Programmation orientée objet en C

1re partie - La méthodologie

Table des matières

L'auteur

CGi

L'article

Publié le 3 août 2005

I. Introduction▲

Le langage C n'est pas un langage orienté objet, mais nous allons voir dans ce document que si l'on structure notre code selon des règles strictes que l'on va établir et que l'on n'en déroge pas, on peut s'approcher de la programmation orientée objet.
Ce document sera accompagné d'un exemple. Une bibliothèque constituant une liste chaînée écrite selon les règles établies dans ce document. Cette liste chaînée sera une pile, fort semblable à celle vue dans l'article "La liste chaînée simple". Dans cet article, nous allons élaborer son code en l'imaginant comme un objet. Une initiation à la programmation objet sera la bienvenue pour la compréhension de ce document.
J'ai construit cet exemple en essayant de me rapprocher de la syntaxe du C++, ce qui crée quelques inconvénients dont vous pourrez prendre connaissance au fil de la lecture de ce document.

Les langages orientés objet possèdent des classes pour construire les objets. Les classes sont des modèles utilisés pour construire les objets. Au sens de la programmation, ce sont des types servant à instancier des objets.
Le langage C ne connaît pas les classes, nous nous orienterons donc vers un type proche : les structures (struct).
Dans les langages objet les classes possèdent des fonctions membres ou méthodes. Le C ne connaissant pas les méthodes, nous utiliserons donc les fonctions.
Le code interne des fonctions de l'exemple étant très proche de l'article "La liste chaînée simple" et n'ayant pas de rapport avec le sujet, il ne sera donc pas commenté dans ce document.

II. Règles d'écriture▲

On va tout d'abord énumérer les règles d'écriture que l'on va s'imposer.

- Toutes les fonctions publiques devront faire référence à une structure du langage C (struct). Concrètement, elles recevront toujours un pointeur sur la structure comme paramètre. Pour uniformiser les fonctions, ce pointeur sera toujours son premier paramètre. Cette structure représentera l'objet.

- Toutes les fonctions publiques devront avoir un pointeur de fonction associé parmi les membres de la structure. Ces pointeurs de fonction seront l'équivalent des fonctions membres du C++.

- À l'utilisation de la classe, on n'appellera jamais une fonction publique directement. On le fera à l'aide du pointeur de fonction que l'on a mis dans la structure.

- À l'utilisation de la classe, on n'accédera jamais aux membres (données) de la structure. On le fera toujours à l'aide des fonctions (par l'intermédiaire de leur pointeur de fonction).

- Pour éviter les redondances de noms de fonction, on préfixera leur nom du nom de la structure. Par exemple TPile_Push() avec le préfixe TPile, car elle fait référence au type stuct TPile.

- Les pointeurs de fonction membres de la structure auront par contre un nom court. Par exemple Push. Ils sont membres d'une structure, il n'y a plus de risque de redondance de nom.

- Les variables créées sur la base de cette structure le seront avec une fonction spéciale que l'on écrira toujours de la même façon : Préfixe_Create pour la création d'objets sur la pile et New_Préfixe pour la création d'objets sur le tas. Ce qui fera : TPile_Create et New_TPile dans l'exemple de ce document. Cette fonction n'aura pas de pointeur correspondant dans la structure. Ceci est normal, quand on l'appelle, l'instance de la structure n'est pas encore créée. Ce sera donc la seule fonction de la classe appelée directement. C'est elle qui crée l'objet, elle devra donc être appelée avant toute utilisation d'un objet. Cette fonction est l'équivalent du constructeur des langages orientés objet.

- La mémoire allouée en interne par la classe sera libérée par une fonction spéciale quand l'objet ne sera plus utile. On écrira toujours cette fonction sous la forme Préfixe_Free soit TPile_Free pour l'exemple. Cette fonction est l'équivalent du destructeur des langages orientés objet.

- Chaque classe sera mise dans un fichier séparé de même que leurs déclarations dans un fichier entête séparé.

Ces conventions sont celles que je me suis données pour écrire cet article, il en existe d'autres, plus normalisées.

III. Mise en œuvre▲

Les données membres ou attributs de l'objet seront donc constitués par les membres d'une structure :

Sélectionnez

```
typedef struct Tpile
        {
                int Nombre;
                struct Titem *Top;
        } Tpile ;
```

Dans l'exemple nous en avons deux : un entier qui contiendra le nombre d'éléments de la pile et un pointeur sur le sommet de la pile.
Les fonctions (membres) doivent accéder à différentes instances d'objets (structures dans notre cas). Elles recevront un pointeur sur ces structures comme paramètre. Ce pointeur est l'équivalent du pointeur **this** des objets en C++. Comme nous l'avons dit, nous mettrons ce pointeur (This) en leur premier paramètre. Voici ci-dessous un exemple de fonction (membres) TPile_Push qui reçoit donc le pointeur This comme premier paramètre :

Sélectionnez

```
int TPile_Push(TPile *This, int Val)
{
        Titem *new_item = malloc(sizeof(Titem));
        if(!new_item) return ITEM_ALLOC_ERROR;
        new_item->Value = Val;
        new_item->prec = This->Top;
        This->Top = new_item;
        This->Nombre++;
        return 0;
}
```

Nous nous étions donné comme règle d'appeler ces fonctions par l'intermédiaire de pointeurs de fonction membre de la structure. Ajoutons ces pointeurs de fonction à la structure :

Sélectionnez

```
typedef struct TPile
        {
                int(*Push)(struct TPile*, int);
                int(*Pop)(struct TPile*);
                void(*Clear)(struct TPile*);
                void(*Free)(struct TPile*);
                int(*Length)(struct TPile*);
                void(*View)(struct TPile*);

                int Nombre;
                struct Titem *Top;
        } TPile ;
```

Mais avant d'utiliser ces fonctions (membres), il est impératif de créer et d'initialiser l'objet. Ce que nous ferons en une seule opération en utilisant une fonction qui créera l'objet (structure), initialisera ses membres et retournera l'objet.
Dans la pratique on va en créer deux, une pour l'initialisation d'un objet de type variable automatique et l'autre pour la création d'un objet de type variable dynamique. On les appellera des constructeurs.
Le premier retourne la copie d'un objet :

Sélectionnez

```
TPile TPile_Create()
{
       TPile This;
       TPile_Init(&This);
       This.Free = TPile_Free;
       return This;
}
```

Le second retourne un pointeur sur un objet créé dynamiquement :

Sélectionnez

```
TPile* New_TPile()
{
       TPile *This = malloc(sizeof(TPile));
       if(!This) return NULL;
       TPile_Init(This);
       This->Free = TPile_New_Free;
       return This;
}
```

Les membres de la structure sont initialisés dans une fonction commune TPile_Init :

Sélectionnez

```
static void TPile_Init(TPile *This)
{
	This->Push = TPile_Push;
	This->Pop = TPile_Pop;
	This->Clear = TPile_Clear;
	This->Length = TPile_Length;
	This->View = TPile_View;
	This->Nombre = 0;
	This->Top = NULL;
}
```

Cette fonction contient les initialisations communes aux deux constructeurs. Elle est donc appelée dans chaque constructeur. Les pointeurs de fonction sont affectés avec l'adresse des fonctions qui leur sont associées et les membres avec leurs valeurs. Le pointeur de fonction sur le destructeur sera initialisé dans les constructeurs, car il sera différent selon que l'objet est créé dynamiquement ou non. Nous aurons donc deux destructeurs.

Sélectionnez

```
void TPile_Free(TPile *This)
{
	TPile_Clear(This);
	puts("Destruction de la pile statique.\n");
}
/*****************************************************************************/

void TPile_New_Free(TPile *This)
{
	if(This) TPile_Clear(This);
	free(This);
	puts("Destruction de la pile dynamique.\n");
}
```

Nous n'aurons pas le souci de savoir lequel on doit appeler, car le pointeur de fonction lui étant destiné pointera sur le bon destructeur. Il est initialisé dans le constructeur.
Vous avez dû remarquer que j'ai mis la fonction TPile_Init en static et que je l'ai déclarée dans le fichier d'implémentation. La raison est que je ne veux pas que l'on y accède de l'extérieur. C'est un moyen de la rendre privée. Ce n'est malheureusement pas aussi simple pour les données membres. Du moins avec la méthode que nous employons dans ce document. On aurait pu les cacher en mettant l'implémentation de la structure dans le fichier de code "pile.c". Mais si on fait cela les pointeurs de fonction qui s'y trouvent, ne seront plus accessibles. On aurait pu aussi déclarer toutes les fonctions dans le fichier d'implémentation et les mettre static, car on ne les appelle jamais directement de l'extérieur. Nous ne le ferons pas, car nous ne pourrions plus dériver notre classe ultérieurement (ceci sera développé dans la deuxième partie de ce document). L'utilisation des pointeurs de fonction peut vous paraître inutile. Ils nous seront utiles pour l'héritage et le polymorphisme.

IV. Utilisation de la classe▲

L'utilisation en est extrêmement simple avec une syntaxe proche du C++. Voici une création d'un objet de type TPile en tant que variable locale :

Sélectionnez

```
	TPile MaPile = TPile_Create();

	MaPile.Push(&MaPile, 10);
	MaPile.Push(&MaPile, 25);

	MaPile.View(&MaPile);

	MaPile.Free(&MaPile);
```

Syntaxe proche du C++ mis à part que pour une variable locale (automatique), il faut appeler le constructeur et le destructeur implicitement. Et ci-dessous la création d'un objet de type TPile en tant que variable dynamique :

Sélectionnez

```
	TPile *MaPile = New_TPile();

	MaPile->Push(MaPile, 10);
	MaPile->Push(MaPile, 25);

	MaPile->View(MaPile);

	MaPile->Free(MaPile);
```

Autre différence avec le C++ le pointeur (this) sur la structure doit être passé implicitement comme paramètre. En C++ il est caché.
On peut remarquer dans les deux exemples d'utilisation de la classe que le code est aussi simple pour l'utilisation d'un objet créé dynamiquement que créé statiquement. Ce qui fait que bien souvent l'on préférera la création dynamique qui est moins lourde en ressources machine. On peut donc ne pas définir le constructeur statique et n'utiliser que des objets créés dynamiquement.
On aurait tout à fait pu se passer des pointeurs de fonction dans la structure et appeler directement les fonctions lors de l'utilisation. Ces pointeurs de fonction ont de plus deux inconvénients. C'est qu'ils alourdissent la taille de la structure et qu'ils obligent à rendre l'interface de la structure publique (définie dans le fichier entête). Mais la façon dont nous avons procédé nous sera utile pour la deuxième partie de ce document dont je vous invite à voir la suite ici : L'héritage.
Voilà pour cette première partie qui est un bon exercice pour la manipulation des pointeurs et des pointeurs de fonction.

V. Codes source de l'exemple▲

Pile.h :

Sélectionnez

```
#ifndef CGI_TPILE_H
#define CGI_TPILE_H

#define ITEM_ALLOC_ERROR  1
#define PILE_EMPTY       -1

#ifdef __cplusplus
  extern "C" {
#endif

/*  Structure représantant un élément de la pile. */
typedef struct Titem
        {
                int Value;
                struct Titem *prec;
        } Titem ;

/*  Structure représantant l'objet pile. */
typedef struct TPile
        {
            /*  Les pointeurs sur fonctions (membres) :                      */
            /*  Push empile une valeur sur la pile.
                retourne ITEM_ALLOC_ERROR si l'allocation a échouée sinon 0
                int Push(TPile, int)                                         */
                int(*Push)(struct TPile*, int);

            /*  Pop retire la dernière valeur empilé sur la pile.
                retourne PILE_EMPTY si la pile est vide.
                int Pop(TPile)                                               */
                int(*Pop)(struct TPile*);

            /*  Clear vide la pile.
                void Clear(TPile)                                            */
                void(*Clear)(struct TPile*);

            /*  Free détruit la pile.
                void Free(TPile)                                             */
                void(*Free)(struct TPile*);

            /*  Lenght retourne le nombre d'élément de la pile.
                int Length(TPile)                                            */
                int(*Length)(struct TPile*);

            /*  View affiche la totalité de la pile en commençant par le sommet.
                void View(TPile)                                             */
                void(*View)(struct TPile*);

            /*  Les données membres :                                        */
                int Nombre;
                struct Titem *Top;

        } TPile ;


/*  Pile_Create crée une pile. */
TPile TPile_Create(void);

TPile* New_TPile(void);

int TPile_Push(TPile*, int);

int TPile_Pop(TPile*);

void TPile_Clear(TPile*);

int TPile_Length(TPile*);

void TPile_View(TPile*);

void TPile_Free(TPile*);

void TPile_New_Free(TPile*);

#ifdef __cplusplus
}
#endif

#endif
```

Pile.c :

Sélectionnez

```
#include<stdlib.h>
#include<stdio.h>

#include "Pile.h"

static TPile* TPile_Init(TPile*);

TPile TPile_Create()
{
       TPile This;
       TPile_Init(&This);
       This.Free = TPile_Free;
       return This;
}
/*****************************************************************************/

TPile* New_TPile()
{
```

```
        TPile *This = malloc(sizeof(TPile));
        if(!This) return NULL;
        TPile_Init(This);
        This->Free = TPile_New_Free;
        return This;
}
/*****************************************************************************/

static void TPile_Init(TPile *This)
{
        This->Push = TPile_Push;
        This->Pop = TPile_Pop;
        This->Clear = TPile_Clear;
        This->Length = TPile_Length;
        This->View = TPile_View;
        This->Nombre = 0;
        This->Top = NULL;
}
/*****************************************************************************/

int TPile_Push(TPile *This, int Val)
{
        Titem *new_item = malloc(sizeof(Titem));
        if(!new_item) return ITEM_ALLOC_ERROR;
        new_item->Value = Val;
        new_item->prec = This->Top;
        This->Top = new_item;
        This->Nombre++;
        return 0;
}
/*****************************************************************************/

int TPile_Pop(TPile *This)
{
        int Val;
        Titem *tmp;
        if(!This->Top) return PILE_EMPTY;
        tmp = This->Top->prec;
        Val = This->Top->Value;
        free(This->Top);
        This->Top = tmp;
        This->Nombre--;
        return Val;
}
/*****************************************************************************/

void TPile_Clear(TPile *This)
{
        Titem *tmp;
        while(This->Top)
          {
             tmp = This->Top->prec;
             free(This->Top);
             This->Top = tmp;
          }
        This->Nombre = 0;
}
/*****************************************************************************/

int TPile_Length(TPile *This)
{
        return This->Nombre;
}
/*****************************************************************************/

void TPile_View(TPile *This)
{
       Titem *tmp = This->Top;
       while(tmp)
          {
             printf("%d\n",tmp->Value);
             tmp = (*tmp).prec;
          }
}
/*****************************************************************************/

void TPile_Free(TPile *This)
{
        TPile_Clear(This);
        puts("Destruction de la pile static.\n");
}
/*****************************************************************************/

void TPile_New_Free(TPile *This)
{
        if(This) TPile_Clear(This);
        free(This);
        puts("Destruction de la pile dynamique.\n");
}
```

Voici un exemple d'utilisation de la pile que nous venons de construire.

main.c :

Sélectionnez

```
#include <stdlib.h>
#include <stdio.h>

#include "Pile.h"

int main()
{
        TPile MaPile = TPile_Create();

        MaPile.Push(&MaPile, 10);
```

```
        MaPile.Push(&MaPile, 25);
        MaPile.Push(&MaPile, 33);
        MaPile.Push(&MaPile, 12);

        puts("Affichage de la pile :");
        MaPile.View(&MaPile);
        puts("------");

        printf("Nb d'elements : %d\n",MaPile.Length(&MaPile));
        puts("------");

        puts("Deux valeurs soutirees de la pile :");
        printf("%d\n",MaPile.Pop(&MaPile));
        printf("%d\n",MaPile.Pop(&MaPile));
        puts("------");

        puts("Affichage de la pile :");
        MaPile.View(&MaPile);
        puts("------");

        MaPile.Clear(&MaPile);
        MaPile.Push(&MaPile, 18);

        puts("Affichage de la pile apres vidage et ajout d'une valeur :");
        MaPile.View(&MaPile);
        puts("------\n");

        MaPile.Free(&MaPile);

        return 0;
}
```

Voici un exemple d'utilisation de la pile que nous venons de construire. Avec une création dynamique de l'objet :

main.c :

Sélectionnez

```
#include <stdlib.h>
#include <stdio.h>

#include "Pile.h"

int main()
{
        TPile *MaPile = New_TPile();

        MaPile->Push(MaPile, 10);
        MaPile->Push(MaPile, 25);
        MaPile->Push(MaPile, 33);
        MaPile->Push(MaPile, 12);

        puts("Affichage de la pile :");
        MaPile->View(MaPile);
        puts("------");

        printf("Nb d'elements : %d\n",MaPile->Length(MaPile));
        puts("------");

        puts("Deux valeurs soutirees de la pile :");
        printf("%d\n",MaPile->Pop(MaPile));
        printf("%d\n",MaPile->Pop(MaPile));
        puts("------");

        puts("Affichage de la pile :");
        MaPile->View(MaPile);
        puts("------");

        MaPile->Clear(MaPile);
        MaPile->Push(MaPile, 18);

        puts("Affichage de la pile apres vidage et ajout d'une valeur :");
        MaPile->View(MaPile);
        puts("------\n");

        MaPile->Free(MaPile);
        MaPile = NULL;

#ifdef __WIN32__
        system("PAUSE");
#endif
        return 0;
}
```

Suite : L'héritage.

Bonne lecture,
CGi.