# TP5 : Pile de génériques et ArrayList

## 1 Mesurer un temps de calcul

Commencez par regarder, dans le dossier *Partie 1* des fichiers du TP, le fichier *mypile_u32.c* : vous y trouverez le code permettant de gérer une pile d'entiers non signés 32 bits.

1. Ecrivez un fichier **main.c** qui testera cette pile : vous créerez une pile, ajouterez 3 éléments, et dépilerez et afficherez tout le contenu de la pile.

2. Le fichier *main.c* qui est joint permet de lancer des empilement et dépilement aléatoires dans une pile (200 millions de fois) afin de calculer le temps nécessaire à ces opérations. Après avoir compilé votre programme avec l'option "-O3", utilisez la commande time pour mesurer le temps de calcul de ce programme (et notez ce temps).

3. Que devez-vous faire pour que créer une nouvelle pile qui ne stocke plus des entiers, mais des doubles ? Testez votre solution en la mettant en place et en la testant dans un programme.

## 2 Une ArrayList

Dans cet exercice, nous verrons comment construire une ArrayList en se basant sur la bibliothèque de liste précédemment construite.

1. Modifiez les deux fonctions **Pile_u32_pop** et **Pile_u32_push**, afin qu'elles ne prennent plus en paramètre ou valeur de retour un entier, mais le **maillon_u32** qui contient l'entier. Vous devrez probablement modifier la fonction **Pile_u32_free**.

2. Pour faire une ArrayList d'entiers 32 bits non signés, vous devez proposer une structure avec deux piles : une pile de **maillon_u32** contenant des données utiles, et une pile de **maillon_u32** libres et que l'on pourra utiliser pour ajouter de nouvelles valeurs.

```
typedef struct arraylist_u32
{
    pile_u32* occupes;
    pile_u32* libres;
} arraylist_u32;
```

   Ces deux piles serviront à stocker les données dans l'Arraylist : la pile **occupes** contiendra les données utiles, la pile **libres** contiendra les maillons libres que l'on pourra utiliser pour stocker des données dans l'Arraylist.
   Le but sera de pré-allouer des maillons libres dans la liste **libres**, et de les utiliser dès que l'on devra ajouter des éléments à la liste **occupés**. Inversement, si on retire un élément de la liste **occupes**, il faudra récupérer le maillon qui contenait cet élément et le replacer dans la liste **libres**.
   Vous devrez proposer plusieurs fonctions :

   — La fonction **Arraylist_u32_new** qui créera une nouvelle **Arraylist**, et initialisera chacune de ses sous listes.
   — La fonction **Arraylist_u32_isEmpty** qui testera si la structure contient des données (tester la sous liste **occupes** pour voir si elle est vide).
   — La fonction **Arraylist_u32_pop** récupérera un maillon de la sous liste **occupes**, récupérera son entier, et replacera le maillon dans la liste **libres**.

— La fonction **Arraylist_u32_push** récupérera un maillon de la sous liste **libres**, y écrira une donnée entière dedans, et replacera la maillon dans la liste **occupes**. Tout l'intérêt d'une Arraylist vient du problème qui se pose si la sous liste **libres** est vide. Dans ce cas, il faut créer un tableau de **maillon_u32**, parcourir le tableau case par case pour lier chaque maillon au suivant, et placer la tête de la liste **libres** sur le début du tableau.

En général, on choisit de créer un tableau de taille égale à deux fois la taille actuelle de l'Arraylist plus un.

3. Reprenez le programme réalisé en tout début de TP (où vous ajoutiez trois éléments à la pile, puis les retiriez), et modifiez-le afin de fonctionner avec une Arraylist. Que constatez-vous lorsque vous exécutez votre programme avec valgrind : avez-vous des fuites mémoire ?

# 3 Libération mémoire de l'Arraylist

On va chercher à résoudre le problème précédent concernant notre Arraylist : nous n'avons aucune fonction de libération mémoire de l'Arraylist.

1. Un moyen pour libérer l'Arraylist de la mémoire serait de vider et libérer les listes **libres** et **occupes** de l'Arraylist. Ecrivez une fonction **Arraylist_u32_free** qui libérera l'Arraylist de la mémoire en vidant chacune de ses listes avec **Pile_u32_free**.

2. Pour libérer l'Arraylist de la mémoire, il faut donc conserver les adresses de chaque tableau de maillons alloué par la fonction **Arraylist_u32_push**. Nous devons nous poser deux questions pour résoudre ce problème :

   — Quel est le type de données associé à un tableau de maillons ?

   — Quelle structure de données utiliser pour stocker ce type de données ?

   Implémentez cette solution, modifiez les fonctions nécessaires, proposez une fonction **Arraylist_u32_free** qui ne provoque pas d'erreur et testez la sur un exemple (utilisez valgrind pour vérifier que tout est libéré).

   On modifie quelques fonctions concernant l'arraylist. Tout d'abord, la structure même d'une arraylist changera pour intégrer cette nouvelle pile :

```c
typedef struct arraylist_u32
{
    pile_u32* occupes;
    pile_u32* libres;
    pile_mstar* tableaux;
} arraylist_u32;
```

   La fonction de création d'une nouvelle arraylist change pour intégrer cette nouvelle pile :

```c
arraylist_u32 *Arraylist_u32_new()
{
    //... On conserve tout le code de la fonction, et on ajoute :
    a->tableaux = Pile_mstar_new();

    if(a->tableaux==NULL)
    {
        fprintf(stderr, "Arraylist_u32_new : Erreur Allocation Memoire d'une sous liste.\n");
        free(a);
        return(NULL);
    }

    return a;
}
```

   Ensuite, nous devons évidemment modifier la fonction d'ajout d'élément dans la liste :

```c
void Arraylist_u32_push(arraylist_u32 *a, uint32_t d)
{
    //... On garde tout le debut de la fonction

    if( Pile_u32_isEmpty(a->libres) )
    {
        //...On garde tout le code de ce bloc, et on ajoute a la fin :

        Pile_mstar_push(a->tableaux, &(tab[0]));
    }

    //...On garde toute la fin de la fonction
}
```

Enfin, nous pouvons proposer une fonction de libération mémoire de l'arraylist :

```c
void Arraylist_u32_free(arraylist_u32 *a)
{
    while(! Pile_mstar_isEmpty(a->tableaux))
    {
        free(Pile_mstar_pop(a->tableaux));
    }
    free(a);
}
```

# 4 Une bibliothèque de pile "adaptable" (générique)

Dans le dossier *Partie 2* des fichiers du TP, regardez les fichiers *main.c*, *mypile.c* et *mypile.h*. Vous trouverez une bibliothèque de fonctions sur une pile qui ne contient pas de données dans le maillon. Les données sont en fait stockées en plus, à côté de chaque maillon, mais pas dans le maillon en lui-même.

Ce système va nous permettre de créer des piles qui pourront stocker différents types de données.

Regardez la fonction de création d'un nouveau maillon, **Maillon_new** : cette fonction alloue suffisamment de mémoire pour un maillon et une donnée d'une certaine taille. Cette taille est enregistrée dans la structure de pile dans la fonction **Pile_new**.

Pour utiliser la pile, il faut ensuite créer trois fonctions qui "surchargeront" les fonctions de la pile : une fonction de création de pile, une fonction d'empilement et une fonction de dépilement. Vous avez un aperçu de trois de ces fonctions dans le fichier *main.c* : **mypile_uint32_new**, **mypile_uint32_pop** et **mypile_uint32_push**. Ces fonctions permettent de manipuler, à travers la bibliothèque *mypile*, une pile stockant des entiers 32 bits non signés.

Regardez attentivement les fonctions **mypile_uint32_pop** et **mypile_uint32_push** afin de comprendre comment les données sont ajoutées aux maillons de la pile.

**Questions**

1. A partir de ce que vous avez lu, proposez trois fonctions permettant de générer la création (**mypile_double_new**), l'empilement (**mypile_double_push**) et le dépilement (**mypile_double_pop**) d'une pile stockant des double, en utilisant la bibliothèque *pile.h*. Testez votre code avec la fonction main ci-dessous (qui devrait compiler et fonctionner une fois les trois fonctions précédemment citées créées) :

```c
int main(int argc, const char * argv[])
{
    pile *p = mypile_double_new();
    mypile_double_push(p, 6.2);
    mypile_double_push(p, 9.8);
    mypile_double_push(p, 3.1);
    mypile_double_push(p, 0.9);
    printf("%f\n", mypile_double_pop(p));
```

```
    mypile_double_push(p, 12.9);
    printf("%f\n", mypile_double_pop(p));
    printf("%f\n", mypile_double_pop(p));
    printf("%f\n", mypile_double_pop(p));
    printf("%f\n", mypile_double_pop(p));

    return(0);
}
```

2. Créez une structure **doublon** :

```
typedef struct doublon
{
    uint32_t i;
    double d;
} doublon;
```

Créez trois fonctions **mypile_doublon_new**, **mypile_doublon_push** et **mypile_doublon_pop** permettant de gérer, au travers de la bibliothèque *pile.h*, une pile de **doublon**. Testez votre code avec le **main** ci-dessous :

```
int main(int argc, const char * argv[])
{
    pile *p = mypile_doublon_new();
    doublon r = {6, 6.2};
    mypile_doublon_push(p, r);
    r.i = 9; r.d = 9.8;
    mypile_doublon_push(p, r);
    r.i = 3; r.d = 3.2;
    mypile_doublon_push(p, r);
    r.i = 0; r.d = 0.9;
    mypile_doublon_push(p, r);
    r = mypile_doublon_pop(p);
    printf("%u %f\n", r.i, r.d);
    r.i = 12; r.d = 12.4;
    mypile_doublon_push(p, r);
    r = mypile_doublon_pop(p);
    printf("%u %f\n", r.i, r.d);
    r = mypile_doublon_pop(p);
    printf("%u %f\n", r.i, r.d);
    r = mypile_doublon_pop(p);
    printf("%u %f\n", r.i, r.d);
    r = mypile_doublon_pop(p);
    printf("%u %f\n", r.i, r.d);

    return(0);
}
```

On se rend compte à partir d'ici que beaucoup de lignes de code se ressemblent. Par exemple, pour créer une nouvelle pile, les fonctions **mypile_uint32_new**, **mypile_double_new** et **mypile_doublon_new** sont quasiment les même.

Nous allons modifier leur appel en faisant une macro. Une macro est un ensemble de lignes de code qui sera associé à un mot clef. Lorsque nous écrirons ce mot clef dans notre code, à la phase de compilation, les lignes de code correspondant au mot clef viendront remplacer ce dernier.

Pour ce faire, dans le fichier *mypile.h*, après les include, on écrira :

```
#define MYPILE_NEW(datatype) \
    Pile_new(sizeof(datatype));
```

Note les antislash après chaque nouvelle ligne... Dans le fichier *main.c*, pour créer une nouvelle pile de **doublon**, on fera

```
p=MYPILE_NEW(doublon);
```

Bon, ça n'a pour le moment pas un grand intérêt de faire ça, mais c'est notre première macro : voilà pourquoi elle est très simple.

On va vouloir rajouter à notre macro une vérification du retour de la fonction de création de pile. Dans ce cas, notre macro faisant plu d'une ligne de code, nous ne pourrons plus écrire

```
p=MYPILE_NEW(doublon);
```

En effet, ceci n'aura plus de sens étant donné que **MYPILE_NEW** représentera plusieurs lignes de code... A quelle ligne de code serait alors effectuée l'affectation à p ?

On va modifier notre macro afin de passer le pointeur de pile en paramètre, et on va allouer directement le pointeur dans la macro :

```
#define MYPILE_NEW(p, datatype) \
    p = Pile_new(sizeof(datatype)); \
    if (p==NULL) \
    { \
        fprintf(stderr, "Memory allocation error.\n"); \
        assert(0); \
    }
```

Dans le fichier *main.c*, pour créer une nouvelle pile de **doublon**, on fera

```
MYPILE_NEW(p, doublon);
```

On va faire de même pour la fonction de dépilement. On créé la macro suivante :

```
#define MYPILE_POP(p, data, datatype) \
    maillon *m = Pile_pop(p); \
    data = *((datatype *)(m+1)); \
    Maillon_free(m); \
```

Pour dépiler un élément de la pile, on fera

```
MYPILE_POP(p, r, doublon)
```

> **Questions**
>
> Que se passe-t-il si vous essayez de dépiler deux éléments de la pile ? Votre programme compile-t-il toujours ?

Une macro consistant simplement à associer des lignes de code à un mot clef, et à intégrer ces lignes de code dans le programme au moment de la compilation, nous aurons un problème si nous appelons deux fois la macro **MYPILE_POP**. En effet, dans ce cas, la variable m sera définie deux fois !

Pour éviter ce désagrément, on peut créer des variables "internes" à la macro. En vérité, ces variables seront internes à un bloc boucle de la macro :

```
#define MYPILE_POP(p, data, datatype) \
    do { \
        maillon *m = Pile_pop(p); \
        data = *((datatype *)(m+1)); \
        Maillon_free(m); \
    } while(0);
```

Ici, la boucle la plus simple est le **do...while(0)**, permettant au code de s'exécuter une seule fois et créant un bloc dans lequel les variables sont internes. On peut alors appeler plusieurs fois, dans la fonction principale, la macro **MYPILE_POP**.

Pour éviter des problèmes, on renommera les variables de façon à diminuer la chance que des variables dans la fonction appelante ne portent déjà le même nom...

```
#define MYPILE_POP(p, data, datatype) \
    do { \
        maillon *MYPILE_POP__m = Pile_pop(p); \
```

```
4         data = *((datatype *)(MYPILE_POP__m+1)); \
5         Maillon_free(MYPILE_POP__m); \
6     } while(0);
```

**Questions**

Faîtes de même pour la fonction d'empilement dans la pile : définissez une macro **MYPILE_PUSH**, et ré-écrivez votre fonction principale afin de n'utiliser que les macros.

Vous avez maintenant une bibliothèque de pile totalement générique. Vous devez simplement spécifier dans les trois macros le type des éléments présents dans la pile, et les fonctions appelées derrière s'adapteront.

# 5 Pile générique et Arraylist

Maintenant, ré-écrivez entièrement l'ArrayList réalisée précédemment afin qu'elle n'utilise que la nouvelle bibliothèque de pile et les macros.