Langage C

Les pointeurs

L2 Mathématique et Informatique

Université de Marne-la-Vallée

n

Pointeurs en C

- Soit `T` un type.
  Un pointeur sur `T` est l'adresse d'un élément de type `T`. La valeur d'une adresse est strictement positive.
- La déclaration d'un pointeur est de la forme :
  `T* p;`.
  Un emplacement mémoire pour la variable `p` est réservé, il faut lui affecter une adresse valide.
  Lors du passage par adresse à une fonction, l'affection à une adresse valide se fait à l'appel.

```
1  int x=5;
2  int *p;
3  /* p pourra contenir l'adresse d'un int*/
4  /* l'instruction *p=1; est une erreur a ce niveau*/
5  /* p ne contient pas encore une adresse valide*/
6  p=&x;/* le contenu de p est l'adresse de x*/
7  printf("%d %d\n",x,*p);
8  *p=3;/* mettre la valeur 3  a l'adresse p*/
9  printf("%d %d\n",x,*p);
```

affiche

```
5 5
3 3
```

Pointeur et tableau

Un tableau est l'adresse (constante) du premier élément du tableau.
Si `t` est un tableau, `t` et `&t[0]` ont même valeur.
La notation `t[i]` désigne le $i^{\text{ème}}$ élément après l'adresse `t`.
Tout pointeur sur une zone d'éléments consécutifs peut être utilisé comme un tableau à l'aide de cette notation.
C'est au programmeur de vérifier la cohérence de cette utilisation pour le type et le nombre des éléments manipulés.

```
1
2  int  t[N] ;/* t est l'adresse d'une zone de N int ,*/
3  int *p;/* p peut contenir une adresse d'int*/
4  p=t; /* p contient l'adresse de T[0]*/
5  for(i=0;i<N;i++)
6      p[i]=i;
7  /*remplit le tableau t avec les entiers consecutifs*/
```

Le type pointeur générique

`void *`
Une variable de type `void *` peut recevoir n'importe quelle adresse.
Le type `void *` est utilisé par les fonctions de gestion dynamique de la mémoire (le tas).
Il permet également l'écriture de fonctions génériques, fonctions pouvant agir sur des données de n'importe quel type.
Les opérateurs suivants sont souvent nécessaires à l'écriture de telles fonctions:

- l'opérateur `sizeof(UnType)` détermine la taille en octet d'un élément de type `UnType`. `sizeof(char)` vaut 1;
- l'opérateur de coercition :
  `(UnType)` placé devant une variable force le compilateur à voir la variable comme un élément du type `UnType`.

Une fonction générique pour l'échange de deux valeurs de type quelconque `MemSwap`

.

- Principe:
  toute valeur est une suite d'octets.
  La fonction va considérer ces suites d'octets comme des tableaux de caractères et les échanger case par case. Elle doit donc connaître la taille en octets des valeurs à échanger.
- Mise en oeuvre
  La fonction reçoit l'adresse de deux éléments ainsi que leur taille.
  Le type des éléments est inconnu, l'adresse est de type `void *`
  À l'aide d'un cast, elle indique que les adresses reçues sont à considérer comme des adresses de caractères `char *`. Les deux adresses sont utilisées comme des tableaux. On parcourt en échangeant case le contenu des cases.

```
1  void MemSwap(void *Un ,void *Deux,int taille){
2     char *U,*D;/* pointeurs utilises comme tableau*/
3     char tmp;/* pour l'echange*/
4     int i;
5     U=(char *)Un;
6     D= (char *)Deux;
7     for (i=0;i<taille;i++){
8        tmp=U[i];
9        U[i]=D[i];
10       D[i]=tmp ;
11     }
12 }
```

Si `p` et `q` sont des `int` l'appel est `MemSwap(&p,&q,sizeof(int))`.
Si `x` et `y` sont des structures de type `Truc` l'appel est
`MemSwap(&x,&y,sizeof(Truc))`

Une fonction de tri générique

```
$>man qsort

NAME
       qsort - sorts an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size\_t nmemb, size\_t size,
                  int(*compar)(const void *, const void *));
DESCRIPTION
       The  qsort()  function sorts an array with nmemb elements of size size.
       The base argument points to the start of the array.

       The contents of the array are sorted in ascending order according to  a
       comparison  function  pointed  to  by  compar, which is called with two
       arguments that point to the objects being compared.

       The comparison function must return an integer less than, equal to,  or
       greater  than  zero  if  the first argument is considered to be respec-
       tively less than, equal to, or greater than the second.  If two members
       compare as equal, their order in the sorted array is undefined.
```

L'allocation dynamique de mémoire

Des espaces mémoire peuvent être alloués en cours d'exécution d'un processus.
Une partie de la mémoire associée à un processus est réservée pour l'allocation dynamique : le tas.
Ces espaces sont accessibles grâce à leur adresse jusqu'à l'ordre explicite de leur libération.
Les 2 fonctions principales d'allocation de mémoire déclarée dans < stdlib.h>

- `void * calloc (size_t nmemb, size_t size);`
- `void * malloc (size_t size);.`

Leur valeur de retour est `void *` puisque le type des éléments destinés à être placés à l'adresse obtenue est inconnu. Le type `size_t` est compatible avec les entiers.

- `void * calloc (size_t nmemb, size_t size);`
  `calloc()` alloue la mémoire nécessaire pour un tableau `nmemb` éléments, chacun d'eux représentant `size` octets, et renvoie un pointeur vers la zone allouée. Cette zone est remplie avec des zéros.
- `void * malloc (size_t size);`
  `malloc()` alloue `size` octets, et renvoie un pointeur sur la zone allouée. Le contenu de la zone de mémoire n'est pas initialisé.

En cas d'échec les deux fonctions renvoient la valeur `NULL` qui correspond à l'adresse 0, non valide.
NULL est une constante symbolique de la librairie standard définie par:

```
#define NULL (void *)0
```

Toujours tester la valeur de retour de ces fonctions avant d'utiliser la place mémoire.

```
if((ptr=(type *)calloc(nb,sizeof(type)))!=NULL)
```

Realloc

- `void *realloc(void *ptr, size_t size);`
- réalloue la zone pointée par `ptr` à la nouvelle taille `size`

  - anciennes données conservées
  - ou tronquées si la taille a diminuée
- possibles recopies de données si l'emplacement est changé (avec le coût que cela produit...)
- `ptr` doit pointer sur une zone valide !

Libération de l'espace dynamiquement alloué

Pas de *ramasse-miettes* en C. L'espace reste réservé tant que l'on n'indique pas explicitement qu'il n'est plus utilisé.
La place en mémoire n'étant pas infinie, le programmeur doit veiller à libérer toute zone qui n'est plus utilisée.
La libération d'une zone précédemment allouée s'effectue avec la fonction `void free (void * ptr);`.
`ptr` doit être l'adresse d'une zone qui a été allouée par une des fonctions d'allocations `malloc(), calloc(), realloc()`.

Utilisation avec des chaînes de caractères

Avec des grands tableaux de nombreuses cases peuvent rester inutilisées. On va écrire une fonction de lecture d'une chaine d'au plus 1024 lettres et allouer l'espace juste nécessaire pour mémoriser la chaîne.

- Principe:
  On lit la chaîne dans un tableau de `MAXLETTRES char`;
  on détermine la place nécessaire pour la chaine saisie; on alloue dynamiquement une zone de cette taille; on recopie la chaine saisie à cet emplacement; on donne au paramètre l'adresse de cet emplacement; on renvoie la longueur de la chaine.
- Mise en oeuvre
  Une chaîne de caractères est l'adresse d'un caractère (`char *`).
  Ce paramètre doit être changé par la fonction, il faut donc envoyer son adresse. La fonction reçoit donc un (`char **`)
  La fonction `size_t strlen(char s[])` renvoie le nombre de caractères avant '\0'+ dans une chaine de caractères.
  La fonction `char* strcpy(char *dest,char *source)` copie la chaîne `source` dans la chaîne `dest`.
  Pour une constante entière n, le format `%ns` dans la fonction `scanf` permet de lire au plus n caractères et d'en faire une chaîne. Il faut donc prévoir un tableau de taille au moins $n + 1$.

```
1  int LireChaine(char **s)
2  {/*place a l'adresse *s, l'adresse d'une nouvelle chaine
3    de caracteres*/
4    char lu[1025];
5    int nbchar;
6    scanf("%1024s",lu);//une place pour '\0'//
7    nbchar=strlen(lu);
8    *s=(char *)malloc(nbchar+1);//une place pour \0//
9    if (*s!=NULL)
10     strcpy(*s,lu);
11    else
12     nbchar=-1;
13 return nbchar;
14 }
15 ...
16 char *chaine;
17  n=LireChaine(&chaine);
```

Tableau dynamique

Une zone mémoire peut être utilisée comme tableau; on peut donc définir dynamiquement (en cours d'exécution) avec une taille adaptée au besoin.
(`Element` est le type des objets manipulés)

```
1  printf("nombre d'elements :")
2  scanf("%d",&taille);
3  if ((zone=(Element *)calloc(taille,sizeof(Element)))==NULL)
4     fprintf(stderr,"plus de place memoire\n");
5  else{
6     for(i=0;i<taille;i++)
7        lireElement(&zone[i]);
8  ....
9        }
```

Pour une utilisation plus générale, on peut définir une structure contenant toutes les informations nécessaires à la manipuilation du tableau dynamique,
par exemple pour des ensembles d'entiers(non triés) :

```
1  typedef struct{
2     int *valeur;/* la zone en memoire*/
3     int max;/* la place disponible*/
4     int taille; /* la place utilisee*/
5  }Ensemble;
```

Pour éviter des appels fréquents aux fonctions d'allocations, on peut allouer la mémoire par blocs de BLOC éléments:

```
1  int initialiseEnsemble(Ensemble *e){
2     if ((*e).valeur=(int *)calloc(BLOC,sizeof(int))==NULL)
3        return 0;
4     (*e).max=BLOC;
5     (*e).taille=0;
6     return 1;
7  }
```

```
1  int ajouteEnsemble(Ensemble *e,int x){
2    if (appartientEnsemble(*e,x)==1)
3      return 1
4    if ((*e).taille==(*e).max){/*plus de place*/
5      int *tmp;
6      if((tmp=realloc((*e).valeur,((*e).max+BLOC)*sizeof(i
7          return 0;/* plus de place pour x*/
8      (*e).valeur=tmp;
9      (*e).max= (*e).max+BLOC;
10   }
11   (*e).valeur[(*e).taille]=x;
12   (*e).taille++;
13   return 1;
14 }
```