abstrait de donnée

Les données n'étant pas accessibles directement, mais par l'intermédiaire d'un pointeur passé à des fonctions, la seule information accessible à l'utilisateur est l'adresse du bloc de mémoire qui contient les données. Etant donné que la structure interne des données est inconnue de l'extérieur, on parle de données 'opaques' ou encore de 'type abstrait de données' (*abstract data type* ou *ADT*).

IV - Implémentation du type abstrait de donnée en C

L'implémentation de l'objet se fait par une structure de données.

```
struct objet
{
   int data;
};
```

Le principe de l'abstraction repose sur la compilation séparée. En effet, on va pouvoir fournir d'une part l'interface publique de l'objet sous la forme d'une définition incomplète de la structure

```
struct objet;
```

et d'autre part, dans le module qui va implémenter les fonctions (et pour commencer, les 'créateurs / destructeurs'), la définition 'privée' de la structure telle qu'elle a été décrite ci-dessus.

IV-A - Constructeur

La définition externe de la structure de données étant incomplète, il n'est évidemment pas possible de définir une instanciation statique de l'objet. Le rôle du constructeur est donc d'allouer dynamiquement une instance de l'objet et éventuellement d'initialiser ses membres à 0.

La méthode la plus évidente est d'utiliser malloc()/free() pour créer / détruire l'objet, mais il est parfaitement possible, dans les cas spécifiques (application embarquée critique, par exemple), d'utiliser un allocateur 'local' utilisant une zone de données 'préallouée' statique ou dynamique selon les besoins et possibilités du système).

```
/* fichier d'implementation */

#include <stdlib.h>

struct objet
{
   int data;
};

/* constructeur */
struct objet * objet_create (void)
{
   struct objet * self = malloc (sizeof *self);

   if (self != NULL)
   {
      /* initialisation a 0 */
      static const struct objet z = {0};
      *self = z;

      /* ajouter ici les eventuels complements...
       * - Autres initialisations
       * - Creation d'objets imbriques etc.
       */
   }

   return self;
}
```

Il est toujours possible de prévoir un mécanisme de paramètrage qui permet d'adapter l'objet à ses besoins.

IV-B - Destructeur

Il s'agit simplement de libérer le bloc alloué en utilisant la méthode réciproque à celle qui a servi à faire l'allocation (malloc() -> free()). Si il existe des objets imbriqués, ceux-ci doivent évidemment être détruits avant la libération de l'objet courant.

```
/* fichier d'implementation */

/* voir details ci-dessus... */

/* destructeur */
void objet_delete (struct objet * self)
{
   if (self != NULL)
   {
      /* ajouter ici les eventuels complements...
       * - Destruction d'objets imbriques etc.
       */
   }

   free (self);
}
```

IV-C - Fichier d'interface

Le fichier d'interface (*header*) comporte la définition incomplète de la structure et les prototypes des fonctions. Ce fichier est inclus dans le fichier d'implémentation de l'objet et dans tous les fichiers utilisateurs. Comme tout fichier d'en-tête, il est protégé contre les inclusions multiples.

```
/* fichier d'interface */

/* structure  */
struct objet;

/* fonctions  */
struct objet * objet_create (void);
void objet_delete (struct objet * self);
```

IV-D - Utilisation

Voici un exemple de création / destruction d'un TAD. Comme tout objet dynamique, sa création peut échouer. Il convient donc de tester la valeur de l'adresse retournée avant de l'utiliser. Une fois l'utilisation terminée, l'objet est détruit, et le pointeur est forcé à NULL, ce qui interdit la réutilisation d'un objet détruit. Penser à mettre à NULL les éventuels alias sur le bloc (autres pointeurs sur le même bloc).

```
/* fichier utilisateur */

#include "objet.h"

{
   struct objet * p_obj = objet_create();

   if (p_obj != NULL)
   {
      /* utilisation... */

      objet_delete (p_obj), p_obj = NULL;
   }
}
```

V - Exemple réel d'ADT

V-A - Spécification

Soit à réaliser un ADT permettant de créer ou utiliser une matrice (tableau à deux dimensions) de valeurs de type double.

Les dimensions minimales sont de 1 x 1. Les dimensions maximales ne sont pas imposées. Elles sont en fait limitées par l'architecture et l'état du système à un moment donné. L'ADT fournit les fonctions de construction/destruction habituelles, ainsi que des fonctions d'écriture/lecture à des coordonnées du tableau. Tout est mis en oeuvre pour assurer la sécurité et la fiabilité de l'ADT.

V-B - Conception

V-B-1 - Nommage

Le choix du nom de l'ADT est important. En effet, il doit à la fois être unique (au projet et/ou à la bibliothèque), sans ambiguïté et court (maximum 4 à 5 lettres) pour ne pas surcharger l'écriture. Il est utilisé comme préfixe à tous les identificateurs publics en relation avec l'ADT, notamment le type et les fonctions.

Le nom retenu est 'matd' (MATrice Double).

V-B-2 - Fonctions

Les fonctions sont

- matd_create()
- matd_delete()
- matd_put()
- matd_get()

V-B-3 - Interfaces détaillées

- matd_create(nb_colonnes, nb_lignes)
- matd_delete()
- matd_put(colonne, ligne, valeur)
- matd_get(colonne, ligne, valeur)

V-C - Réalisation en langage C

La réalisation se compose d'un fichier d'interface "matd.h" et d'un fichier d'implémentation "matd.c".

V-C-1 - Fichier d'interface

J'ai pour habitude d'utiliser un alias sur le type structure (typedef), ce qui a pour avantage d'alléger l'écriture. Ce n'est pas indispensable, et il est tout à fait possible d'utiliser directement 'struct xxx' si on préfère.

J'ai choisi le type size_t pour les dimensions et les coordonnées, car c'est le type le plus adapté à ce genre de valeur en C. La définition de size_t se trouve dans <stddef.h>

Le paramètre 'valeur' de matd_get() est de type pointeur sur double, car il s'agit de récupérer une valeur. On fournit donc à la fonction l'adresse de la variable de destination.

Les fonctions (autres que constructeur et destructeur) retournent un compte rendu d'exécution : 0 = OK 1 = erreur.

```
#ifndef H_MATD
#define H_MATD

/* matd.h */

#include <stddef.h>

/* types */
typedef struct matd matd_s;

/* fonctions */
matd_s *matd_create (size_t nb_colonnes, size_t nb_lignes);
void matd_delete (matd_s * self);

int matd_put (matd_s * self, size_t colonne, size_t ligne, double valeur);
int matd_get (matd_s * self, size_t colonne, size_t ligne, double *p_valeur);

#endif /* guard *
```

V-C-1-a - Note de codage

Il est possible d'aller plus loin dans l'abstraction en définissant un alias englobant la notion de pointeur. Je n'encourage pas cette pratique, car elle peut provoquer de la confusion, mais il faut savoir qu'elle existe. Appliquée à l'exemple en cours, le fichier d'interface se présentait ainsi (non testé):

```
#ifndef H_MATD
#define H_MATD

/* matd.h */

#include <stddef.h>

/* types */
typedef struct matd * matd_p;

/* fonctions */
matd_p matd_create (size_t nb_colonnes, size_t nb_lignes);
void matd_delete (matd_p self);

int matd_put (matd_p self, size_t colonne, size_t ligne, double valeur);
int matd_get (matd_p self, size_t colonne, size_t ligne, double *p_valeur);

#endif /* guard */
```

V-C-2 - Fichier d'implémentation

Je choisis une implémentation linéaire. Une implémentation par tableau de pointeurs sur tableaux dynamiques de doubles est aussi possible.

Allocation d'un tableau de colonnes x lignes doubles.

```
/* matd.c */

#include "matd.h"
#include <stdlib.h>

/* structures */
struct matd
```

```
{
   size_t nb_colonnes;
   size_t nb_lignes;
   double *tab;
};

/* fonctions privees */

static double *get_addr (matd_s * self, size_t colonne, size_t ligne)
{
   double *p = NULL;

   if (ligne < self->nb_lignes && colonne < self->nb_colonnes)
   {
      p = self->tab + (ligne * self->nb_colonnes) + colonne;
   }

   return p;
}

/* fonctions publiques */

matd_s *matd_create (size_t nb_colonnes, size_t nb_lignes)
{
   matd_s *self = NULL;

   if (nb_lignes > 0 && nb_colonnes > 0)
   {
      self = malloc (sizeof *self);

      if (self != NULL)
      {
         static const matd_s z =
         {0};
         *self = z;
         {
            size_t const size = nb_lignes * nb_colonnes;
            double *tab = malloc (sizeof *tab * size);

            if (tab != NULL)
            {
               size_t i;
               for (i = 0; i < size; i++)
               {
                  tab[i] = 0;
               }

               self->nb_lignes = nb_lignes;
               self->nb_colonnes = nb_colonnes;
               self->tab = tab;

            }
            else
            {
               free (self), self = NULL;
            }
         }
      }
   }
   return self;
}

void matd_delete (matd_s * self)
{
   if (self != NULL)
   {
      free (self->tab), self->tab = NULL;
   }
   free (self);
}

int matd_put (matd_s * self, size_t colonne, size_t ligne, double valeur)
```

```
{
   int err = 1;

   if (self != NULL)
   {
      double *p = get_addr (self, colonne, ligne);

      if (p != NULL)
      {
         *p = valeur;
         err = 0;
      }
   }
   return err;
}

int matd_get (matd_s * self, size_t colonne, size_t ligne, double *p_valeur)
{
   int err = 1;

   if (self != NULL)
   {
      double const *p = get_addr (self, colonne, ligne);

      if (p != NULL)
      {
         if (p_valeur != NULL)
         {
            *p_valeur = *p;
            err = 0;
         }
      }
   }
   return err;
}
```

V-C-3 - Remarque

L'initialisation de la structure est faite par la recopie d'une structure d'instance unique (définie 'static const') initialisée à 0. Pour faciliter la maintenance, j'ai écrit 'z = {0}'.

Certains compilateurs un peu sensibles peuvent signaler un warning à propos de valeurs d'initialisation manquantes. On peut ignorer ces warnings ou écrire l'initialisation complète : z = {0, 0, NULL}

V-D - Tests unitaires

Voici un test élémentaire qui permet de vérifier le bon fonctionnement de l'ADT. (syntaxe, compilation, comportement basique). Il manque des tests aux limites pour vérifier la stabilité de l'ADT dans les cas extrêmes.

```
/* test.c */

#include "matd.h"
#include <stdio.h>
#include <stdlib.h>

#define TERR(e)                                                 \
do                                                              \
{                                                               \
   if (e)                                                       \
   {                                                            \
      printf ("Error %d at %s:%d\n",e ,__FILE__, __LINE__); \
      exit (EXIT_FAILURE);                                      \
   }                                                            \
}                                                               \
while (0)
```

```c
int main (void)
{
   enum
   {
      NB_COL = 2,
      NB_LIN = 3,
      dummy
   };

   matd_s *m = matd_create (NB_COL, NB_LIN);

   if (m != NULL)
   {
      int err;

      err = matd_put (m, 0, 0, 0.0);
      TERR(err);
      err = matd_put (m, 0, 1, 0.1);
      TERR(err);
      err = matd_put (m, 0, 2, 0.2);
      TERR(err);
      err = matd_put (m, 1, 0, 1.0);
      TERR(err);
      err = matd_put (m, 1, 1, 1.1);
      TERR(err);
      err = matd_put (m, 1, 2, 1.2);
      TERR(err);

      {
         size_t i;

         for (i = 0; i < NB_COL; i++)
         {
            size_t j;

            for (j = 0; j < NB_LIN; j++)
            {
               double d;

               err = matd_get (m, i, j, &d);
               TERR(err);

               printf ("%.2f ", d);
            }
            printf ("\n");
         }
      }
      matd_delete (m), m = NULL;
   }

   return 0;
}
```

Résultats

```
0.00 0.10 0.20
1.00 1.10 1.20
```