

1 Organisation mémoire

La **mémoire** sert à **stocker** des informations.

Les **informations** sont stockées dans des variables en **mémoire** à «un certain endroit» : à une **adresse**.

Si l'on voit la mémoire comme un «grand tableau», une **adresse** est l'équivalent d'un **indice** dans ce tableau.

Une adresse est donc une **valeur numérique** : un **pointeur**.

Pour manipuler des adresses, on a besoin de variables pouvant contenir des adresses : variables **de type pointeurs**.

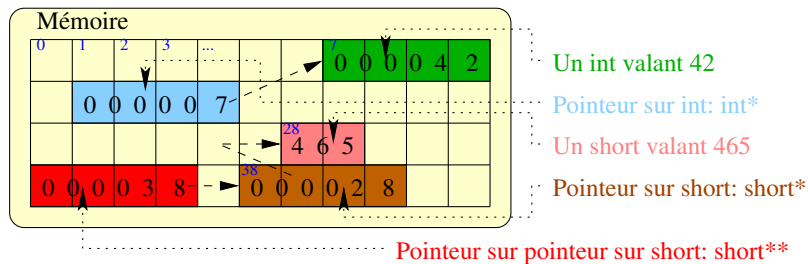
2 Pointeurs

Un pointeur désigne un «point» dans la mémoire : location **d'une donnée** ou d'une **zone contiguë de données**.

2.1 Déclaration de pointeur

On déclare un pointeur en donnant le **type de donnée se trouvant à l'adresse qu'il désigne(ra)**, ***** et son **nom**.

```
int *a ; // a est un pointeur vers un/des int.
short *b ; // b est un pointeur vers un/des short (int).
short **c ; // c est un pointeur vers un/des short*.
```



2.2 Extraction d'adresse et initialisation de pointeur

On obtient l'**adresse** d'une **variable** par l'opérateur **&**.
Permet d'**affecter** (ou d'**initialiser**) un pointeur.

```
int a ; // a est un int.
int *b ; // b est un pointeur vers un int.
b = &a ; // b contient l'adresse de a -> Pointeur vers a.
```

2.3 Lecture à une adresse

On accède à la valeur contenue à une adresse par l'opérateur *****.

```

int a ;           // a est un int.
int *b ;          // b est un pointeur vers un int.
b = &a ;           // b contient l'adresse de a -> Pointeur vers a.
*b = 5 ;           // On stocke 5 dans *b. -> Dans a.
a = 1 + *b ;       // On stocke dans a 1 + la valeur contenue a
                    // l'adresse b. -> a = a + 1.

```

Attention : avoir un pointeur **ne signifie pas** avoir le droit de manipuler la mémoire à l'adresse contenue par ce pointeur !

Attention : Le pointeur doit **désigner** une **zone mémoire** qui **vous a été attribuée**.

Le **type** de la donnée **lue à une adresse** est celui de ce sur quoi **pointe** le **pointeur** : les pointeurs sont **typés**.

2.4 Pointeurs et structures

```
struct foo_t { int ch ; };
```

Accès au champ d'une valeur de type **struct** : notation **pointée** :

```
struct foo_t my_foo ;
my_foo.ch = ...
```

Accès au champ d'une valeur de type **pointeur sur struct** : notation **fléchée** :

```
struct foo_t *my_pfoo ;
my_pfoo->ch = ...
```

Équivalence : **my_pfoo->ch = (*my_pfoo).ch**

3 Allocation de mémoire

On a parfois des besoins de mémoire **imprévisibles à la compilation** lors de l'**exécution**.

Une machine étant un objet « *fini* », la mémoire est disponible en quantité **limitée**.

On **demande** de la mémoire à l'**exécution** par **allocation dynamique**.

Fonction d'allocation **malloc** :

```
void* malloc (int size) ;
— Argument : nombre d'octets (contigus) demandés.
— Résultat : l'adresse début de la zone octroyée.
```

S'il a été **impossible d'allouer** de la mémoire, la **valeur** particulière de pointeur, **NULL**, est retournée.

L'opérateur **sizeof** permet d'obtenir le **nombre d'octets** nécessaires pour un **type** de donnée.

```

#include <stdlib.h> // Pour malloc, free.

int main ()
{
    float *t = malloc (5 * sizeof (float)) ; // Alloc "tableau" de 5 floats.
    if (t != NULL) { // Si allocation réussie...
        t[0] = 4.5 ; t[1] = 6.0 ; // Ecriture dans "le tableau".
    }
    return (0) ;
}

```

4 Libération de mémoire

Un bloc de mémoire qui **n'est plus utile** doit être **restitué** au système.

Fonction de libération **free** :

void malloc (**void** *ptr);

— Argument : adresse de **début** de la zone **allouée**.

— Résultat : néant.

La taille à libérer est connue du système.

```
#include <stdlib.h> // Pour malloc, free.

int main ()
{
    float *t = malloc (5 * sizeof (float)) ; // Alloc "tableau" de 5 floats.
    if (t != NULL) { // Si allocation reussie...
        t[0] = 4.5 ; t[1] = 6.0 ; // Ecriture dans "le tableau".
        free (t) ; // Liberation de la memoire.
    }
    return (0) ;
}
```