Rappel pointeurs

September 27, 2014

Variables de type pointeur

```
int a;
double b;
int *p_int;
double *p_db;
```

Variables de type pointeur

```
int a;
double b;
int *p_int;
double *p_db;

a=6;
d=-3.14;
```

Opérateur &

```
int a;
double b;
int *p_int;
double *p_db;

a = 6;
d = -3.14;

p_int = &a;
p_db = &b;
```

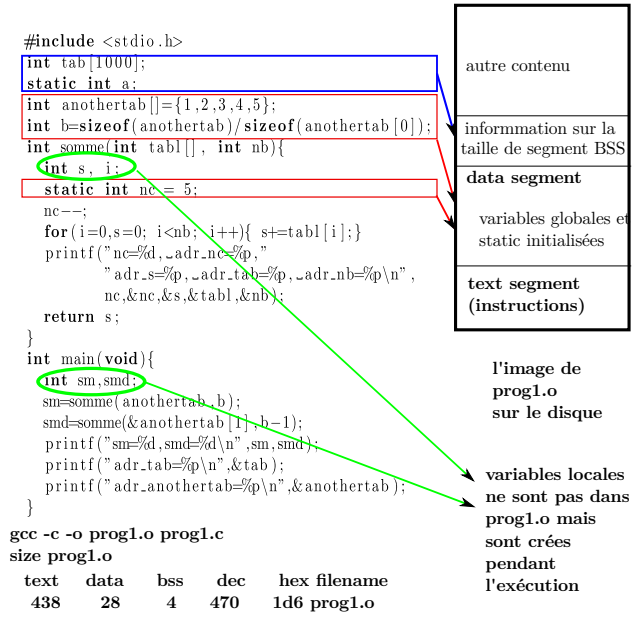
Opérateur *

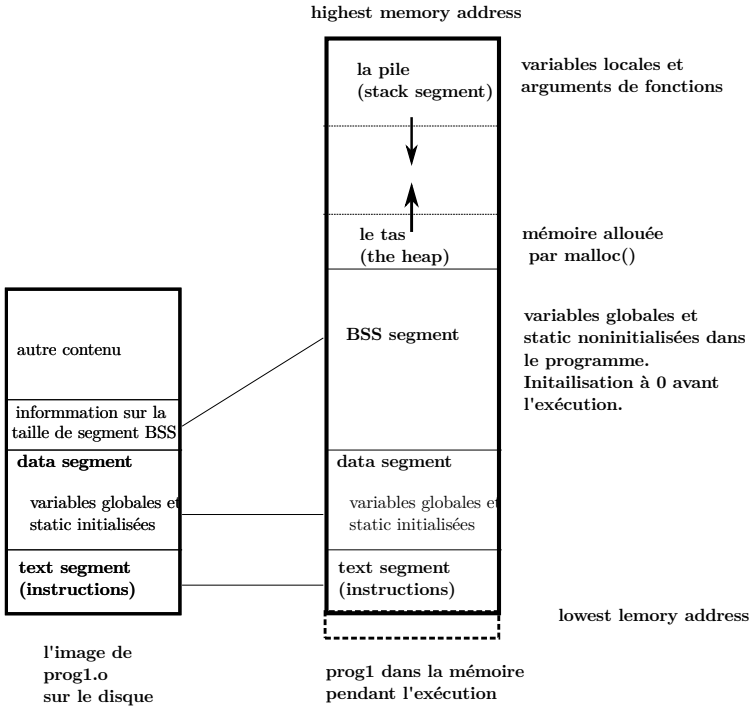
Opérateur * appliqué à un pointeur p retourne la donnée stockée à l'adresse p.

```
int j;
j = *p_int + 5;
```

p_int est de type int *
*p_int est de type int

```
p_db = &d;
*p_db = 9.98;
```





Pointeurs génériques

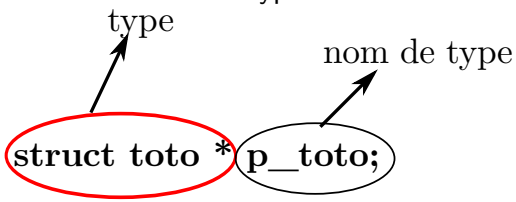
`void *pg;`
Pointeur générique utilisé pour pointer vers les données de type quelconque.
Avertissement: pas d'arithmétique de pointeurs avec les pointeurs génériques.
Comparer deux pointeurs génériques avec == et !=.
Affectation entre pointeur générique et pointeur quelconque :

```
void *pg;
int *pa, *pb;
int l = 6;
pa = &l;
pg = pa;
pb = pg;
```

Définition de type avec typedef

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

p pointeur vers une donnée de type `struct toto`.




```
p_toto a;    /*a - une variable de type pointeur
                 vers struct toto */
```

a = malloc(sizeof(struct toto));

Pointeurs de structures

```
struct toto{
   double prix;
   int numero;
};
struct toto a;
struct toto *p, *q;

a.prix = 99.99 ;
a.numero = 5 ;
p = &a;
q=malloc(sizeof(stuct toto));
q->prix = a.prix * 1.1;
q->numero = p->numero + 2;
```

Les deux dernières instructions peuvent être écrites comme

```
(*q).prix = a.prix * 1.1 ;
(*q).numero = (*p).numero + 2;
```

Allocation dynamique de la mémoire

```
void *malloc(size_t nb_octets)
void *calloc(size_t nb_elem, size_t taille_elem)
void *realloc(void *adr, size_t taille)
void free(void *adr)
```

`malloc, calloc,realloc` retournent `NULL` en cas d'erreur.
`malloc` alloue la mémoire de taille `nb_octets` d'octets et retourne l'adresse de bloc alloué
`calloc` alloue `nb_elem * taille_elem` octets. Les octets alloués sont initialisés à `0`.
`free` libère la mémoire allouée avec une de fonctions de la famille `malloc`. `adr` une adresse valable obtenue par un appel à `malloc`.

realloc

```
void *realloc(void *adr, size_t taille)
```

`realloc` alloue une zone de mémoire de taille `taille` et retourne l'adresse de la mémoire allouée. La fonction recopie les données se trouvant à l'adresse `adr` (la zone de mémoire `adr` devait être obtenu par un appel une fonction de la famille `malloc`. La fonction libère la mémoire dont l'adresse est `adr`. L'adresse `adr` ne sera plus valable.
En cas d'erreur la fonction retourne `NULL` et `adr` reste valable.

realloc (exemple)

```
1  struct tab{
2    int nb_elem;
3    int *t;
4  };
5  struct tab m; int *tmp;
6  m.nb_elem = 10; m.t = malloc(sizeof(int) * m.nb_eler
7  for(i = 0 ; i < m.nb_elem ; i++)
8    m.t[i]=i;
9  ....
10 tmp=realloc(m.t, sizeof(int)*2*m.nb_elem);
11 if(tmp == NULL){/* traiter les problemes de realloc
   }
12 else{ /* si realloc a reussi*/
13   m.nb_elem *= 2;
14   m.t = tmp;
15 }
```

Copier une zone de mémoire

```
#include <string.h>
void *memcpy(void *dest, const void *source,
             size_t taille)
void *memmove(void *dest, const void *source,
              size_t taille)
```

`memcpy`, `memmove` copient `taille` octets de l'adresse `source` vers l'adresse `dest`. Si les deux zones se recouvrent il faut utiliser `memmove`.

```
int tab[ ]={1,2,3,4,5,6,7,8,9,10};
int tabtab[2 * sizeof(tab) / sizeof(tab[0]) ];
memcpy(tabtab, tab, sizeof(tab)*sizeof(int) );
memcpy(&tabtab[10], tab, sizeof(tab)*sizeof(int) );
```

A la suite de l'exécution de ce fragment de code `tabtab` contient 20 entiers : $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

Initialiser une zone de mémoire

```
void *memset(void *debut, int valeur,
             size_t taille)
```

`memset` initialise `taille` octets à l'adresse `debut` avec `valeur` convertie en `unsigned char`.

```
char t[1001];
memset(&t, 'a', 1000);
t[1000]=0;
```

`t` initialisé comme une chaîne de 1000 caractères `'a'`

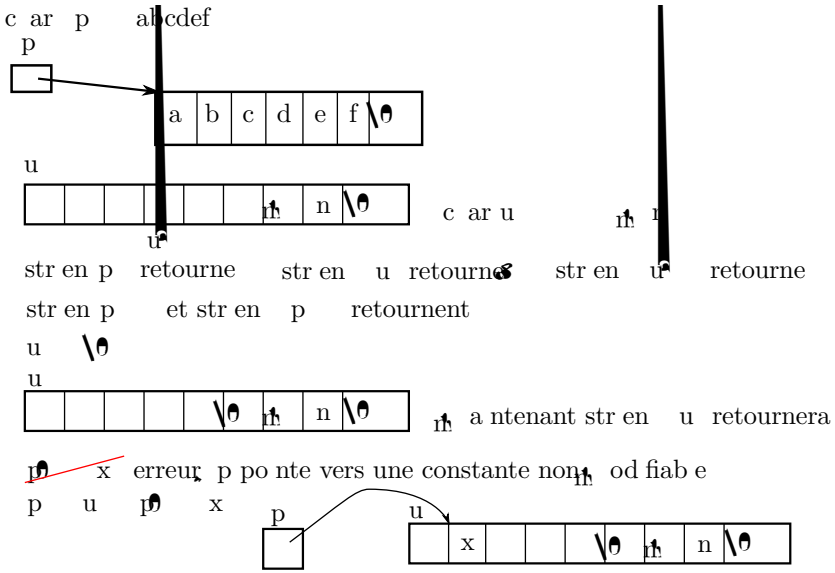
Chaînes de caractères en C

Chaîne de caractères en C : une suite d'octets qui termine avec le caractère `'\0'`
Le code ASCII du caractère `\0` est 00000000 en binaire (et il est différent du code ASCII du caractère `'0'`, voir le tableau de code ASCII que vous trouverez facilement sur internet).
Donc si

```
char c, d, e;
c = '\0';
d = 0;
e = '0';
```

alors `c` et `d` contiennent la même valeur par contre la valeur de `e` est différente.
On utilise le pointeur `char *` qui pointe sur le premier caractère d'une chaîne.
En C : `char`, `octet`, `byte` c'est la même chose.
et toujours

`sizeof(char) == 1`



Quelques fonctions utiles

```
size_t strlen(const char *s)
```

```
char *strcpy(char *dest, const char *source)
```

copie la chaîne à l'adresse `source` vers l'adresse `dest`. La zone `dest` doit être su samment grande.

```
int strcmp(const char *s1, char *s2)
```

compare les chaînes `s1` et `s2` et retourne

- -1 si `s1 < s2`
- 0 si `s1 == s2`
- 1 si `s1 > s2`

```
int strcat(char *dest, char *source)
```

concatène `source` à la suite de `dest`. `dest` doit avoir assez d'espace pour stocker le résultat.

Conversion de chaînes

```
double atof(const char *s)
int atoi(const char *s)
long atol(const char *s)
```

`atoi(''45'')` retourne 45.

concaténer deux chaînes de caractères (méthode 1)

```
char *a="abcdefghij";
char *b="klmnopqrs";
char *p, *q, *s, *r;
int i,j;
i=strlen(a); j=strlen(b);
r = malloc(i+j+1);
p = a; q = b; s = r;
while( *p ){
     *r = *p;
     r++;
     p++;
}
while( *q ){
  *r++ = *q++;
};
*r='\0';
printf("%s\n",s);
```

concaténer deux chaînes de caractères (méthode 2)

```
char *a="abcdefghij";
char *b="klmnopqrs";
char *p, *q, *s, *r;
int i,j;
i=strlen(a); j=strlen(b);
r = malloc(i+j+1);
p = a; q = b; s = r;
while( *p ){
    *r = *p;
    r++;
    p++;
}
while( *q ){
  *r++ = *q++;
};
*r='\0';
printf("%s\n",s);
```

concaténer deux chaînes de caractères (méthode 3)

```
char *a="abcdefghij";
char *b="klmnopqrs";
char  *r;
int i,j;
i=strlen(a); j=strlen(b);
r = malloc(i+j+1);
while( a[i] ){
    r[j++] = a[i++];
}
i = 0;
while( b[i] ){
  r[j++] = b[i++];
}
r[j]='\0';
printf("%s\n",r);
```

concaténer deux chaînes de caractères (méthode 3)

```
char *a="abcdefghij";
char *b="klmnopqrs";
char  *r;
int i,j;
i=strlen(a); j=strlen(b);
r = malloc(i+j+1);
strcpy(r,a);
strcat(r,b);
printf("%s\n",r);
```

concaténer deux chaînes de caractères (méthode 4)

```
char *a="abcdefghij";
char *b="klmnopqrs";
char  *r;
int i,j;
i=strlen(a); j=strlen(b);
r = malloc(i+j+1);
memcpy(r, a, i);
memcpy(r+i, b, j+1);
printf("%s\n",r);
```

Test et conversion de caractères

Les tests:

```
int isalpha(int c) /* lettre */
int isupper(int c) /* majuscule */
int islower(int c) /* minuscule */
int isdigit(int c) /* chiffre */
int isalnum(int c) /* chiffre ou lettre */
int isspace(int c) /* espace */
```

et les conversions:

```
int tolower(int c) /* vers minuscule */
int toupper(int c) /*vers majuscule*/
```