TD n°2&3
Allocation mémoire & Piles « dynamiques » avec parcours d'un arbre de décision

1 Introduction

Dans les applications écrites en C jusqu'à présent, deux mécanismes d'allocation mémoire étaient mis en jeu :

- L'allocation mémoire ***statique*** pour les variables globales ainsi que pour les variables locales étiquetées **`static`**. Ces variables sont créées (allocation mémoire et initialisation) en début de programme dans un espace mémoire dédié (segment *data* ou *bss)* ; elles sont détruites en fin d'exécution. Cependant leur taille doit être connue dès l'écriture du programme, afin que le compilateur génère le code permettant de manipuler ces variables correctement.
- L'allocation mémoire ***automatique*** qui concerne les variables locales déclarées dans des fonctions ou plus généralement à l'intérieur de blocs encadrés d'accolades. De telles variables seront créées et détruites automatiquement à l'activation ou à la terminaison de la fonction ou du bloc dans lequel elles sont déclarées. Ici encore, le compilateur doit connaître la taille de celles-ci afin de gérer leur création et leur suppression. De telles variables sont généralement créées sur la pile d'exécution.

Dans ces deux cas, la taille des variables doit être connue dès l'écriture du programme et ne peut dépendre d'informations qui ne seront connues qu'au moment de l'exécution du programme. Ceci est souvent contraignant car cela oblige à sur-dimensionner des tableaux afin de traiter quelques cas particuliers alors que dans la majorité des cas des tailles inférieures conviennent.

2 Allocation mémoire dynamique

2.1 Définition

L'allocation dynamique de mémoire est effectuée en fonction de besoins qui se révèlent au cours de l'exécution du programme. Le développeur de l'application maitrise l'instant où l'allocation mémoire sera réalisée, l'instant où l'espace mémoire alloué sera libéré, ainsi que la taille de l'espace mémoire alloué. Il peut, en outre, ajuster en cours d'exécution la taille de l'espace mémoire alloué.

L'allocation dynamique de mémoire est réalisée dans un espace mémoire appelé le tas (heap en anglais)

2.2 Fonctions C d'allocation dynamique de mémoire

Préambule

Les prototypes des fonctions présentés ci-dessous sont présents dans le fichier *stdlib.h* (ce fichier se trouve dans les répertoires de librairies standard) qu'il faut donc inclure dans vos programmes pour pouvoir correctement utiliser ces fonctions :

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

Rappel sur le void * et le forçage de type

Les espaces mémoires alloués, ajustés en taille ou supprimés sont simplement désignés par leur adresse de début et, étant donné que ces espaces mémoires sont destinés à contenir toutes sortes d'informations, ces adresses de début seront caractérisées comme étant de type pointeur non typé (ou pointeur générique) : `void *`

Bien que l'on puisse affecter à une variable de type pointeur typé, la valeur d'un pointeur non typé, il est d'usage d'appliquer un forçage de type (cf. exemple du *malloc)*

2.2.1 malloc()

```
void *malloc (size_t size);
```

La fonction **`malloc()`** alloue *size* octets et renvoie un pointeur sur l'espace mémoire alloué. Le contenu de cette zone de mémoire n'est pas initialisé.

La valeur retournée est un pointeur sur la zone mémoire allouée ou **`NULL`** si la demande échoue.

Exemple :

```
char *Strdup (const char *s)
{
   char *p = (char *)malloc(strlen(s) + 1);
   if (p != NULL)
      return strcpy(p, s);
   return p;
}
```

2.2.2 calloc()

```
void *calloc (size_t nmemb, size_t size);
```

La fonction **`calloc()`** alloue l'espace mémoire nécessaire pour un tableau *nmemb* éléments, chacun d'eux occupant *size* octets. Cette zone est entièrement initialisée à zéro.

La valeur retournée est un pointeur sur la zone mémoire allouée ou **`NULL`** si la demande échoue.

2.2.3 free()

```
void free (void *ptr);
```

La fonction **free()** libère l'espace mémoire pointé par *ptr*, obtenu lors d'un appel antérieur à **malloc()**, **calloc()** ou **realloc()**. Si le pointeur *ptr* n'a pas été obtenu par l'un de ces appels ou s'il a déjà été libéré avec **free()**, le comportement est indéterminé. Si *ptr* est **NULL**, aucune tentative de libération n'a lieu.

free() ne renvoie pas de valeur.

2.2.4 realloc()

```
void *realloc (void *ptr, size_t size);
```

La fonction **realloc**() ajuste la taille du bloc de mémoire pointé par *ptr* pour l'amener à une taille de *size* octets. Lorsque la nouvelle taille est supérieure à l'ancienne, les données initiales sont préservées et le supplément d'espace mémoire n'est pas initialisé.

Si *ptr* est **NULL**, l'appel de **realloc()** est équivalent à **malloc(size)**, sinon *ptr* doit désigner une zone mémoire créée à l'aide de **malloc()**, **calloc()** ou **realloc()**.

Lorsque *size* vaut zéro, l'appel est équivalent à **free(ptr)**.

Il est possible que la taille du bloc mémoire pointé par *ptr* ne puisse pas être augmentée ; dans ce cas, un nouveau bloc mémoire d'une taille égale à *size* octets sera alloué puis, les données du bloc initial y seront recopiées et enfin le bloc initial sera libéré.

La fonction **realloc()** retourne un pointeur sur le nouvel espace mémoire ou **NULL** si la demande échoue ou si *size* valait zéro. En cas d'échec, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.

2.2.5 Qu'appelle-t'on « fuite mémoire » ?

On parle de fuite mémoire lorsque des espaces mémoires alloués par **malloc()**, **calloc()** ou **realloc()** deviennent inaccessibles et par conséquent ne peuvent plus être libérés ; l'espace mémoire disponible pour des allocations dynamiques ultérieures se réduisant d'autant. Un outil libre comme *Valgrind* (http://valgrind.org) permet de détecter ce genre de problème.

Il n'y a pas de « *garbage collector* » comme cela existe en Java ou en C# : le *garbage collector* encore appelé *ramasseur de miettes* ou *glaneur de cellules* a en charge de récupérer les espaces mémoires devenus inaccessibles par l'application qui les a alloués

Tous les espaces mémoires alloués lors de l'exécution d'un programme étant restitués lors de sa terminaison, le problème de fuite mémoire est donc particulièrement sensible pour des applications qui sont lancées pour s'exécuter sur de longues périodes (serveurs, ...)

2.2.6 Exercices :

Exercice n°1

Comment allouer un tableau dynamique *T* d'entiers dont la taille *n* est donnée par l'utilisateur lors de l'exécution du programme ?

Initialiser son contenu avec les valeurs de 1 à *n*.

Exercice n°2

Comment allouer et initialiser à 0 un tableau *Td* de flottants en double précision, dont la taille *n* est donnée par l'utilisateur lors de l'exécution du programme ?

Exercice n°3

Récupérer les espaces de mémoire occupés par *T* et *Td*.

3 Application : pile contigüe et allocation dynamique

Il est question ici de pile d'éléments contigus, c'est-à-dire rangés dans des cases mémoire successives : l'implémentation de ce type de piles utilise donc un tableau.

Il pourrait s'agir d'un tableau statique si l'on connait a priori la taille maximale comme cela avait été défini au TD de l'an dernier, mais dans cet exercice on ne souhaite pas se contraindre en fixant cette taille maximale.

La taille de cette pile évoluera donc en fonction des empilements et dépilements réalisés : elle s'appuiera donc sur un tableau **dynamique**.

Par conséquent la déclaration du type *pile_t* définie précédemment devient :

```
typedef struct
{
   int sommet ;        /* l'indice du sommet de pile          */
   elt_t * data ;      /* l'adresse de la zone de stockage    */
} pile_t ;
```

3.1 Implémentation de quelques fonctions de gestion de piles dynamiques

On adoptera les mêmes conventions que celles utilisées précédemment, de sorte que, seule l'implémentation des fonctions du module de gestion de pile devra être modifiée. Les applications déjà réalisées utilisant le module de gestion de pile n'auront pas à être modifiées ni recompilées. Une fois le fichier d'implémentation du nouveau module de gestion de pile, écrit, il suffit d'effectuer l'édition de liens avec les autres fichiers objet de l'application.

Dans nos piles, l'élément d'indice 0 existe et représente le fond de la pile (sorte de butée ou de sentinelle)

Exercice n°4 : la fonction init()

Réécrire la fonction d'initialisation d'une pile. Cette fonction crée le « fond de pile » et initialise le sommet de la pile créée.

En cas d'échec lors de l'allocation de mémoire, l'application est arrêtée et l'opération fautive est affichée sur l'écran. Ce dernier point peut être pris en charge par la macro-fonction *assert*, définie dans *assert.h* , qu'il faut donc inclure en tête de programme.

Son prototype est le suivant :

```
void init(pile_t *pPile) ;
```

Exercice n°5 : la fonction pop()

Réécrire la fonction `pop()` qui dépile l'élément en sommet de pile et retourne sa valeur. La taille de la pile sera alors ajustée.

En cas d'échec, l'application est arrêtée et l'opération fautive est affichée sur l'écran. Deux causes d'échec doivent être envisagées :

- La pile était vide.
- L'ajustement de sa taille échoue.

Son prototype est le suivant :

```
elt_t pop(pile_t *pPile) ;
```

Exercice n°6 : la function push()

Réécrire la fonction `push()` qui empile un élément sur une pile.

Son prototype est le suivant :

```
void push(pile_t *pPile, elt_t elt);
```

En cas d'échec lors de l'allocation mémoire, l'application s'arrête en désignant l'opération fautive.

3.2 Défaut et éléments de solution

Le défaut majeur de cette version est le coût important de l'opération *realloc*. En effet pour garantir la conservation des données (en cas d'augmentation de la taille, bien sûr), elle réalise souvent une allocation d'une nouvelle zone mémoire d'une taille suffisante, suivie de la copie des données initiales et terminée par la libération de la zone mémoire initiale.

Deux solutions :

1. Allouer et libérer par blocs de *n* éléments, pour minimiser les appels à *realloc* (voir paragraphe suivant),
2. Adopter une représentation chaînée des éléments de la pile (voir TD suivant).

3.2.1 Exemple d'implémentation de la première solution

Les constantes symboliques *SEUIL_ALLOC* et *SEUIL_FREE* définissent les tailles des blocs alloués et libérés. Si leurs valeurs sont différentes, on crée un effet d'hystérésis pouvant réduire les opérations `realloc`.

```
typedef struct
{
   int sommet;          /* l'indice du sommet de la pile       */
   Elt_t * data ;       /* l'adresse de la zone de stockage    */
   int taille ;         /* Taille maximale actuelle de la pile */
} pile_t ;
```

Les fonctions qui suivent peuvent être partiellement présentées ou proposées à titre d'exercice.

Exercice n°7

Réécrire les fonctions **`push()`** et **`pop()`** prenant en compte les considérations précédentes. Écrire une nouvelle fonction d'initialisation d'une pile ayant comme paramètre supplémentaire sa taille initiale.

```
void init(pile_t *pPile, unsigned int tailleIniale) ;
```

3.2.2 Représentation chaînée

Les structures de données dynamiques chaînées feront l'objet d'une prochaine séance de TD et du TP associé.

4 Piles et parcours de structures arborescentes

Une structure arborescente est constituée de sommets aussi appelés nœuds, reliés entre eux par des arcs. Elle permet de représenter efficacement des structures de données acycliques et hiérarchisées (organigramme d'une société, système de fichiers, arbre généalogique, arbre de décision, etc.).

Dans l'ensemble des sommets, on distingue le nœud *racine* sur lequel aucun arc incident n'aboutit, les nœuds *feuilles* d'où ne partent aucun arc ; le reste des nœuds étant des *nœuds internes*. Il est aussi fréquent d'utiliser un vocabulaire plus *familial* : on parle par exemple de nœud *père* pour désigner le nœud ascendant direct et de nœuds *fils*, pour désigner les nœuds descendants directs : on distingue d'ailleurs quelques fois le *premier fils* de tous ses *frères*.

Les algorithmes itératifs de parcours de structures arborescentes, utilisent de façon **explicite** une pile pour mémoriser les sommets traversés sur un chemin depuis la racine et être ainsi capables de remonter dans l'arbre et prendre d'autres directions. L'usage d'une pile est **implicite** dans le cas d'algorithmes récursifs.

Nous allons illustrer l'utilisation d'une pile dans le cas d'une exploration partielle d'un arbre de décision permettant dans un premier temps de traiter le problème des permutations des

éléments d'un ensemble et dans un deuxième temps, celui du placement des dames sur un échiquier.

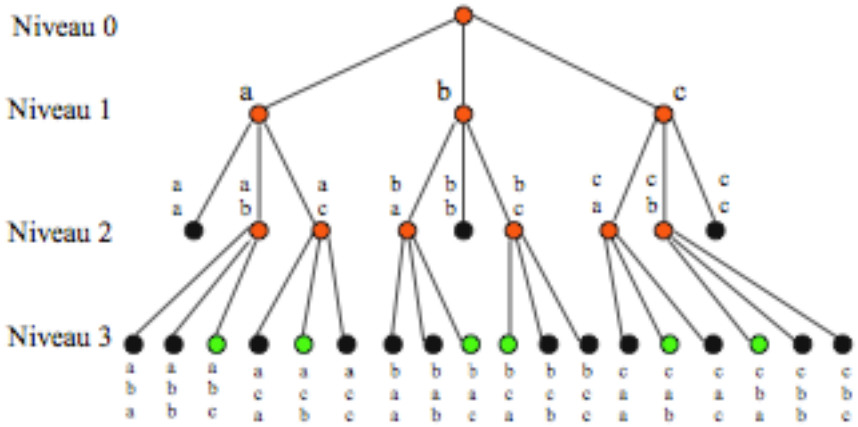
4.1 Le problème des permutations

Il s'agit de concevoir le programme permettant d'obtenir toutes les permutations des n éléments d'un ensemble. Ces solutions sont les feuilles d'un arbre de décision dont les nœuds internes portent des permutations partielles. On parle d'arbre de décision parce que, lors de sa construction ou de son exploration, la valeur du nœud courant permet de décider de la poursuite de la construction ou de la direction à prendre lors d'un parcours en profondeur d'abord (Depth *First Search)*.

Dans le problème qui nous intéresse, pour chaque nœud il y a n choix possibles, a priori.

Exemple :

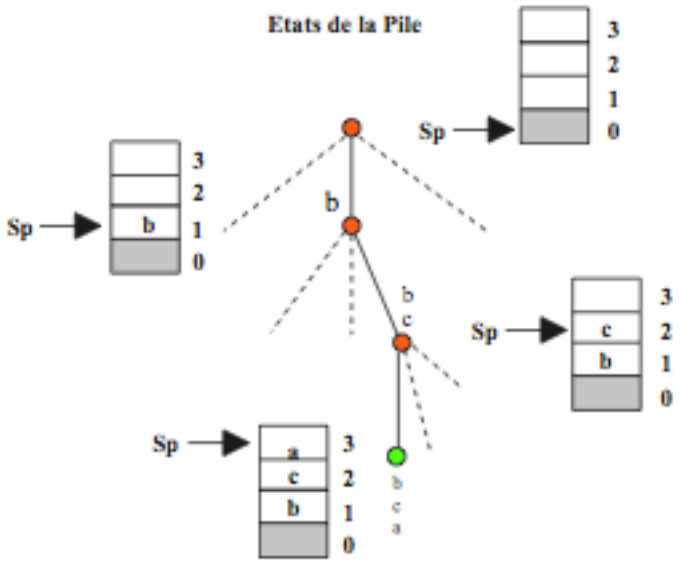
Soit l'ensemble $E = (a, b, c)$, l'arbre de décision est le suivant :



Lorsqu'on construit cet arbre, on peut directement éliminer les nœuds portant deux éléments identiques. Ainsi dans l'exemple ci-dessus, le nœud « *aa* » n'est pas valide : l'exploration de la branche qui y arrive peut être abandonnée.

On note qu'un nœud est déterminé par le niveau où il se situe et par la liste des choix successifs qui ont permis de l'atteindre, soit pour le dire autrement, par l'empilement de choix valides. Nous aurons donc besoin d'une pile pour réaliser le parcours de l'arbre de décision ; à tout instant, sa valeur détermine le nœud courant.

Exemple :



Pour le problème des permutations, un nœud est valide lorsque la liste des choix effectués ne comporte pas deux éléments identiques, ou encore lorsque le choix qui vient d'être fait, n'avait pas encore été fait.

Exercice n°8

Décrire en pseudo-code l'algorithme de parcours en profondeur d'abord d'un tel arbre de décision.

Cette description fera usage :

- de traitements appliqués au nœud courant et supposés déjà définis comme « *Passer au premier fils* », « *Remonter au père* », « *Passer au frère suivant* »,
- de prédicats s'appliquant au nœud courant, eux aussi supposés déjà définis, comme « *A-t-il un autre frère ?* », « *Est-il valide ?* », « *Est-il terminal ?* » (« est-ce une feuille ?),
- et du prédicat « *Le parcours est-il terminé ?* ».

Ces traitements et prédicats devront, bien entendu, être développés avant l'algorithme ; et ils feront directement appel au sommet de la pile qui caractérise le nœud courant.

Note :

Soit « *ab* » le nœud courant, alors :

- « *aba* » est son premier nœud fils
- « *a* » est son nœud père
- « *ac* » est son nœud frère suivant.