Implémentation guidée des concepts
Objets : 2ème partie
Implémentation de la Généricité

La première idée pour obtenir de la généricité en C consiste à faire tout le travail soi-même :

En mettant le type dans le nom de la fonction, on a des pseudo-templates.

```
int   maxint  (int   a, int   b) { return ((a) >= (b) ? (a) : (b)); }

float maxfloat(float a, float b) { return ((a) >= (b) ? (a) : (b)); }
```

Il s'agit d'un travail à base de copier/coller, rechercher/remplacer, très laborieux dès qu'il prend un peu d'ampleur, et qu'il faut recommencer dès qu'on veut instancier la fonction avec un nouveau type. La perte de temps est considérable.

Une autre solution se profile avec l'utilisation d'une macro.
Exemple :

```
#define MAX(A,B) ((A) >= (B) ? (A) : (B))
```

Cela dit, on connaît les nombreux problèmes que cela pose :

- augmentation non négligeable de la taille du code,
- pas de contrôle des types,
- et surtout, *erreur* dans des cas comme `MAX(i++, 2)` , où le remplacement brutal de A et B pose problème.

Il n'est donc pas envisageable de procéder de cette façon.

C'est pourtant dans les macros qu'on trouvera une solution satisfaisante à ce problème de généricité. Il suffit de combiner les deux solutions précédentes : à l'aide d'une macro, on génère *automatiquement* les différentes « instances » de la fonction.

Exemple:

```
      #define DEFINIR_MAX(TYPE) \

      TYPE max##TYPE(TYPE a, TYPE b) {return a >=b ? a : b;}
```

A noter que le *##* concatène le nom, dans le cas ci-dessus *max* et *TYPE* seront concaténés pour donner le nom de la fonction.

Le programme suivant est maintenant possible :

```
/* test_max.c */

#include <stdio.h>

#define DEFINIR_MAX(TYPE) \
TYPE max##TYPE(TYPE a, TYPE b) {return a >=b ? a : b;}

DEFINIR_MAX(int)      /* maxint(int a, int b) est maintenant disponible */
DEFINIR_MAX(float)    /* maxfloat(float a, float b) est maintenant disponible */


int main(int argc, char* argv[])
{
   printf("%d\n", maxint(1,2));
   printf("%f\n", maxfloat(1,2));
   return 0;
}
```

Une fois ce principe compris, il s'agit d'implémenter une pile générique avec ce principe.

Avec ce qui précède, rien n'empêche de créer toute une bibliothèque générique en s'inspirant soit de ce qui a été proposé dans l'ouvrage de Gourdin sur la programmation par objets en C (1990 chez Lavoisier et Eyrolles) soit des travaux du Dr. Pierre Chatelier ancien ISIMA F2, auteur d'ouvrages et d'un site sur la métaprogrammation Template en C++ suite à l'ancien cours d'Objet Avancé que je proposais et au TP sélectionné pour proposer ce tutorial.

On peut aussi s'aider de l'option –E du compilateur (expand) pour voir le code intermédiaire après traitement par le préprocesseur. Cette option est très utile pour la mise au point du TP.

Nous allons implémenter pour l'exemple une pile très simple, de taille maximale *N* fixée, et codée de la manière suivante :

> Un tableau de *N* cases, appelé *pile* qu'on alloue si la pile doit servir
> Un pointeur *top* sur le sommet de la pile (*i.e.* une case du tableau)

Si la pile est vide, alors *top* contient `NULL`.

De prime abord, on écrit un code ressemblant à :

```
#define TAILLE_MAX 5
typedef enum{FAUX=0, VRAI=1} bool ;

#define DECLARER_PILE(TYPE)                                    \
typedef struct Pile##TYPE                                      \
{                                                              \
      TYPE  pile[TAILLE_MAX];                                  \
      TYPE* top ;                                              \
}Pile##TYPE##_t ;                                              \
                                                               \
/* empile la valeur */                                         \
void empiler##TYPE(TYPE valeur, struct Pile##TYPE* this) ;     \
                                                               \
/* dépile en retournant le sommet */                          \
TYPE depiler##TYPE(struct Pile##TYPE* this) ;                  \
```

```
                                                                \
// Prédicat : la pile est-elle vide ?                           \
bool estVide##TYPE(struct Pile##TYPE* this) ;                   \
                                                                \
// renvoie le sommet de pile                                    \
TYPE sommet##TYPE(struct Pile##TYPE* this) ;

#define IMPLEMENTER_PILE(TYPE)                                  \
//on met ici le code des "méthodes" de la pile                  \
```

Une séparation est proposée entre la macro 'interface' qui fait la déclaration du type et la macro qui est chargée de l'implémentation. Au-dessus du programme principal (attention à ne pas appeler ces macro-instructions dans le main), on pourra maintenant écrire :

```
DECLARER_PILE(int)
DECLARER_PILE(float)
IMPLEMENTER_PILE(int)
IMPLEMENTER_PILE(float)
```

Puis dans une fonction, quelque chose comme :

```
Pileint_t   pi ;
Pilefloat_t pf ;

pi.top = NULL;
empilerint(5, &pi);
depilerint(&pi);

pf.top = NULL;
empilerfloat(5, &pf);
depilerfloat(&pf);
```

Remarque : les problèmes du piège de l'initialisation du pointeur *top*, et les contrôles d'erreur, sont abordés dans la suite.

Même si nous avons maintenant une pile générique assez pratique, on s'aperçoit qu'il va être laborieux d'avoir à appeler *empilerint* ou *empilerfloat* selon la pile qu'on utilise.

Il va plutôt être intéressant d'utiliser le principe d'encapsulation comme dans le premier TP, avec des méta-classes, pour pouvoir écrire plutôt :

```
Pileint_t   pi ;
Pilefloat_t pf ;

LaMetaPileint.Construire(&pi);
pi.myClass->empiler(5, &pi);
pi.myClass->depiler(&pi);

LaMetaPilefloat.Construire(&pf);
pf.myClass->empiler(5, &pf);
pf.myClass->depiler(&pf);

/* destructions inutiles (tout est statique) */
```

On y gagne énormément en facilité d'utilisation (pour l'initialisation *via* le constructeur comme pour l'appel des méthodes).

De plus, la nécessité d'utiliser un constructeur nous permet de complexifier la pile de façon transparente : on va maintenant lui allouer sa taille à la construction en lui passant en paramètre la taille maximum. Le *malloc()* se faisant dans le constructeur et le *free()* dans le destructeur (qui devient alors nécessaire) , l'utilisateur ne les voit même pas.

On peut alors écrire:

```
Pileint_t   pi ;
Pilefloat_t pf ;

LaMetaPileint.Construire(&pi, 2);   /* on peut empiler 2 éléments */
                                    /* au maximum */
pi.myClass->empiler(5, &pi);
pi.myClass->depiler(&pi);
LaMetaPileint.Detruire(&pi);        /* nécessaire pour le free() */

LaMetaPilefloat.Construire(&pf, 3); /* on peut empiler 3 éléments */
                                    /* au maximum */
pf.myClass->empiler(5, &pf);
pf.myClass->depiler(&pf);
LaMetaPilefloat.Detruire(&pf);      /* nécessaire pour le free() */
```

Le code source que nous fournissons au final est aussi plus complet pour la gestion des erreurs.

Problèmes éventuels

La méthode que nous avons utilisée pour la généricité, à base de macros, pose cependant quelques problèmes mineurs :

1) rien n'empêche l'utilisateur d'appeler plusieurs fois `DECLARER_PILE(TYPE)` ou `IMPLEMENTER_PILE(TYPE)` avec le même `TYPE`, ce qui se traduira par des redéfinitions.

   Il peut aussi se tromper et utiliser `IMPLEMENTER_PILE` avant `DECLARER_PILE`.

2) On ne peut écrire `DECLARER_PILE(int *)`.

   Il faut faire :

   ```
   typedef int* pInt;
   DECLARER_PILE(pInt) ;
   ```

   Sinon la macro générera un code non compilable à cause des espaces et de l'étoile.

Solutions :

1) Le problème n'est qu'apparent puisque c'est le compilateur, ou l'éditeur de liens, qui indiquera une erreur (*variable indéfinie*, ou *duplicate symbol*). On aurait cependant aimé (mais on ne peut pas) que `DECLARER_PILE()` et `IMPLEMENTER_PILE()` contiennent des directives de compilation sécurisantes (comme pour les inclusions de fichier *.h*) du genre :

```
#define DECLARER_PILE(TYPE)          \
   #ifndef __DECLARERPILE##TYPE##__ \
   #define   DECLARERPILE##TYPE##   \
   /* . . . code ici . . . */       \
   #endif                           \
```

et

```
#define IMPLEMENTER_PILE(TYPE)                          \
   #ifndef __IMPLEMENTERPILE##TYPE##__                  \
      #ifndef __DECLARERPILE##TYPE##__                  \
            #error "Vous devez d'abord declarer "       \
            "l'interface via DECLARERPILE("#TYPE        \
            ")"                                         \
      #endif                                            \

#define __IMPLEMENTERPILE##TYPE##__                     \
   /* . . . code ici . . . */                           \
#endif                                                  \
```

Malheureusement, une macro ne peut définir une autre macro, et le code ci-dessus ne passe pas l'étape de pré-processing.

2) Ce problème est à souligner, mais nous ne connaissons pas de solution plus simple. Nous aurions pu transformer les macros de la façon suivante:

```
#define DECLARERPILE(TYPE_REEL, TYPE)     \
      typedef TYPE_REEL TYPE;             \
      /* . . . code ici . . . */          \

#define IMPLEMENTERPILE(TYPE)             \
      /* . . . code ici . . . */          \
```

On peut alors écrire :

```
      DECLARER_PILE(int, Int)
      IMPLEMENTER_PILE(Int)
```

Cela permet d'utiliser une *PileInt_t*, une *MetaPileInt*, qu'on peut préférer pour des raisons "cosmétiques" aux versions sans majuscules ; mais surtout, on peut écrire :

```
      DECLARER_PILE(int *, pInt)
      IMPLEMENTER_PILE(pInt)
```

Pour faire une pile de pointeurs sans écrire soi-même le *typedef*.

Il y a cependant des inconvénients non négligeables:

> Si on écrit `DECLARER_PILE(int, int)`, par exemple, cela génère un warning à la compilation. Il faut donc faire attention à rester cohérent dans les noms de types entre
>
> `DECLARER_PILE` et `IMPLEMENTER_PILE`.
>
> En effet, si l'on écrit

```
DECLARER_PILE(int, Int)
```

On est obligé d'utiliser

```
IMPLEMENTER_PILE(Int)
```

et non

```
IMPLEMENTER_PILE(int)   /* erreur difficile à voir */
                        /* en lecture rapide */
```

Le fait d'avoir des synonymes est source potentielle d'erreurs, et même si elles seraient là plutôt facilement repérables et corrigeables, autant les éviter. En forçant l'utilisateur à faire le *typedef* lui-même, on est plus sûr qu'il pensera à utiliser le nouveau nom.