TP3 – Lecture d'adresses

Langage C (LC4)

semaine du 11 février

Question 1. Recopier la macro suivante.

```
#define affiche_nbr(a) printf("%p (%lu)\n", a, (unsigned long) a)
```

Elle sert à afficher un nombre entier en écriture hexadécimale (plus pratique pour repérer les puissances de 2 quand on a l'habitude) et en écriture décimale entre parenthèses (plus pratique pour faire des additions et des soustractions).
Pour afficher l'adresse d'une variable `n`, vous utiliserez l'instruction `affiche_nbr(&n)`, pour afficher l'adresse contenue dans une variable `p` de type pointeur, vous utiliserez `affiche_nbr(p)` (et bien sûr `affiche_nbr(&p)` pour afficher l'adresse de la variable `p`.)

1 Les trois zones de la mémoire

1.1 La pile (*stack*) et la zone statique

Question 2. Déclarez et initialisez les variables suivantes :
– deux variables globales (c'est-à-dire en dehors de toute fonction) de type caractère ;
– deux variables globales de type entier ;
– deux variables globales de type pointeur (`int *` par exemple) ;
– deux variables de type caractère dans la fonction `main` ;
– deux variables de type entier dans la fonction `main` ;
– deux variables de type pointeur d'entier dans la fonction `main`.
Affichez les adresses de chacune de ces variables. Vous remarquerez que :
– les variables globales ne sont pas stockées au même endroit que les variables définies dans une fonction (pour les premières, on l'appelle la zone statique, pour les dernières, la pile).
– les variables se suivent dans la mémoire, mais pas dans le même ordre dans les deux zones de la mémoire.

Question 3. Affichez les valeurs de `sizeof(int)`, `sizeof(char)` et `sizeof(int *)` et comparez avec le nombre d'octets qui séparent les variables déclarées à la question précédente. Vous remarquerez que ce nombre d'octets est parfois supérieur (le processeur manipule les octets par blocs de 4, voire de 8, donc il « arrondit » les adresses pour se simplifier la tâche, on dit qu'il les *aligne*).

Question 4. Écrivez une fonction `void f1(int *p)` dans laquelle vous déclarerez une variable de type `int`, initialisée à 7 et une de type `char`, initialisée à `'A'`. Dans cette fonction, vous afficherez :
– le contenu de l'adresse pointée par le paramètre `p`

– la valeur de `p`
– l'adresse de `p`
Après l'affichage, vous modifierez la valeur de l'entier pointé par `p` *puis* la valeur de `p` (en lui affectant la valeur `NULL`).

Question 5. Dans la fonction `main`, affectez à l'un des pointeurs d'entiers l'adresse d'une des variables entières déclarées à la question 2. Appelez la fonction `f1` dans le `main` en lui passant comme argument ce pointeur. Après l'appel, affichez l'adresse et la valeur de ce pointeur, ainsi que la valeur de l'entier pointé. L'appel à la fonction a modifié l'entier mais pas le pointeur (lors de l'appel, le contenu du pointeur a été recopié, et continue de pointer sur le même endroit de la mémoire).

Question 6. Écrivez une fonction `void f2(int *p)` dans laquelle vous déclarerez deux variables de type `int` non initialisées. Dans cette fonction, vous afficherez :
– l'adresse du paramètre ;
– l'adresse de chacune des variables ;
– le contenu des variables.
Dans la fonction `main`, appelez cette fonction *juste* après avoir appelé `f1`. Vous remarquez que les adresses des paramètres et des variables sont les mêmes que lors de l'appel de `f1` : La place mémoire utilisée par `f1` a été libérée quand la fonction s'est terminée, donc elle est reprise à l'identique pour l'appel à `f2`. De ce fait, les variables non initialisées de `f2` contiennent ce qui avait été mis là par `f1`.

Question 7. Ajoutez à `f1` une variable entière statique initialisée à 9 (`static int n = 9;`) et affichez son adresse en même temps que les autres ainsi que son contenu. Vous remarquerez que cette variable est stockée dans la zone… statique.
Comme dernière instruction de la fonction, vous incrémenterez la valeur de cette variable.

Question 8. Dans la fonction `f2`, appelez la fonction `f1`. Vous constatez que dans ce deuxième appel de `f1`, l'adresse de la variable statique est identique, mais que sa valeur a changé. Contrairement aux autres variables de la fonction, celle-ci a été conservée à la fin du premier appel de `f1`. Par ailleurs, les adresses des autres variables et des paramètres sont différentes : la zone mémoire utilisée par `f2` n'a pas encore été libérée.

1.2 Le tas (*heap*)

Question 9. Dans le `main`(vous pouvez entamer un autre fichier si vous voulez), affichez les adresses renvoyées par cinq appels successifs à la fonction `malloc`, pour réserver successivement 1, 2, 12, 13 puis 1 octet de mémoire.

Les adresses allouées se trouvent dans une nouvelle zones de mémoire appelée le tas. Par ailleurs, vous remarquerez que le nombre d'octets qui sépare deux zones allouées est plus grand que la taille de la zone allouée. La place supplémentaire est occupée par des informations stockées par `malloc`, qui serviront notamment à la fonction `free`.

1.3 Valeur et contenu d'un tableau

Question 10. Dans la fonction `main`, déclarez `int t[3];` puis affichez les adresses `&t`, `&(t[0])`, `&(t[1])`, `t` et `t+1`. Vous remarquez qu'un tableau n'est qu'une variable de type pointeur et les crochets, un opérateur pour accéder à la case pointée et à ses voisines.

Question 11. Affectez une autre valeur (par exemple `NULL`) à `t`. Cela produit une erreur : `t` est défini comme une constante.

Question 12. Modifiez la déclaration de `t` pour en faire une variable pointeur pour laquelle vous allouez une zone de 3 entiers (avec `malloc`). Affichez les mêmes adresses que pour la précédente déclaration de `t`. Les observations précédentes sont toujours valables, à ceci près que `t` ne pointe plus vers la même partie de la mémoire.

Question 13. Affectez une autre valeur (par exemple `NULL`) à `t`, cette fois ça marche, affectez la valeur 12 à `*t`, puis la valeur 13 à `t[0]`, puis affichez le contenu de chacune des deux (en fait il s'agit de la même case mémoire, d'adresse `t`).

Question 14. Dans un nouveau fichier, déclarez `char *s = "hello";` et affichez `s`. Vous constatez que la chaîne `"hello"` est stockée proche de la zone statique, mais pas immédiatement voisine des autres données observées précédemment dans cette zone. Affectez `'H'` à `s[0]`, cela produit une erreur : la zone mémoire elle-même n'est pas accessible en écriture, la chaîne `"hello"` est constante. On peut en revanche affecter une autre adresse à `s`, qui elle n'a pas été déclarée comme constante.

Question 15. Affectez à `s` l'adresse d'un espace mémoire de 10 caractères réservé par `malloc` et recopiez le contenu de la chaîne `"hello"` dans cet espace, au moyen de la fonction `strcpy` (inclure `string.h`, voir TP6).

Question 16. Recopiez la fonction suivante :

```
void aaa(char *s)
{
  while (*s) *(s++) = 'A';
}
```

Appelez cette fonction en passant la chaîne `s` en paramètre puis affichez le contenu de la chaîne avec `printf("%s\n", s);`
Analysez le code de la fonction. Placé à droite de la variable, le `++` incrémente `s` *après* avoir pris sa valeur courante pour calculer le reste de l'expression.
Ici, on parcourt la chaîne sans utiliser un indice (`s[i]`), on se contente d'incrémenter un pointeur. Parfois, les manipulations de tableaux (ou une chaînes de caractères) sont plus simples de cette façon. Lorsqu'on fait un parcours séquentiel d'un tableau ou d'une chaîne, cette méthode est plus efficace.

2 Tableaux de structures et tableaux de pointeurs

Question 17. Recopiez les déclarations suivantes :

```
struct st3 { int a, b, c; };
struct st3 st = {1, 2, 3};
int *t = (int *) &st;
```

Affichez les adresses de la variable `st` et de chacun de ses trois champs. Affichez les trois premières cases du tableau `t`.
Vous constatez que les champs d'une structure sont contigus en mémoire et que s'ils sont du même type, ils peuvent donc être vus comme les cases d'un tableau.

Question 18. Écrivez une fonction `struct st3 clone_st3(struct st3)` qui affiche les adresses des trois champs de la structure passée en paramètres et recopie le contenu de la structure dans une autre structure qui est renvoyée comme résultat. Applelez cette fonction dans le `main`, récupérez sa valeur de retour dans une autre variable dont vous afficherez les adresses des champs.
Notez bien que les adresses sont toutes différentes à chaque fois, ce qui signifie que lors d'un appel de fonction, les valeurs de tous les champs sont recopiées. Quand les structures ont beaucoup de champs, cela peut entraîner beaucoup de recopies inutiles, d'où l'intérêt de prendre en paramètre des pointeurs sur les structures plutôt que les structures elles-mêmes.

Question 19. Définissez un type `pst3` qui désignera un pointeur sur `struct st3`. Modifiez la fonction précédente pour que son paramètre et sa valeur de retour soit un `pst3`. Attention, vérifiez que votre fonction a bien dupliqué chacun des champs de la structure elle-même et non simplement recopié l'adresse.

Question 20. Créez un tableau de 10 éléments de type `struct st3` et affichez, au moyen d'une boucle, les adresses de chacun des trois champs de ces 10 éléments.

Question 21. Créez un tableau de 10 éléments de type `pst3`, affectez à chacune de ses cases l'adresse d'une zone de mémoire allouée pour contenir un `struct pst3`. Affichez les adresses de chacun des trois champs de ces 10 structures ainsi que les adresses de chacune des cases du tableau, et de la variable tableau elle-même.