Langages et Programmation

Allocation Dynamique - Chaînage

Enregistrement / Structure

Un enregistrement (ou *structure* dans la terminologie C) est un type de données qui permet de représenter un n-uplet de valeurs de différents types. Formellement il s'agit donc d'un *produit de type*.

L'exemple suivant déclare une structure de nom `personne` qui contient trois *champs* : un `nom` et un `prenom`, qui sont des chaînes de caractères, et un `age`, qui est un entier.

```
struct personne
   {
   char nom[20];
   char prenom[20];
   int age;
   } ;
```

On peut ensuite déclarer des variables de type `struct personne` : `struct personne p1, p2 ;`

L'accès aux différents champs d'une variable de type structure se fait alors de la manière suivante :

```
p1.age=28 ; p1.nom[0]='M' ; strcpy (p2.nom, p1.nom) ; ...
```

On peut également directement initialiser une structure lors de sa déclaration :

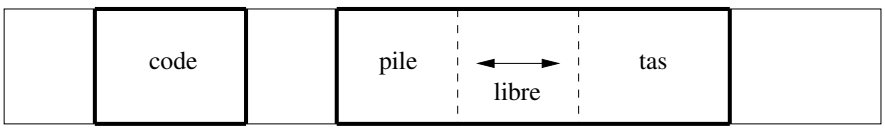
```
struct personne p1 = {"TALON", "Achille", 63} ;
```

Notons enfin que l'affectation entre structure est permise (`p2=p1`), mais pas leur comparaison (`p1 == p2`) ! On peut bien sur déclarer des tableaux de structures : `struct personne Repertoire[50]`.

Allocation Dynamique

Les déclarations de variables rencontrées jusqu'à présent (qu'elles soient de type entiers, tableaux, structures, etc.) se traduise par une allocation statique de mémoire : l'espace mémoire nécessaire est calculée à l'exécution, cette mémoire est allouée dans la *pile d'exécution*, leur *durée de vie* est celle de la fonction dans laquelle elles sont déclarées (ou celle du programme lui-même pour les variables *globales*).

Les langages de programmation offrent également un mécanisme permettant de gérer (i.e., allouer/libérer) un espace mémoire arbitraire, *au cours de l'exécution d'un programme* : on parle d'*allocation dynamique*. Techniquement, cet espace mémoire est alloué dans un zone de la mémoire centrale distincte de la pile, appelée le *tas* (ou *heap*) :



En C, la primitive d'allocation usuelle est `malloc()`, qui renvoie l'adresse (donc un *pointeur* de type `void`) d'un espace mémoire dont la taille (en octets) est spécifiée en paramètre. Une conversion de type permet alors de transformer ce pointeur "non typé" en un pointeur "typé", ce qui en facilitera l'utilisation :

```
int *p1 ;
p1 = (int *) malloc (sizeof(int) * 10) ;
```

La déclaration précédente est donc équivalente à celle d'un tableau de 10 entiers, mais alloué dynamiquement. On peut donc écrire `*p1=5`, `p1[2]=6`, `*(p1+2)=6`, etc. Notons que l'instruction `int *p2 ; p2 = p1` aura pour effet de mémoriser également dans `p2` l'adresse du même espace mémoire. Une telle affectation n'est pas possible dans le cas de tableaux alloués statiquement :

```
int T1[10] ; int T2[10] ; T2 = T1 ; /* affectation interdite ! */
```

En principe, l'espace mémoire ainsi alloué à une durée de vie égale à celle du programme. Il est toutefois possible (et même conseillé) de *libérer* (entiérement) cet espace mémoire avec la fonction `free` : `free(p1) ;`.

Dans certains langages (Ada, Java, etc.) il existe un mécanisme supplémentaire de gestion de l'espace mémoire appelé le *ramasse-miettes* (ou *garbage collector*, ou *glaneur de cellules*). Son rôle est de libérer automatiquement les espaces mémoires allouées dans un programme qui ne sont plus référencés par aucune variable "active" (par exemple si la fonction dans laquelle `p1` et `p2` ont été déclarées est terminée). Ce mécanisme délivre ainsi le programmeur du problème de la libération de la mémoire.

Remarques :
- Un appel à la fonction `malloc()` peut ne pas s'exécuter correctement (par exemple s'il n'y a pas assez de mémoire disponible) ; dans ce cas la valeur renvoyée est la valeur `NULL`.
- Libérer (avec `free()`) un espace mémoire qui n'a pas été alloué auparavant dans le programme (ou qui a déjà été libéré) provoque une erreur à l'exécution.

Séquences Chaînées

L'allocation dynamique offre un moyen supplémentaire de représenter une séquence d'éléments de même type, la représentation par chaînage à l'aide de pointeurs. Cela consiste à allouer (à la demande) une "cellule" par élément de la séquence, chaque cellule étant une structure comprenant :
- la valeur d'un élément de la séquence ;
- un pointeur vers la cellule contenant l'élément suivant (ou la valeur `NULL` s'il n'y a pas de suivant).

La séquence elle-même est alors représentée par un pointeur sur son élément de tête.

L'intérêt de cette représentation (par rapport à une représentation de type "tableau") est alors :
- de faire varier dynamiquement la taille de la séquence (sans hypothèse sur sa taille maximum au moment de l'écriture du programme) ;
- de "déplacer" des éléments à l'intérieur d'une séquence, ou entre séquences, uniquement par affectation de pointeurs, sans recopies d'éléments (c.f. exos ci-après).

L'inconvénient est bien sur l'absence d'"accès direct" au ième élément (il faut au préalable parcourir les (i-1) premiers éléments).

Exemple de déclaration en C :

```
struct cellule {    /* definition d'un type cellule */
       int elem ;
       struct cellule *suc ;
} ;

struct cellule *S ;       /* adresse de l'element de tete */
```

Quelques exercices

On considère dans toute la suite des séquences d'entiers représentées par chaînage de cellules. On accède donc à une séquence par l'adresse de son élément de tête.

Parcours.
Ecrire un programme qui compte le nombre d'éléments de S strictement supérieurs à une constante `K`

Partition.
Ecrire un programme qui partitionne une séquence S en deux sous-séquences S_1 et S_2 telles que S_1 contiennent les éléments de S strictement supérieurs à une constante `K`, S_2 les éléments de S inférieurs ou égaux à `K`. Seuls les chaînages entre les éléments de S doivent être modifiés.

Insertion.
Ecrire un programme qui ajoute "à sa place" un élément de valeur x dans une séquence S ordonnée croissante.

Suppressions.
On suppose que S est une séquence ordonnée croissante. Ecrire un programme qui supprime de S les éléments de valeur impaire. La nouvelle séquence obtenue doit rester ordonnée croissante. Les cellules supprimées seront libérées.

Interclassement.
On suppose que S_1 et S_2 sont deux séquences ordonnées croissantes. Ecrire un programme qui produit séquence S_3 correspondant à l'interclassement de S_1 et S_2. On considèrera deux cas :

1. S_3 est construite avec les cellules de S_1 et S_2 (ces deux séquences sont donc "vides" à la fin du programme).
2. S_3 est une nouvelle séquence (S_1 et S_2 restent inchangées).