Allocation de mémoire

Jean Simard

Mardi 4 novembre 2008

1 Fonction `malloc`

La fonction `malloc` va nous permettre d'allouer de l'espace mémoire pour sauvegarder des variables. Cette fonction sera obtenue par l'appel de la bibliothèque `stdlib`.

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

Dans quels cas cette fonction peut-elle être utilisée ? En effet, on est en droit de se poser cette question car jusqu'à aujourd'hui, nous n'en avons jamais eu besoin. Pour répondre à cette question, prenons un exemple.

Imaginons un carnet d'adresses. L'utilisateur de ce programme peut au choix ajouter, modifier ou supprimer un contact dans ce carnet d'adresses. Un contact est constitué par une structure contenant par exemple, le nom, le prénom, l'adresse, le numéro de téléphone et l'adresse de courriel. Avec nos connaissances actuelles, nous devrions déclarer un tableau de grande taille N en espérant que l'utilisateur n'enregistre pas plus de N contacts. Mais comment faire lorsque $N+1$ contacts sont enregistrés ? Autre problème, pourquoi déclarer un tableau de taille $N = 10000$ lorsque l'utilisateur n'enregistre que 20 contacts ?

Nous allons donc choisir une autre stratégie et allouer ou libérer la mémoire au-fur-et-à-mesure que l'utilisateur créé ou supprime des contacts. Pour allouer la mémoire, nous allons utiliser la fonction `malloc`. Il faut également maîtriser la notion de pointeurs. Reprenons l'exemple des structures déclarées dans les TP précédents pour les nombres complexes. La fonction `malloc` prend en argument la taille de l'espace mémoire à allouer en octets. Pour plus de simplicité, il faut utiliser la fonction `sizeof`. Et la fonction `malloc` renvoie l'adresse de l'espace mémoire nouvellement alloué. On récupérera donc la valeur de cette fonction dans un pointeur.

Code 1: AllocationComplexe.c

```
typedef struct sComplexe
{
  double dReelle;
  double dImaginaire;
} Complexe;

Complexe * pComplexe = NULL;
pComplexe = malloc( sizeof( Complexe ) );
```

Cependant, cet exemple comporte un problème de conversion. En effet, la fonction `malloc` pouvant être utilisée pour allouer de l'espace mémoire pour n'importe quel type (structure, `int`, `char`, `float *`...) elle renvoie un pointeur de type neutre `void *`. Il va donc falloir convertir ce pointeur dans le type que l'on désire.

CODE 2: AllocationComplexeConverti.c

```
typedef struct sComplexe
{
  double dReelle;
  double dImaginaire;
} Complexe;

Complexe * pComplexe = NULL;
pComplexe = (Complexe *) malloc( sizeof( Complexe ) );
```

Avant la fin du programme, il va falloir libérer cette mémoire. En effet, pour les variables déclarées dynamiquement, la mémoire ne se libère pas automatiquement à la fin d'un bloc. L'espace est alloué pour toute la durée du programme. Pour cela, on utilise la fonction `free`. On donne en paramètre un pointeur sur l'espace mémoire alloué. Dans le cas précédent, on écrira

```
free( pComplexe );
```

Exercice 1 *Le programme devra demander à l'utilisateur son nom, son prénom et son adresse de courriel. L'objectif sera de limiter l'espace mémoire utilisé pour sauvegarder ces informations en allouant dynamiquement l'espace mémoire pour l'ajuster à la longueur des chaînes de caractères. Nous utiliserons la structure* `Contact` *(voir* CODE *3).*

CODE 3: Structure d'un contact

```
typedef struct sContact
{
  char * cNom;
  char * cPrenom;
  char * cCourriel;
} Contact;
```

Vous remarquerez dans cette structure que les chaînes de caractères sont représentés par de simples pointeurs. On peut en déduire qu'aucun espace mémoire n'est disponible pour sauvegarder des caractères. Il va donc falloir effectuer les allocations mémoires.

On utilisera la fonction `scanf` *pour lire une chaîne de caractères.* ***Attention*** *– Le caractère de contrôle* `%s` *arrête la lecture de caractères dès qu'elle rencontre un blanc (espace, tablature, passage à la ligne). Un nom ou un prénom en deux parties ne fonctionnera donc pas. Vous pouvez par exemple utiliser le caractère* souligné `'_'` *au moment où vous donner votre nom (ou votre prénom) pour contrer ce problème.*

Afin d'utiliser la fonction `scanf` *avec le caractère de contrôle* `%s`*, il va nous falloir une variable pouvant contenir une chaîne de caractères. Ne connaissant pas*

à l'avance la taille du prénom, du nom ou de l'adresse de courriel, on utilisera un tableau de caractères temporaire de taille suffisamment grande pour contenir tous les caractères (signalons toutefois que c'est une manière non sûre de procéder !). Ensuite, on pourra allouer l'espace mémoire grâce à la fonction `strlen`*. Puis nous pourrons recopier la chaîne dans la structure à l'aide de la fonction* `strcpy`*. On écrira également une fonction qui permettra d'afficher les chaînes de caractères de la structure. Ce sera une fonction qui prendra une structure* `Contact` *avec un passage par référence (voir* Code *4).*

Code 4: Exemple d'exécution

```
> ./contacts
Entrer votre nom : SIMARD
Entrer votre prenom : Jean
Entrer votre courriel : jean.simard@limsi.fr
L'adresse de courriel de SIMARD Jean est jean.simard@limsi.fr
>
```

2 Liste chaînée

Dans le langage C, les listes chaînées sont obtenues à l'aide de structures. Chaque instance de la structure est un des maillons de la chaîne. Afin de connaître le maillon suivant, la structure contient un pointeur qui contient l'adresse de la structure suivante dans la chaîne.

```
typedef struct sMot
{
  char * cMot;
  Mot * pMotSuivant;
} Mot;
```

Cependant, il ne faut pas oublier qu'au moment de déclarer ce pointeur sur la structure, nous sommes en train de déclarer cette même structure. Nous sommes donc en train d'essayer de déclarer un pointeur sur un type qui n'existe pas encore. L'écriture ci-dessus n'est donc pas autorisée car on utilise le type `Mot` alors qu'il n'existe pas encore. Heureusement, le langage C va nous permettre de contourner ce problème car même si le type `Mot` n'existe pas encore, le type `struct sMot` existe déjà. Nous allons donc écrire

```
typedef struct sMot
{
  char * cMot;
  struct sMot * pMotSuivant;
} Mot;
```

Ensuite, il suffit de bien initialiser la valeur du champ `pMotSuivant` avec le pointeur correspondant.

Dans le programme, une variable sera déclarée pour toujours contenir l'adresse du premier maillon de la liste chaînée (sinon, on perd l'endroit où se trouve la liste chaînée dans la mémoire).

On notera que le pointeur du dernier maillon de la chaîne ne doit pointer sur rien. Dans le langage C, un pointeur qui ne pointe sur rien est initialisé avec la valeur `NULL`. De la même façon que la caractère `'\0'` dans les chaînes de caractères, la valeur `NULL` sera un indicateur pour indiquer la fin de la liste chaînée (voir FIG. 1).

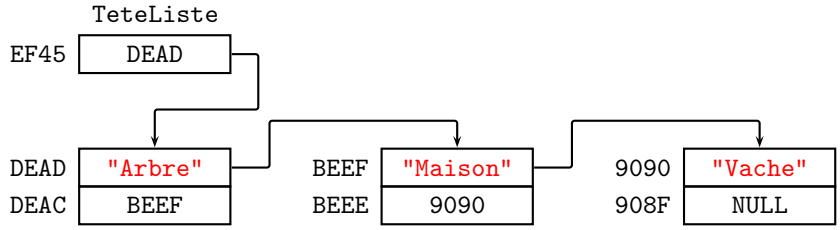

FIG. 1: Liste chaînée.

Exercice 2 *Nous allons créer un programme qui sauvegarde tous les mots donnés en arguments dans des maillons de listes chaînées. Par exemple, on exécute le programme avec la ligne de commande suivante*

```
> ./dictionnaire Bonjour tout le monde
```

Les mots devront être sauvegardés dans une liste chaînée dont les maillons seront de type `Mot` *déclarée ci-dessus. Le résultat sera une liste chaînée contenant quatre maillons contenant* `"Bonjour"`, `"tout"`, `"le"` *et* `"monde"` *(le nom du programme sera ignoré). On créera une fonction qui permettra d'afficher chaque mot de cette liste chaînée (voir* CODE *5).*

CODE 5: Exemple d'exécution

```
> ./dictionnaire Bonjour tout le monde
monde
le
tout
Bonjour
>
```

On remarque que l'ordre des mots n'est pas respecté. Cela dépendra de la façon dont vous ajouterez un maillon dans votre liste chaînée et de la manière dont vous procédez à l'affichage de la liste chaînée.

3 Les pointeurs de pointeurs

Nous avons déjà vu comment fonctionnait les pointeurs. On déclare une variable qui contient l'adresse d'une autre variable. Que se passe-t-il si cette autre variable est déjà un pointeur ? Rien de spécial, cette opération est totalement autorisée. Mais alors quelle peut en être l'utilité ? Nous allons le voir sur l'exemple de la matrice. En effet, les allocations mémoires nous on fait découvrir comment

déclarer des tableaux à une dimension de façon dynamique. Dans le cas de tableaux à deux dimensions, nous allons avoir des pointeurs de pointeurs. Et dans le cas de tableaux à trois dimensions, des pointeurs de pointeurs de pointeurs. Par exemple, la matrice suivante

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{1}$$

sera sauvegardée en mémoire selon la graphique présent sur Fig. 2.

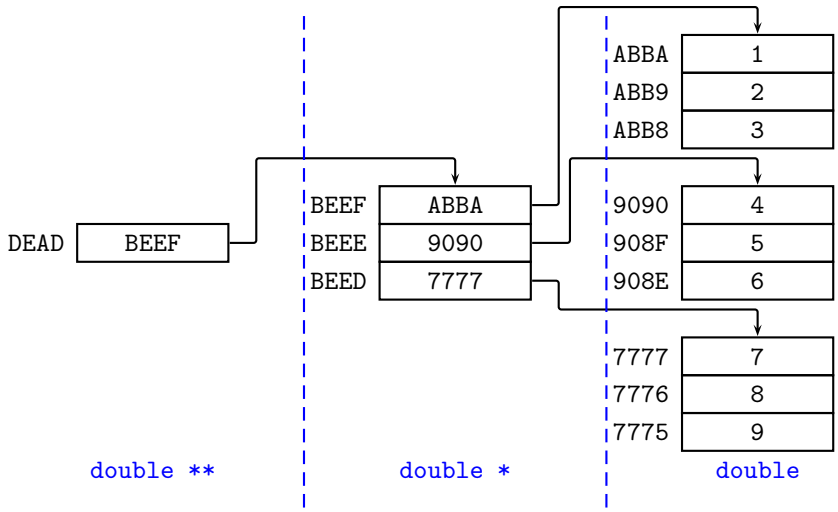

Fig. 2: Pointeurs de pointeurs.

Exercice 3 *Nous prendrons la structure de matrice définie par le* Code *6.*

Code 6: Structure de matrice

```
typedef struct sMatrice
{
  double ** dMat;
  unsigned int uiNbLignes;
  unsigned int uiNbColonnes;
} Matrice;
```

L'objectif est d'écrire un programme qui demandera à l'utilisateur de donner le nombre de lignes et le nombre de colonnes de la matrice puis d'allouer l'espace de mémoire nécessaire en fonction des valeurs lues.