Pointeurs génériques void *

Parfois nous ne connaissons pas de type exact de pointeur ou nous écrivons une fonction dont le paramètre est un pointeur qui peut pointer vers différents types de données. Dans ce cas on utilise le pointeur générique `void *`. Pour déclarer une variable de ce type:

```
void * pg;
```

Avertissement. L'arithmétique de pointeurs ne s'applique pas aux pointeurs génériques. Nous pouvons faire une conversion entre un pointeur générique et un autre pointeur sans faire le changement de type:

```
  void *pg;
  int *pl, *pt;
  int l=6;

  pl = &l;
  pg=pl;  /* pointeur generique = pointeur */
  pt=pg;  /* pointeur = pointeur générique */
```

Par contre pour faire une affectation entre deux pointeurs de types différents (mais non génériques) il faut faire une conversion explicite de types:

```
char *pc;
int *pl;
....
pl = (int *)pc;
....
pc = (char *)pl;
```

Réduction d'un tableau vers un pointeur lors d'un passage par paramètre

A l'intérieur d'une fonction dont un paramètre est un tableau comme par exemple dans

```
double somme(double tab[],  ...){
    ...
}
```

ce tableau se réduit à un pointeur, c'est-à-dire C voit cette fonction comme

```
double somme(double *tab, ...){

}
```

où tab donne l'adresse du premier élément du tableau. Si on applique par exemple à tab l'opérateur **sizeof** à l'intérieur de la fonction somme cette opérateur retournera la valeur `sizeof(double *)`, c'est-à-dire la taille d'un pointeur et non la taille du tableau.

L'opérateur `sizeof`

```
sizeof(type)
```

donne le nombre d'octets nécessaires pour stocker une donnée du type type. Par exemple `sizeof(int)` donne le nombre d'octets de mémoire pour stocker un entier int. Notez: en C nous avons toujours `sizeof(char)` égal 1. `sizeof exp` évalue le type de l'expression et donne le nombre d'octet nécessaires pour stocker une donnée de ce type. Applications: dans

```
int tab[]={-3, 6, 5, 2, 1, 9,-32};
int taille = sizeof tab / sizeof(int);
```

sizeof tab donne la taille de `tab` en octets divisé par la taille de int en octets nous donne dans la variable taille le nombre d'éléments de tab. Encore mieux:

```
int taille = sizeof tab/sizeof tab[0];
```

reste correct même si on change le type de tab de int vers un autre type.

Définition d'un type avec typedef

Soit

```
struct toto{
  int nombre;
  double alpha;
  double *tab;
};
```

une définition d'une structure. Dans ce cas une variable «pointeur vers struct toto» sera déclarée comme

```
struct toto *p;
```

`typedef` nous permet de définir un type «pointeur vers struct toto»

```
typedef struct toto *p_toto;
```

et maintenant nous pouvons déclarer une variable `x` de `type p_toto`

```
p_toto p;
```

En général une déclaration de type est de forme `typedef type nom_de_type;` Dans l'exemple précédent `struct toto *` c'était le type et `p_toto` le nom que nous lui avons donné dans typedef.

Pointeurs de structures et opérateur ->.

Soit

```
struct toto{
   double prix;
   int numero;
};
typedef struct toto toto;
typedef toto *p_toto;

toto a;
p_toto pa;
```

Donc a c'est une variable de type structure. Pour accéder aux champs de a on utilise l'opérateur `. (point)` . c'est-à-dire

```
a.numero = 30;
a.prix=9.99;
```

La variable `pa` c'est un pointeur vers une structure `toto` . Nous pouvons faire par exemple

```
pa = malloc(sizeof(toto));
```

pour allouer la mémoire nécessaire pour stocker une structure de type `toto` et ensuite

```
pa->numero = 25;
pa->prix = 99.99;
```

pour remplir les deux champs. C'est-à-dire si on a l'adresse (pointeur) d'une structure alors pour accéder aux champs de cette structure on utilise l'opérateur `->`

Allocation dynamique de la mémoire.

```
#include <stdlib.h>
void *malloc(size_t nb_octet);
void *calloc(size_t nb_elem, size_t taille_elem);
void *realloc(void *adr, size_t taille);
void free(void *adr);
```

La fonction `malloc` alloue une zone de mémoire de taille nb_octets d'octets. Elle retourne l'adresse de la zone allouée ou NULL en cas d'erreur.

La fonction `calloc` alloue une zone de mémoire permettant de stocker nb_elem de taille taille octets chacun. (Donc au total elle alloue nb_elem * taille octets.) Les octets de la zone allouée sont initialisés à 0. Elle retourne l'adresse de la zone allouée ou NULL en cas d'erreur.

La fonction `realloc` alloue une zone de mémoire de taille taille d'octets. Cette zone est initialisée avec le contenu de la mémoire dont l'adresse est donné par le premier paramètre adr. La fonction realloc retourne l'adresse de la zone allouée en cas de succès ou NULL en cas d'erreur. Si le paramètre adr est NULL la fonction realloc se comporte comme malloc. Si le premier paramètre est différent de NULL alors ce paramètre doit donner l'adresse d'une zone de mémoire allouée auparavant avec un appel à malloc, calloc ou realloc. Si *. le paramètre adr est différent de NULL et* . l'appel à realloc est réussi realloc libère la mémoire à l'adresse adr, l'adresse adr n'est plus valable. Par contre, en cas d'échec de realloc l'adresse adr reste valable.

La fonction `free` libère une zone mémoire de l'adresse adr. Le paramètre adr doit indiqué une adresse valable d'une zone mémoire allouée auparavant avec `malloc, calloc ou realloc` . Si adr est NULL la fonction free ne fait rien. Chaînes de caractères en C

Rappelons d'abord que `sizeof(char)=1` , c'est dire un caractère occupe un octet de la mémoire. Une constante chaîne, comme `"abcdef"` , est de type `char *` . Dans la mémoire elle est stockée comme une suite de caractères terminée par le caratère `'\0'` . Donc une instruction

```
char *p = "abcdef" ;
```

a pour l'effet de mettre dans la variable `p` **l'adresse du premier caractères de la chaîne :**

Nous pouvons mettre aussi une chaîne de caractères dans un tableau de caractères:

```
char tab[] = "xyzwupqr" ;
```

déclare et initialise un tableau de caractères :

Notez que **le dernier élément du tableau tab contient bien le caractère** `'\0'` . Les fonctions de la bibliothèque standarde qui agissent sur les chaînes sont regroupées dans `string.h` La fonction

```
#input <string.h>
size_t strlen(const char* cs);
```

retourne la longueur de la chaîne `cs` . Plus exactement elle compte les caractères à partir de l'adresse `cs` jusqu'à la plus proche occurrence de `'\0'` . Le caractère `'\0'` n'est pas compté. Par exemple

```
char t[] = "alamx6kota";
printf("longueur=%d\n", length(t) ); /* affiche 10*/
t[4] = '\0'; /*remplace 'x' par '\0'*/
printf("longueur=%d\n", length(t) ); /* affiche 4*/
```

Les fonctions de conversion de chaînes.

```
#include <stdlib.h>
double atof(const char *s);
int atoi(const char *s);
```

La fonction `atof` convertit la chaîne `s` en un nombre `double`, la fonction `atoi` fait une conversion vers `int`.

Teste et conversion de caractères

Les fonctions suivantes vérifient si le caractère passé en paramètre appartient à une classe de caractères (et retournent 1 en cas de réponse positive, 0 sinon) :

```
#include <ctype.h>
int isalpha(int c) /* lettres    */
int isupper(int c) /* majuscules */
int islower(int c) /* minuscules */
int isdigit(int c) /* chiffre    */
int isalnum(int c) /* lettre ou chiffre */
int isspace(int c) /* espace     */
```

Les fonction suivantes font une conversion majuscule <--> minuscule:

```
#include <ctype.h>
int tolower(int c) /* vers minuscule */
int toupper(int c) /* vers majuscule */
```

et retourne le caractère après la conversion (plus exactement l'octet du poids faible de la valur `int` retournée contient le caractère après la conversion). Les deux fonctions retournent leur paramètre `c` non converti si `c` n'est pas une lettre.

Paramètres de la fonction main

La fonction main peut-être définie comme

```
int main(int argc, char *argv[])
```

Le paramètre `argc` donne le nombre d'éléments dans le tableau `argv` de chaînes de caractères (en fait le tableau contient `argc+` 1 éléments. Le tableau `argv` est initialisé de façon suivante: Supposons que notre programme qui contient la fonction `main` soit dans un fichier `toto.c` et après la compilation l'exécutable soit dans un fichier `toto` (ou toto.exe si par hasard vous travaillez sous `Win$$`). Maintenant supposons que l'on exécute `toto` en tapant la commande

```
toto  ala -34r -o sortie
```

`argv` Dans ce cas la fonction main reçoit en paramètre le tableau `argv` initialisé comme:

c'est-à-dire `argv[0]` est initialisé avec le nom de l'exécutable, les paramètres de la lignes de commande alimentent les éléments `argv[1] ... argv[argc - 1]` et `argv[argc] contient NULL`.