Langage C et aléa, séance 3

École des Mines de Nancy, séminaire d'option Ingénierie Mathématique

Frédéric Sur
http://www.loria.fr/~sur/enseignement/coursCalea/

1 Les pointeurs

Cette séance est une introduction aux pointeurs en langage C. N'hésitez pas à consulter un ouvrage détaillé.

1.1 Déclaration de pointeurs

Un pointeur est une variable contenant une adresse de la mémoire, généralement celle à partir de laquelle est stockée une variable (numérique, ou une *structure* comme on verra plus loin).

Un pointeur sur une variable de type `type` est déclaré par : `type* pointeur;` (ou `type *pointeur;` ou `type * pointeur...`)

Pour l'affectation d'un pointeur, nous disposons de l'opérateur `&` qui renvoie l'adresse mémoire à partir de laquelle est stockée la valeur d'une variable.

Exemple :

```
double var=3.14;    /* la variable var est de type double */
double* point_var;  /* point_var est un pointeur vers un double */
point_var = &var;  /* on stocke l'adresse mémoire de var dans point_var */
printf("%p \n",point_var);   /* affiche la valeur de point_var (une adresse) */
```

On peut définir des pointeurs sur des pointeurs.

Une fois qu'un pointeur est déclaré et initialisé, on peut accéder au contenu de l'adresse mémoire contenue dans le pointeur grâce à l'opérateur `*` (dit de *déréférencement* ou d'*indirection*), de la manière suivante :

```
int a = 2;     /* a est initialisé a 2 */
int* p = &a;  /* dans le pointeur p on stocke l'adresse de a */
*p = 3;       /* on stocke 3 dans l'adresse pointée par p : c'est celle de a ! */
```

À la fin de l'exécution de ces trois lignes, la variable `a` vaut donc 3.

Dans certains cas, on a besoin du pointeur NULL, qui pointe sur une zone de mémoire fictive non-accessible.

Exemple : `type* var = NULL;`

1.2 Arithmétique des pointeurs

On dispose des opérateurs + et − sur les pointeurs. Attention, il ne faut pas confondre avec leur équivalent sur les variables de type numérique !

Si on ajoute (resp. retranche) une valeur à un pointeur, alors cela revient à lui ajouter (resp. retrancher) le produit de cette valeur par le nombre de cases mémoires utilisées par les variables du type pointé. L'unité est le `char` qui occupe une case mémoire (un octet). Un `int` peut occuper 4 octets, cela dépend du système (vous pouvez étudier la « taille » relative aux types en affichant les valeurs de `sizeof(int)`, `sizeof(double)...`)

Exemple :

```
int a=2;
int* p = &a;
```

Alors p+1 désigne la case mémoire située après les cases qui stockent le contenu de la variable `a`.

Exercice 1 (Arithmétique des pointeurs)

Commentez l'affichage du programme suivant, et comparez les valeurs des pointeurs. Que donnent des executions successives ?

```
#include<stdio.h>
int main(void)
{
  double var=3.14;
  double *point_var, *pointeur2;
  int var2=5, *pointeur3;
  point_var=&var;
  pointeur2=point_var+1;
  pointeur3=&var2+1;
  printf("%d %d \n",sizeof(double),sizeof(int));
  printf("%p  %p %p \n", point_var, &var, pointeur2);
  printf("%p %p \n",&var2,pointeur3);
}
```

1.3 Retour sur les tableaux

Une déclaration d'un tableau à `N` éléments de type `type` revient à réserver une plage de mémoire contiguë pour stocker `N` variables de ce type.

Exemple : `int tab[2]` réserve une plage de mémoire contiguë pour stocker deux entiers.
En fait, la variable `tab` est un pointeur sur le premier élément du tableau.

D'après les deux sections précédentes :
- `tab` contient l'adresse mémoire de la première case du tableau,
- `*tab` est la valeur du premier élément du tableau,
- `tab+1` est l'adresse mémoire de la deuxième case du tableau,
- `*(tab+1)` est la valeur du deuxième élément du tableau,
- etc.

Ainsi, on peut dire que `tab[i]` est un raccourci pour : `*(tab+i)`. Voilà pourquoi les cases d'un tableau sont numérotées de 0 à `taille-1`...

Attention, il n'y a aucune vérification de dépassement des bornes du tableau. On peut ainsi lire (ou écrire) dans la mémoire à des endroits inattendus, éventuellement sur d'autres variables utilisées par le programme. Au mieux le système envoie une *erreur de segmentation*, au pire on ne se rend compte de rien, et entre les deux le programme fait n'importe quoi (enfin pas ce qui est espéré), sans forcément renvoyer d'erreur de segmentation.

Exercice 2 (tableaux et pointeurs)

Écrivez un programme qui déclare un tableau de `double`s de taille 10, et qui initialise l'élément d'indice `i` à i, par une boucle `for` et l'utilisation des pointeurs. Dans la boucle on affichera l'adresse mémoire où est stocké le i-ème élément du tableau.

Interprétez le résultat connaissant la taille d'un `double` (donnée par `sizeof(double)`).

Même question avec un tableau d'entiers.

1.4 Allocation dynamique de la mémoire

La fonction `malloc` renvoie un pointeur sur une zone mémoire contiguë de taille donnée (inclure `stdlib.h`).

Exemple :

```
float* p = malloc(10*sizeof(float));
```

permet de déclarer un pointeur sur un bloc mémoire dans lequel on peut stocker 10 `float`s.

Si l'allocation n'a pas pu se faire (pas de bloc de mémoire contiguë suffisamment grand), `malloc` renvoie le pointeur `NULL`.

Utilisation possible :

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

int main(void) {
  int i;
  float* p = malloc(10*sizeof(float));
  if (p!=NULL) {
    for (i=0;i<10;i++)
      *(p+i)=i/3.0;   /* ceci est equivalent à p[i]=i/3.0 ! */
  }
  else printf("Probleme malloc\n");
  printf("p[2] vaut:%f\n",p[2]);
  free(p);
}
```

La fonction `free(p)` permet de libérer la zone mémoire allouée par `malloc` sur laquelle pointe `p`. Il est nécessaire de libérer explicitement la mémoire lorsque l'on ne s'en sert plus car ceci n'est pas fait automatiquement (pas de *garbage collector* comme en Java). De plus il faut le faire tant que `p` pointe encore vers cette zone mémoire, après ce n'est plus possible.

Imaginez les problèmes si vous allouez de la mémoire dans une boucle appelée un certain nombre de fois, sans jamais la libérer...

Remarque : en fait le test `if (p!=NULL)` peut être remplacé par `if (p)` (rappelez-vous comment sont « simulés » les booléens en C).

2 Passage d'arguments de fonctions

2.1 Passage par valeur

En C, lorsque l'on passe des arguments à une fonction, les valeurs des variables correspondantes sont en fait copiées par la fonction, qui travaille sur ces copies. On parle de *passage par valeur*.

Exemple : (essayez ! qu'est-ce qu'affiche le `printf` ?)

```
#include<stdio.h>

void fonction_inutile(double x, double y) {
  x=100*x;
  y=y+1;
}

int main(void) {
  double arg1=3.4, arg2=1.0;
  fonction_inutile(arg1,arg2);
  printf("%f et %f\n", arg1, arg2);
}
```

2.2 Passage par référence

Un moyen de contourner le "problème" mis à jour dans l'exemple précédent est de passer non pas les *valeurs* des arguments, mais leur *adresse mémoire*. On parle de *passage par référence*.

Exemple : (essayez, et commentez l'affichage)

```
#include<stdio.h>

void fonction_utile(double* x, double* y) {
  *x=100*(*x);
  *y=(*y)+1;
}

int main(void) {
  double arg1=3.4, arg2=1.0;
  fonction_utile(&arg1,&arg2);
  printf("%f et %f\n", arg1, arg2);
}
```

Cette fois on stocke les arguments de la fonction (qui sont les adresses mémoires où sont stockées les variables `arg1` et `arg2`) dans des pointeurs `x` et `y`. Manipuler `*x` et `*y` revient donc à manipuler les valeurs de `arg1` et `arg2`.

Étant donné ce qui a été dit dans la section 1.3, que pensez-vous du passage d'un tableau en argument d'une fonction ?

Remarque 1 : si on a besoin de la taille du tableau dans la fonction, on n'a pas d'autre possibilité que la passer en argument.
Remarque 2 : on ne peut pas renvoyer un pointeur sur une variable locale (qui n'existe pas en dehors de la fonction). On ne peut donc pas retourner directement un tableau déclaré dans une fonction. Le moyen le plus simple de faire est que l'appelant déclare le tableau et le passe en argument à la fonction.

Exercice 3 (passage par référence)

Écrivez une fonction qui échange les valeurs de deux variables de type entier, et renvoie leur somme. Testez.

Exercice 4 (fonctions sur les tableaux)

Écrivez une fonction qui prend comme argument un tableau `tab` et sa taille `t`, et l'initialise de manière à ce que `tab[t]=t`.

Utilisez cette fonction dans un programme pour initialiser un tableau, puis affichez les valeurs stockées pour vérifier que l'initialisation est correcte.

Écrivez une fonction qui prend comme argument un tableau et sa taille, le modifie en multipliant les valeurs stockées par 2, et retourne le plus grand élément. Testez cette fonction.

---

3 Structures

Le langage C ne dispose pas de la notion de classe et d'objet comme Java (ou C++), qui permettent de définir une entité regroupant un certain nombre de variables et de méthodes. On dispose tout de même de la notion de *structure*, qui permet de regrouper des variables.

On déclare une structure (avant le `main`) par :

```
struct nom_de_la_structure
   {
   type_var1 var1;
   type_var2 var2;
   ...
   type_varn varn;
   };
```

`var1`, `var2`,...`varn` sont les *membres* de la structure.

Exemple :

```
struct Personne
   {
   char nom[20];
   int age;
   double taille;
   };
```

On déclare des variables de type structure là où on en a besoin par :

```
struct nom_de_la_structure nom_de_la_variable;
```

Exemple :

```
struct Personne p1;
```

On peut initialiser une structure (dans notre exemple) par :

```
struct Personne p1 = {"Toto", 22, 180};
```

On accède aux membres d'une structure par `.` (point).

Toujours avec notre exemple :

```
printf("%s a %d ans et mesure %f cm.\n",p1.nom,p1.age,p1.taille);
```

affiche : `Toto a 22 ans et mesure 180 cm.`

On a un opérateur d'affectation sur les structures (= comme utilisé précédemment). Par contre, pas d'opérateur de comparaison.

On peut créer des tableaux de structures, et des pointeurs sur les structures.

Remarque : le mot-clé `typedef` permet de créer des « nouveaux » types. Dans notre cas, cela simplifie la déclaration. Ainsi, dans l'exemple précédent il est possible de déclarer la structure `Personne` par :

```
typedef struct
   {
   char nom[20];
   int age;
   double taille;
   } Personne;
```

Les variables de type `Personne` seront alors déclarées comme toute variable de type classique par :
`Personne p1;`

Exercice 5 (type complexe)

Définissez une structure permettant de représenter les nombres complexes.

Écrivez des fonctions pour les calculs usuels sur les complexes.

Modifiez votre programme d'estimation de l'aire de l'ensemble de Mandelbrot pour intégrer cette structure et ces fonctions.

Exercice 6 (liste chaînée triée)

On considère la structure suivante :

```
typedef struct liste
   {
   int valeur;
   struct liste *suivant;
   } liste;
```

Que contient cette structure ?

Une liste va être représentée par le pointeur sur son premier élément. On initialise une liste par :
`liste* maliste = NULL;`

Le « maillon » d'une liste se déclare et initialise de la façon suivante :

```
liste* maillon = malloc(sizeof(liste));
*maillon.valeur = 10;
*maillon.suivant = autre_maillon;
```

Remarque : `*maillon.valeur` se raccourcit en `maillon->valeur`

On voudrait insérer 10 entiers au fur et à mesure dans cette liste de manière à ce qu'ils soient triés dans l'ordre décroissant, par un appel à :
`inserer(&maliste,a);`
où `a` contient la valeur de l'entier à insérer au bon endroit, et `&maliste` est l'adresse de `maliste` (passage par référence).

Écrivez la fonction `inserer`. Écrivez également une fonction pour afficher à l'écran les valeurs stockées dans la liste.
