INFO2009-Introduction à l'informatique

Aide à la réussite - Les tableaux, les boucles, la complexité et les invariants

Bertrand Alexis

Université de Liège

2024

Tableau

Un tableau est une collection de variables du même type. Ces éléments sont ordonnés de sorte à ce que chacun est caractérisé par un indice unique au sein du tableau. En C (et dans la majorité des programmes informatiques), le premier indice du tableau de taille ℓ est 0 et le dernier est $\ell - 1$, en sachant que tous les indices sont consécutifs. Un tableau de taille ℓ aura donc les indices $0, 1, 2, \ldots, \ell - 2, \ell - 1$.

Définir un tableau

Afin de définir un tableau unidimensionnel (un vecteur) de taille fixe, nous effectuons l'instruction : `type var[nb_elements];`
Il est également possible d'initialiser un tableau lors de sa définition.

Exemple

```
int var[2] = {1,2};
```

Accéder à un élément du tableau

Afin d'accéder à un élément du tableau, nous devons donner l'indice de ce dernier dans le tableau. On peut faire ceci de cette manière : `var[i]`, où i correspond à l'indice de l'élément qu'on tente d'accéder. Attention que si on tente d'accéder à un indice qui n'est pas dans le tableau, nous obtiendrons une erreur d'accès invalide à la mémoire, aussi appelé une erreur de segmentation.

Accéder à un élément du tableau par un pointeur

Quand on initialise une variable de type tableau, la variable correspond en fait à l'identificateur du tableau et correspond à l'endroit dans la mémoire où se situe le tableau. Grâce à cette représentation, faire l'instruction `var + i` correspond à récupérer l'endroit dans la mémoire où se situe le ième élément du tableau. Dans ce cas, les deux instructions suivantes sont pareilles et renvoient le même résultat, à savoir la valeur du ième élément du tableau : `var[i] == *(var + i)`.

Fonction avec un tableau en argument (1/2)

Afin de passer un tableau en argument, il suffit de mettre `type var[]` comme argument dans la définition de la fonction. Il est alors possible de donner le tableau en argument en appel à la fonction en mettant l'identificateur du tableau. Comme on a vu que `var[i] == *(var + i)`, on peut facilement comprendre que on peut également mettre `type* var` comme argument de la fonction à la place.

Fonction avec un tableau en argument (2/2)

Exemple

```
void f(type *var){
    ...
}

int main(){
    type var[i];
    f(var);
    return EXIT_SUCCESS;
}
```

Types de passage de paramètre

Rappel

Comme vous avez dû le voir dans le cours théorique, les arguments peuvent être passés par valeur ou par adresse.

Étant donné que `var[i] == *(var + i)`, on peut facilement comprendre que donner un tableau en argument correspond à donner son adresse mémoire et que du coup, on effectue toujours un passage par adresse dans ce cas.

Tableau multidimensionnel

Il n'est pas que possible de créer un vecteur. Nous pouvons créer un tableau avec autant de dimensions qu'on veut. Pour chaque dimension, il suffit de préciser sa taille lors de la définition du tableau : `type var[`ℓ_1`][`ℓ_2`]...`. Accéder à un élément du tableau fonctionne de la même façon : `var[i][j][...]`.

Créer un tableau à taille dynamique

Parfois, nous ne connaissons pas la taille d'un tableau au préalable. Il est donc intéressant d'allouer un tableau dynamiquement. Pour ce faire, il va falloir utiliser une des fonctions `malloc`, `calloc`, ou `realloc`.

Exemple

```
type* var = malloc(nb_element * sizeof(type));
```

Désallouer un tableau à taille dynamique

Attention !

Lorsqu'on alloue dynamiquement une variable en C, il faut obligatoirement libérer l'espace utilisé grâce à la fonction `free(var)` avant la fin du programme, sinon nous risquons des pertes de mémoires. Les pertes de mémoires sont des endroits de la mémoire qui ne seront plus accessibles même après la fin de l'exécution du programme car réservés à des variables du programme.

Note

Certains langages de programmation ont ce qu'on appelle un "garbage collector", il s'agit d'un moyen de récupérer toute la mémoire qui a été allouée au programme et qui n'est pas utilisé afin de libérer l'espace si nécessaire. Ce n'est cependant pas le cas en C. La règle un malloc = un free est alors d'application !

Les différents types de boucle

Il y a 3 types de boucles en C : les `while`, les `do while`, et les `for`.

while

```
while(condition){...}
```

do while

```
do{...}
while(condition);
```

for

```
for(init var; condition; opération fin de boucle){...}
```

Complexité

Nous utilisons la notation $\mathcal{O}$ pour désigner la complexité en temps d'une fonction. Cette complexité correspond au nombre d'itération que devra faire la fonction dans le pire des cas, donc le nombre d'itération maximum.

Exemple

La complexité en temps d'une fonction qui doit faire n itération est donc $\mathcal{O}(n)$.

Triplet d'Hoare

Un triplet d'Hoare est une formule qui permet de raisonner formellement sur le fonctionnement d'un fragment de code. Un triplet d'Hoare correspond donc à la formule

$$\{\text{Précondition}\}\text{Fragment de code}\{\text{Postcondition}\}$$

La précondition est une formule logique qui doit être vraie avant d'exécuter le fragment de code, et la postcondition est une formule logique qui doit être vraie après avoir exécuté le fragment de code

Invariant de boucle

Un invariant de boucle est une formule logique qui doit être vrai avant d'entrer dans la boucle, à chaque itération de la boucle, et à la fin de la boucle. Lorsqu'on l'utilise avec le triplet d'Hoare, nous pouvons dégager les propriétés suivantes :

- L'invariant est une conséquence de la précondition du triplet. L'invariant est donc vrai chaque fois que la précondition l'est. Précondition $\implies$ Invariant
- Si l'invariant est vrai avant une itération de la boucle, alors l'invariant l'est également après cette itération. $\{\text{Invariant} \wedge \text{Condition}\}\text{Itération}\{\text{Invariant}\}$
- En sortie de boucle, l'invariant implique la postcondition du triplet. L'invariant est donc vrai en sortie de boucle, comme la postcondition. $(\text{Invariant} \wedge \neg\text{Condition}) \implies$ Postcondition

Prouver la terminaison d'un programme

Afin de prouver la terminaison d'un fragment de code, en plus de démontrer que son triplet d'Hoare est valide, il faut aussi démontrer la validité d'un variant de boucle, aussi appelé *fonction de terminaison*. Un variant de boucle doit satisfaire la propriété disant qu'à chaque itération complète de la boucle, on diminue sa valeur. Pour ce faire, on peut s'aider de l'invariant et montrer que le variant va diminuer tout en restant borné par l'invariant.

Exercices

1. Écrire une fonction qui renvoie vrai si la chaine de caractères donnée en entrée est un palindrome et faux sinon.
   - Donner la complexité de cette fonction.
   - Par la méthode des invariants, démontrer que la valeur retournée par la fonction est correcte. Démontrer également que la fonction se termine.
2. Écrire une fonction qui prend en entrée un tableau `t` trié dans l'ordre croissant, sa taille ℓ, ainsi qu'un entier `n`, et qui effectue une recherche dichotomique afin de trouver le nombre `n` dans le tableau `t`. La fonction doit retourner l'indice du tableau contenant le nombre `n`. Si `n` n'est pas présent dans `t`, la fonction doit retourner -1.
   - Donner la complexité de cette fonction.
   - Par la méthode des invariants, démontrer que la valeur retournée par la fonction est correcte. Démontrer également que la fonction se termine.

Exercices

3. (Examen d'août 2021) Écrire une procédure prenant en arguments un tableau `t` de nombres réels, le nombre $n \geq 0$ d'éléments de `t`. L'opération effectuée par cette procédure consiste à modifier le contenu du tableau `t` de la façon suivante : si $t_0, t_1, t_2, \ldots$ sont les contenus initiaux des cases `t[0]`,`t[1]`,`t[2]`,... de `t`, alors après l'exécution de la procédure, la première case de `t` doit contenir t_0, la deuxième $t_0 + t_1$, la troisième $t_0 + t_1 + t_2$, et ainsi de suite.
   - Donner la complexité de cette fonction.
   - Par la méthode des invariants, démontrer que la fonction est correcte. Démontrer également que la fonction se termine.

Exercices

4. Écrire une fonction prenant en arguments un entier `n` strictement positif et qui retourne la factorielle de ce nombre.
   - Donner la complexité de cette fonction.
   - Par la méthode des invariants, démontrer que la valeur retournée par la fonction est correcte. Démontrer également que la fonction se termine.
5. Écrire une fonction prenant en arguments un tableau d'entiers ainsi que sa taille et qui place dans une variable `min` et `max` respectivement le plus petit et le plus grand entier dans le tableau.
   - Donner la complexité de cette fonction.
   - Par la methode des invariants, démontrer que la fonction est correcte. Démontrer également que la fonction se termine.

Indice

Penser aux types de passage en paramètre.