Travaux Dirigés d'informatique

Notes sur le langage *C*

—Deug Sciences—

Tableaux, Pointeurs, Allocation dynamique

Nous présentons de la manière la plus simple possible un petit résumé sur les différences entre tableaux et pointeurs, tant au niveau de la syntaxe, des opérations réalisables, que de la représentation en mémoire. Nous expliquons ensuite l'intérêt et la nécessité de l'allocation dynamique. Cette feuille de rappels est finalement ponctuée par des petits exercices élémentaires.

Introduction

La mémoire réservée par l'ordinateur à tout processus peut se visualiser de manière linéaire comme une succession de cases mémoire que l'on numérote arbitrairement (1 numéro=1 adresse). Lorsque l'on déclare une variable dans un programme *C*, le compilateur prévoit de réserver une succession de cases unitaires (1 case unitaire=1 octet), appelée *emplacement mémoire* pour cette variable.

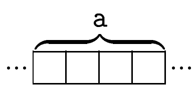
Le programme n'est autorisé *a priori* à travailler que sur des emplacements qui sont réservés de manière statique (au lancement du programme) et les accès à la mémoire sur des endroits non-réservés sont souvent la cause de l'interruption du programme en cours d'exécution avec génération d'un fichier `core` (qui contient des informations relatives à l'exécution du programme jusquà l'interruption).

De manière générale, il est important de comprendre quelles sont les opérations valides sur la mémoire réservée à un programme.

Effet d'une déclaration de variable au niveau mémoire et opérations valides sur cette variable

Variable simple

Supposons que les objets de type `int` occupent 4 octets mémoire. La déclaration `int a;` dans un programme *C* permet au compilateur de prévoir un emplacement mémoire de taille 4 (`sizeof(int)`=4) pour `a` :



Lorsque la variable `a` sera rencontrée plus bas dans le programme *C*, deux informations suffiront pour consulter ou modifier le contenu de l'emplacement mémoire de `a` : l'adresse de `a` et l'étendue de la place qui lui est réservée (`sizeof(int)`). Autrement dit, l'emplacement mémoire est totalement défini par l'adresse du premier octet de l'emplacement et par l'étendue de l'emplacement. Si l'une de ces données est erronée, il y a un *grand risque d'interruption* du programme : il faut à tout moment maîtriser les deux paramètres lorsque l'on consulte la mémoire.

Les modifications d'une variable se font par l'affectation (`=`), ou par le passage par adresse dans une fonction (rappelez-vous qu'en C, tous les objets passés en arguments sont des recopies : on ne peut modifier un objet que si on dispose de la donnée de son emplacement mémoire, à savoir son adresse et sa taille—cette dernière est implicite grâce au type de l'objet).

⇒ Retenez que :

La mémoire réservée pour un programme doit se visualiser comme une succession de cases contiguës

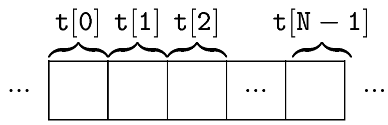
Un regroupement de cases consécutives est appelé un *emplacement mémoire*

Un *emplacement mémoire* est référencé de manière unique par la donnée d'une adresse mémoire et celle de la taille de l'objet codé en mémoire

Une déclaration de variable entraîne toujours la réservation d'un emplacement mémoire

Tableau

De même qu'un entier est codé sur 4 cases mémoire (en général) consécutives, un tableau de `N` entiers est codé sur `N` emplacements d'entiers consécutifs. Soit donc la déclaration `int t[N];`



Chaque `t[i]`, `i` $\in \{0, .., \mathtt{N}-1\}$ possède exactement le même statut que la variable `a` vue plus haut.

La question importante est : *peut-on parler d'emplacement mémoire pour un tableau ?* En C, la déclaration `int t[N]` définit effectivement un emplacement mémoire puisqu'elle demande au compilateur de prévoir de réserver `N` emplacements consécutifs de taille `sizeof(int)` à partir de l'adresse `t`. Il ne faut donc pas confondre la notion usuelle de tableaux (ensemble de cases) et celle donnée en C (adresse de la première case+taille du tableau). Pour simplifier les choses, si le compilateur prévoit de réserver l'emplacement mémoire pour `t` à l'adresse `10000`, alors il est commode de remplacer virtuellement `t` partout dans le programme C par la valeur `10000`. On évite ainsi d'écrire des instructions non valides :

Objet/Instruction	valide	dangereux	signification
`t`	$\surd$		l'adresse de la première case de `t`
`t[0]`	$\surd$		le contenu de la première case de `t`
`*t`	$\surd$		le contenu de la première case de `t` (préférer malgré tout la notation `t[0]`)
`&t`	$\surd$	$\surd$	la même chose que `t`. A proscrire !
`&(t[1])`	$\surd$		l'adresse de la deuxième case de `t`
`t+1`	$\surd$	$\sim$	l'adresse de la deuxième case de `t` (préférer la notation `&(t[1])`)
`t++`		$\surd$	INTERDIT!!!
`t=...`		$\surd$	INTERDIT!!!

$\Rightarrow$ De manière générale, toute opération tendant à modifier `t` est strictement interdite (ne sont autorisés que les accès aux cases du tableau) : que diriez-vous si vous voyiez l'instruction `10000=11000` (quand on substitue `t` par `10000`), ou encore `10000++` ?

▸ **Exercice 1.** *Pourquoi après la déclaration* `int t[N]`, *les deux notations* `t` *et* `&t` *sont-elles équivalentes ?*

Un tableau peut être passé en argument d'une fonction sans aucun problème (sauf qu'il faut connaître la taille des cases du tableau pour que le compilateur comprenne sur quel type d'objet il travaille), puisqu'il référence des emplacements mémoire. La modification des cases d'un tableau ne nécessite donc pas un passage par adresse du tableau (d'ailleurs, voyez l'exercice précédent pour vous en convaincre).

$\Rightarrow$ Retenez que :

Un *tableau* se déclare comme une variable, mais n'en a pas le statut, puisqu'il n'est pas modifiable

Dans un tableau, seules les cases sont modifiables

Pointeurs

Bien que présentant quelques points communs avec les tableaux, les pointeurs sont des objets desservant des objectifs totalement différents. Les pointeurs ont été introduits pour *se promener* sur les différents emplacements de la mémoire. Pourquoi les emplacements ? Simplement parce qu'il est plus facile de sauter d'un objet à un autre que de parcourir chacun de ses octets.

Ainsi, lorsqu'un pointeur `p` sur des objets de type `T` est déclaré, l'instruction `p++` fait passer directement à l'emplacement de l'objet suivant (qui est censé être aussi de type `T`).

La définition exacte d'un pointeur est : *une variable qui contient une adresse vers un objet de type donné*.

De même, lorsqu'un pointeur se promène sur la mémoire, il se peut qu'il aille bien au-delà de ce qu'il est autorisé à parcourir : une erreur d'exécution se produit (tel un `bus error`) et un fichier `core` est généré avant l'interruption définitive du programme.

Voici la déclaration d'un pointeur de caractères :

```
char * s;
```

que l'on lit de la droite vers la gauche en disant `s` *est un pointeur (*) de caractères.* Par ailleurs, si `s` est effectivement un pointeur de caractères (c'est-à-dire `s` contient l'adresse d'un emplacement mémoire qui contient un caractère), alors `*s` est la valeur de ce caractère, à savoir la valeur à l'adresse que contient `s`.

La grande différence avec un tableau, est que `s` est vraiment une variable, au sens où on peut lui affecter une valeur : l'opération `s=...` est donc tout à fait valide. Plus généralement, toutes les opérations sur les tableaux sont valides pour les pointeurs (un pointeur peut être positionné sur un tableau), et en plus, toutes les opérations tendant à modifier la valeur du pointeur sont valides (`s++`, `s=t`, etc.)

L'avantage du pointeur, nous venons de le dire, est la possibilité de se promener sur la mémoire. C'est par ce biais que nous sommes capables d'aller modifier des objets : il en va ainsi pour les passages d'arguments par adresse (par pointeur), où le pointeur passé en paramètre nous autorise à modifier l'objet pointé.

▸ **Exercice 2.** *Soit les déclarations suivantes :*

```
int t[10];
int *p=t;
```

Que signifie la deuxième déclaration ? Que pensez-vous du résultat des instructions :

```
printf("%p, %p\n",t,&t);
printf("%p, %p\n",p,&p);
```

⇒ Retenez que :

Un *pointeur* est une variable qui contient une adresse mémoire
⇒ Un pointeur référence un emplacement mémoire
Il ne faut pas confondre pointeur et tableau !...
Un pointeur est indispensable pour modifier la mémoire

Allocation dynamique—besoins en mémoire supplémentaire

L'expression *allocation dynamique* est le reflet d'un phénomène naturel en programmation : la nécessité pour certains programmes "gourmands" d'utiliser de plus en plus de mémoire.

Les deux instants où le programmeur a besoin de l'allocation dynamique sont les suivants :

1. lorsqu'il faut augmenter ou diminuer la taille en mémoire d'un objet ;

2. lorsqu'on a besoin d'un nouvel emplacement mémoire.

Une fonction *C* réalise la première tâche : c'est la fonction `realloc()`, qui prend en premier argument l'objet à redimensionner et sa nouvelle taille en deuxième argument. Deux fonctions *C* très similaires s'occupent de la deuxième tâche : `malloc()` et `calloc()`. Par ailleurs, la fonction `free()` libère l'espace alloué pour son argument, ce qu'il ne faut surtout pas oublier de faire lorsqu'un objet devient inutile : cela permet de redonner de la mémoire allouable pour les prochains besoins en mémoire.

Lorsque nous déclarons un pointeur `p` sur un objet de type `T`, deux cas se présentent fréquemment : soit le type de l'objet correspond à une cellule (disons un simple enregistrement —`struct`), soit il s'agit de simuler un tableau de taille variable —appelé parfois *vecteur*. Ces deux types d'objets correspondent à des besoins relativement différents, mais la philosophie d'allocation mémoire reste la même.

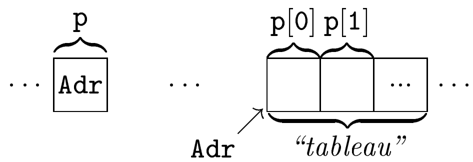
Pointeur vers une structure

Lorsqu'une fonction renvoie un résultat d'un certain type `T`, ce type peut bien évidemment être autre chose qu'un type de base (`int`, `char`, `float`, etc.).

Dans le cas où l'objet renvoyé est une structure, il peut arriver que l'on soit obligé de renvoyer un pointeur vers une structure nouvellement allouée. Cette remarque prend toute son importance lors de l'introduction des structures récursives (M3MI). Les détails sont fournis en travaux dirigés.

Pointeur en tant que tableau redimensionnable

Lorsque l'on ne connaît pas à l'avance la taille d'un *tableau* d'objets de type `T`, on a tout intérêt à déclarer un pointeur sur le type `T`. En effet, soit la déclaration `T * p;`, où l'on voit que `p` est censé contenir l'adresse d'un objet de type `T`. Comme on sait qu'un tableau est une succession d'emplacements mémoire contigus, il suffit maintenant de dire que `p` va référencer le premier emplacement mémoire `Adr` d'une telle suite d'emplacements, ce qui permet de simuler un tableau (au sens *C*) :



Une telle simulation n'est réalisable que si l'on sait comment créer le *"tableau"*. La fonction `malloc()` (ou `calloc()`) se charge de renvoyer l'adresse d'une succession d'emplacements nouvellement réservés en mémoire : il n'y a donc plus qu'à mettre le résultat de l'appel à `malloc` dans notre variable `p` : `p = (T *)malloc(N*sizeof(T));`, où `N` est le nombre d'emplacements (la taille du *tableau*) désirés.

La question que vous vous posez peut-être est : *"quel est l'intérêt du* `malloc`*, alors qu'on aurait pu déclarer* `T p[N]` *?"*. En fait, il faut cerner vos besoins : Si le tableau codé par `p` doit s'étendre (et/ou se réduire) au fur et à mesure de l'exécution du programme, alors il faut se poser la question de savoir s'il est avantageux d'utiliser une structure dynamique (un pointeur vers un tableau de taille variable), car il se peut qu'un tableau statique de taille raisonnable suffise. De plus, si vous utilisez un pointeur, alors il vous faudra trouver un moyen pour mémoriser la taille de votre *tableau*, afin de ne pas déborder sur les endroits de la mémoire où vous n'avez pas le droit de vous promener.

▸ **Exercice 3.** *Supposons que vous écriviez un programme qui ouvre un fichier et qui stocke les lignes dans un "tableau" de lignes. La taille de chacune des lignes n'est pas connue à l'avance, pas plus que le nombre de lignes du fichier. Comment vous y prenez-vous ?*

⇒ Retenez que :

L'*allocation dynamique* concerne essentiellement la simulation de tableaux de taille variable et les structures récursives ;

Il peut être judicieux d'utiliser des grands tableaux, plutôt que de faire des `malloc` et `realloc`