1 Pointeurs en C

Un pointeur en C est une variable (ou une constante) dont **la valeur représente l'adresse d'une autre variable**.

1.1 Déclaration d'un pointeur

Comme toute variable, un pointeur doit être déclaré :

```
int* p;
```

On déclare un pointeur en mettant le **type de la variable pointée** suivi d'une étoile. On signale ainsi que p est une variable pointeur qui est sensée pointer un entier.

On peut également positionner l'étoile (pour la déclaration du pointeur près du nom du pointeur, ainsi :

```
int *p;
```

Attention : quel que soit l'écriture c'est p le pointeur !

1.2 Initialisation d'un pointeur

Un pointeur non initialisé à une valeur indéterminée qui est appelée NULL :

Voici comment on initialise un pointeur pour désigner une variable :

```
int i,j; // voici quelques variables...
p=&j; //p contient maintenant l'adresse de j!
p=&i; //p contient maintenant l'adresse de i!
```

1.3 Utilisation d'un pointeur pour manipuler la variable pointée

Pour atteindre la valeur de la variable pointée par p on utilise la syntaxe *p par exemple :

```
printf("%d",*p); //on lit la valeur
                 // pour l'afficher
*p=15; //on ecrit dans la valeur
(*p)++; //on incremente la valeur!
```

1.4 Pointeur void*

On peut déclarer un pointeur comme étant sans type : dans ce cas il désigne simplement une adresse mémoire sans référencer une valeur avec sa taille.

```
void* p; //p represente un pointeur sans type
```

2 Utilité des pointeurs

Les pointeurs sont utilisés chaque fois qu'on a besoin d'atteindre un emplacement mémoire sans pouvoir forcément savoir comment désigner cet emplacement (par un nom de variable par exemple).

Ceci a lieu essentiellement lors d'appels de fonctions.

2.1 Passage de paramètres par adresse à une fonction

Lorsque on appelle une fonction en C, les paramètres sont passés par copie des valeurs de variables (ou de constantes) à la fonction qui recopie ces valeurs dans la pile dans les variables locales de la fonction. Si on désire modifier les variables passées en paramètre dans la fonction on est confronté à un problème !

Exemple :

```
void swap(int i, int j)
    {
    int temp=i;
    i=j;
    j=temp;
    }

    // exemple d'utilisation de cette fonction:
    int x=5;
    int y=10;
    swap(x,y); // on appelle swap en passant x et y en parametres
    printf("x=%d y=%d\n",x,y);
    // Affiche quoi?
```

Cette façon de faire échoue, parce que la fonction swap() échange les copies de x et y, mais pas x et y !

Pour échanger x et y, il faut qu'on transmette à la fonction swap des paramètres qui représentent les adresses des variables dont on veut échanger les valeurs :

```
void swap(int* pi, int* pj)
    {
    int temp=*pi; //on copie la valeur pointee par pi dans temp!
    *pi=*pj; // on copie la valeur pointee par pj dans la valeur pointee par pi
    *pj=temp; // on remet la valeur copiee dans temp dans la valeur pointee par pj
    }

    // exemple d'utilisation de cette fonction:
    int x=5;
    int y=10;
    swap(&x,&y); // on appelle swap en passant les adresses de x et y en parametres
    printf("x=%d y=%d\n",x,y);
    // Affiche x=10 y=5 !!
```

Donc on utilise un paramètre de type pointeur si la fonction désire modifier la donnée passée en paramètre.

3 Les pointeurs et les tableaux

3.1 Que représente le nom d'un tableau en C ?

Le nom d'un tableau est un pointeur constant pointant sur la valeur (typée en général) de la première case du tableau. Par exemple :

```
int tab[10];
// tab est de type int*
// tab represente l'adresse de tab[0]
// tab et &(tab[0] sont synonymes
// tab=... est interdit, car tab est constant!
```

ceci explique les prototypes de certaines fonctions, comme strlen :

```
int strlen(char* str);

//exemple d'appel de cette fonction
char chaine[]="Bonjour";

printf("%d",strlen(chaine));
// on voit qu'on passe en parametre le nom du tableau de caracteres!
```

A remarquer que lorsqu'on désire passer un tableau en paramètre, pour éviter une copie massive (pour des gros tableaux) il vaut mieux passer en paramètre une adresse, à savoir l'adresse du premier élément du tableau !

3.2 Arithmétique du pointeur

Si un pointeur pointe vers une case de tableau, vous pouvez lui ajouter ou lui soustraire un entier. cet entier représente le nombre de cases dont on veut déplacer le pointeur.

```
// on suppose que p pointe une case d'un tableau
p++; // p pointe la case suivante
p--; // p pointe la case precedente
p=p+3; //p pointe trois cases de tableau plus loin
p=p-4; //p pointe quatre cases de tableau plus haut
```

Attention : ça n'a de sens que si p pointe dans un tableau

Remarques :
— C ne fait aucune vérification de bornes lorsque vous déplacez le pointeur
— Si tab est un tableau tab+1 a un sens, mais tab=tab+1 ou tab++ n'en a pas !
— Si p1 et p2 pointent tous les deux le même tableau p2-p1 représente le nombre de cases entre les 2.

4 Allocation dynamique de mémoire

4.1 Le problème

Dans une application informatique on ne connaît pas toujours à l'avance la quantité de mémoire nécessaire pour faire telle ou telle action. Surtout en ce qui concerne les tableaux ! Par exemple si votre application est un traitement de texte, vous voulez charger le document depuis un fichier. Qu'utilisez-vous comme taille de tableau pour charger tout le texte ? Ça va dépendre bien-sûr de la taille du fichier ! D'où le problème !

```
// on recupere la taille du fichier
n=...
Ensuite on essaie de faire :
char document[n];
// sauf que ca ne marche pas : n doit etre une constante ou defini par un define.
```

C'est pourquoi il faut un mécanisme permettant de chercher à s'allouer un bloc mémoire au moment de l'exécution du programme et non au moment de la compilation. Dans le cas du traitement de texte, on pourra regarder d'abord la taille du fichier, puis on fera l'allocation du bloc mémoire nécessaire.

On parle d'allocation dynamique de mémoire.

4.2 La fonction malloc

La fonction malloc permet d'allouer un bloc de mémoire libre de la RAM et de retourner l'adresse de ce bloc Il suffit alors de transtyper correctement ce paramètre adresse pour l'adapter au type de données souhaité dans le bloc alloué. Voici le prototype de la fonction malloc :

```
void* malloc(int size);
```

Le paramètre size : est le nombre d'octets désirés dans le bloc.

La valeur de retour est un pointeur void* (sans type) et vaut NULL si l'opération échoue.

4.3 Utilisation de malloc pour allouer dynamiquement un tableau à 1 dimension

Supposons qu'on veuille allouer un tableau de n réels (n étant le nombre de réels désiré dans le tableau : c'est une variable). Le code suivant fait l'affaire :

```
float* p;
p=(float*)malloc(n*sizeof(float));
if(p!=NULL)
    {
    //tout s'est bien passe: p pointe le debut du tableau alloue
    }
else
    {
```

```
    //probleme!
    }
```

4.4 Comment libérer le bloc de mémoire alloué dynamiquement

Il est nécessaire de libérer (redonner la mémoire à l'OS). Pour cela il ne faut pas perdre la valeur de l'adresse retournée par malloc() ! Donc moralité : ne touchez pas à la valeur de p.

Pour libérer le bloc mémoire :

```
free(p); //libere le bloc
p=NULL; // indique que le pointeur est non valide (bloc libere)
```

Dans cet exemple on affecte p avec NULL à la fin, car si par malheur vous ne le faites pas et que vous tentiez d'exécuter l'instruction free(p) ; une 2 fois le comportement sera complètement imprévisible. Si par contre vous l'exécutez avec p étant NULL il n'y a aucun risque !

4.5 La fonction realloc

Il est souvent nécessaire pendant l'exécution du programme d'agrandir l'espace d'un bloc. Comme on ne peut le faire en ajoutant simplement de l'espace, on doit :

— allouer un nouveau bloc plus grand,
— copier les données de l'ancien dans le nouveau
— libérer l'ancien bloc

Sauf que ce travail est réalisé par la fonction système realloc() !

Voici le prototype de realloc() :

```
void *realloc (void *ptr, int size);
```

— ptr pointant l'ancien bloc
— size est la taille en nombre d'octet du bloc à réallouer
— la valeur de retour est l'adresse du premier octet du nouveau bloc
— si realloc() échoue, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.
— realloc marche aussi si size est inférieur à la taille de l'ancien bloc

5 Les pointeurs de structure

Supposons une structure :

```
typedef struct{
char nom[500];
char prenom[500];
int age;} Personne;
```

Si on désire passer une variable de ce type à une fonction, on voit bien que la copie nécessite une grande quantité de données. C'est plus simple de passer une adresse de structure. C'est d'ailleurs le même problème que les tableaux.

Lorsqu'on accède à un élément de type structure avec le nom de la variable on utilise l'opérateur de résolution .

```
 Personne pers1;
 printf("Nom:%s Prenom:%s Age%d",pers1.nom,pers1.prenom, pers1.age);
```

Si par contre on y accède par l'intermédiaire d'un pointeur on utilise l'opérateur de résolution ->

```
 Personne pers1; //une personne
 Personne* p=&pers1; //un pointeur qui pointe cette personne
 printf("Nom:%s Prenom:%s Age%d",p->nom,p->prenom, p->age);
```