Langage C
Pointeurs

1 Pointeurs et adresses

La mémoire vive de la machine peut être vue comme une suite d'octets[1], chaque octet possédant une adresse unique. Il est commode et habituel d'écrire les adresses sous formes de nombres hexadécimaux positifs - c'est par exemple de cette manière que les adresses sont affichées par le déboguer `gdb`. Rappelons que les entiers s'écrivent en base hexadécimale (base 16) avec les 16 chiffres

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`

avec la correspondance suivante :

décimal	hexadécimal	binaire
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	a	1010
11	b	1011
12	c	1100
13	d	1101
14	e	1110
15	f	1111

A noter que les adresses affichées sur la sortie standard par `printf`, ou encore affichées par un déboguer, sont des adresses virtuelles. L'adresse réelle en mémoire d'un objet n'a pas d'intérêt pour nous : elle peut changer au cours du temps, la correspondance entre adresses réelles et virtuelles étant gérée automatiquement par un composant de la machine appelé MMU (Memory Management Unit).

1. C'est une vision très approximative, mais suffisante pour le moment.

1.1 L'opérateur &

L'opérateur `&`, ou *opérateur de prise d'adresse*, permet comme son nom l'indique de récupérer l'adresse d'une variable où d'un élément d'un vecteur.

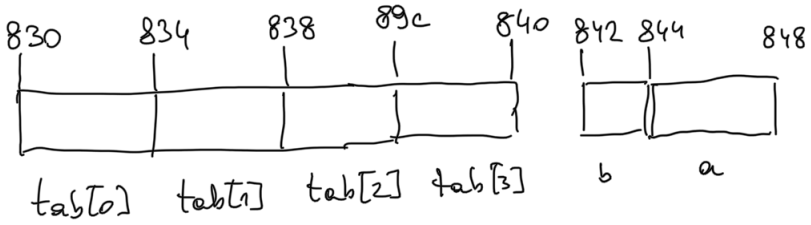
Exemple. Considérons un programme, exécuté sous `gdb`, atteignant les lignes suivantes :

```
1 int a;
2 short b;
3 int tab[] = {1,-12,66,7};
```

Rappelons qu'une commande de la forme `p` *expr* ("p", pour "print") permet d'afficher sous `gdb` la valeur d'une expression. On peut par exemple afficher les adresses de `a` et `b`, ou encore les adresses d'eléments de `tab`. Sur ma machine, cela donnera, pour une exécution donnée :

```
(lldb) p &a
(int *) $0 = 0x00007ffeefbff844
(lldb) p &b
(short *) $1 = 0x00007ffeefbff842
(lldb) p &tab[0]
(int *) $2 = 0x00007ffeefbff830
(lldb) p &tab[1]
(int *) $3 = 0x00007ffeefbff834
(lldb) p &tab[4]
(int *) $4 = 0x00007ffeefbff840
```

On constate que les adresses sont affichées avec 16 chiffres hexadécimaux. Sachant qu'un octet s'écrit avec 2 chiffres en base hexadécimale, chaque valeur d'adresse s'écrit donc sur ma machine avec 8 octets [2].
La variable `b` se trouve à l'adresse `0x00007ffeefbff842`. La variable `a` est à l'adresse `0x00007ffeefbff844`. On observe égalemement que : l'adresse de l'élément `tab[0]` du vecteur `tab` est `0x00007ffeefbff830` ; l'adresse de `tab[1]` est `0x00007ffeefbff834` c'est-à-dire 4 octets plus loin en la mémoire. Autrement dit, sur ma machine, une donnée de type `int` occupe 4 octets.
A noter que l'élément `tab[4]` n'existe pas dans `tab` (son dernier élément est `tab[3]`), mais l'expression `&tab[4]` est tout-à-fait est correcte : elle vaut l'adresse du premier octet immédiatement après le dernier élément de `tab` [3].
Le diagramme suivant illustre les emplacements de `a`, `b`, et des éléments `tab` en mémoire, pour cette exécution (seuls les trois derniers chiffres des adresses sont indiqués) :



2. Donc ma machine est dotée d'un processeur $8 \times 8 == 64$ bits.
3. Nous reviendrons sur ce point à la section 5.1.

Notez qu'il y a un « trou » de deux octets entre le dernier élément de `tab` et la variable `b` – les octets ne sont donc pas tous utilisés. Nous pouvons aussi voir aussi que l'adresse de `a` est 2 octets plus loin que celle de `b`, qui est de type `short` : sur ma machine, un `short` occupe donc 2 octets en mémoire.

1.2 Variables de type pointeur

En C les adresses sont stockées dans des variables de type pointeur (ou plus simplement : des pointeurs). Une variable de type *pointeur vers `int`* permettra de stocker l'adresse de toute variable de type `int`, une variable de type pointeur vers `short` permettra de stocker l'adresse de toute variable de type `short`, etc.

Déclaration de pointeurs. Sans surprise, le fragment de code suivant déclare trois variables de type `int`, `short` et `double` :

```
int i;
short s;
double d;
```

Le fragment suivant, en revanche, déclare trois variables respectivement de types *pointeur vers* `int`, *pointeur vers* `int short` et *pointeur vers* `int double` :

```
int *pi;
short *ps;
double *pd;
```

Plus généralement, une variable `q` permettant de stocker les adresses de données de type `data` aura une déclaration de la forme :

```
data *q;
```

On peut par exemple écrire :

```
int **ppi;
```

où `pi` est alors de type *pointeur vers pointeur vers `int`* (un pointeur de pointeur, ou *handle*). On peut déclarer plusieurs variables d'un type pointeur commun, en écrivant :

```
int *p, *q, *r;
```

Notez la répétition de l'étoile `*` avant chaque nom de variable. Si l'on écrit

```
int *p, q, r;
```

la variable `p` sera de type pointeur vers `int`, mais `q` et `r` seront de type `int` et non de type pointeur vers `int`.

Affectation de pointeurs. Considérons le fragment de code suivant :

```
int a = 5;
int *p;
p = &a;
```

Rappelons que l'opérateur `&` permet de récupérer l'adresse d'une variable (ou plus généralement, d'une donnée) en mémoire.

Il est important de comprendre que les valeurs renvoyées par l'opérateur `&` sont *typées* : à gauche d'une variable de type `int`, l'opérateur renverra l'adresse de cette variable sous la forme d'une *valeur d'adresse de type pointeur vers `int`* ; à gauche d'une variable de type `short`, cette valeur d'adresse sera de type pointeur vers `short`, etc.

Reprenons l'exemple précédent :

```
1 int a = 5;
2 int *p;
3 p = &a;
```

A l'exécution :
- la variable `a` est de type `int`,
- la variable `p` est de type pointeur vers `int`,
- appliqué à `a`, l'opérateur `&` renvoie l'adresse de cette variable, sous la forme d'une valeur d'adresse de type pointeur vers `int`.
- la variable `p` reçoit cette valeur d'adresse.

L'affectation est donc correcte en termes de typage.

Après cette affectation, on dit que `p` *pointe vers* `a`.

Remarque. On ne pourrait pas écrire par exemple :

```
1 int a = 5;
2 int *p;
3 p = a;
```

sans un message d'avertissement du compilateur :

```
warning: assignment makes pointer from integer without a cast
```

Le message est clair, et conforme aux règles de typage ci-dessus : l'expression `a` est de type `int` (`integer`), le type de `p` impose à l'expression à droite de l'affectation d'être de type pointeur vers `int` (`pointer`), mais il n'y a pas de conversion explicite (`cast`) transformant la valeur de `a` en une expression du bon type [4].

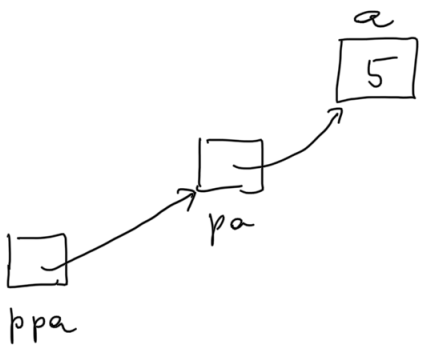
Initialisation immédiate de pointeurs Comme toute variable, une variable de type pointeur peut recevoir une valeur dès sa déclaration. Dans l'exemple ci-dessous, la variable `pa` est un pointeur vers un `int`, immédiatement initialisé avec l'adresse de la variable `a`. La variable `ppa` est un pointeur vers une donnée de type `int *` (*i.e.* de type pointeur vers pointeur vers `int`) immédiatement initialisé avec l'adresse de la variable `pa` :

```
1 int a = 5;
2 int *pa = &a;
3 int **ppa = &pa;
```

Noter là-encore la cohérence du typage : la variable `pa` est de type pointeur vers `int` ; l'expression `&pa` est de type pointeur vers pointeur vers `int`, qui est le type de `ppa`.

4. Ceci de toute manière aurait de grandes chance d'entraîner tôt ou tard le plantage du programme pour des raisons qui seront claires plus loin

En règle générale, les valeurs numériques exactes de `&a` ou de `&pa` ont peu d'intérêt. Ce qui est important en revanche, c'est le fait que `pa` contienne l'adresse en mémoire de `a` et que `paa` contienne l'adresse de `pa`, autrement dit que `pa` pointe vers `a` et que `ppa` pointe vers `pa`, ce que nous pouvons représenter de la façon suivante :



2 L'opérateur *

L'opérateur étoile `*`, ou *opérateur de déréférencement*, se place toujours devant un pointeur. Il peut-être utilisé aussi bien à gauche qu'à droite d'un opérateur d'affectation `=`, avec deux sens différents.

Remarque. L'opérateur de déréférencement ne doit pas être confondu avec le symbole étoile `*` indiquant, dans une déclaration de variable, qu'il s'agit d'un pointeur. Dans

```
int *p;
```

l'étoile *n'est pas* cet opérateur, mais seulement une indication du type de `p` : la variable `p` est un pointeur vers `int`, et non un `int`. De même,

```
int a = 5;
int *p;
p = &a;
```

peut s'écrire :

```
int a = 5;
int *p = &a;
```

Là-encore, le symbole étoile n'est pas l'opérateur de déréférencement, mais seulement une indication du type de `p`.

2.1 L'opérateur * dans une expression à droite d'une affectation

Prenons l'exemple suivant :

```
int a = 42, b = 5, c;
int *p, *q;

p = &a;         // p pointe vers a
q = &b;         // q pointe vers b
c = *p + *q;    // acces aux donnees stockees en &a et &b.
```

Comme précisé ci-dessus, les symboles étoiles dans les déclarations de `p` et `q` ne sont pas l'opérateur de déréférencement. En revanche, dans la dernière affectation, les étoiles qui précèdent `p` et `q` sont bien cet opérateur dans les deux cas. L'effet de cet opérateur est décrit par la règle suivante :

> *Règle de déréférencement à droite.* Lorsqu'une variable `p` de type pointeur vers `data` contient l'adresse d'une donnée de type `data`, l'expression `*p` vaut cette donnée, et est de type `data`.

En particulier, lorsque un pointeur `p` pointe vers une variable, l'expression `*p` vaut la valeur courante de cette variable. Reprenons l'exemple. Après les deux premières affectations, `p` contient l'adresse de `a` et `q` l'adresse de `b`. Dans la troisième,

- l'expression `*p` vaut la donnée stockée à l'adresse stockée dans `p`,
- *i.e.* `*p` vaut la donnée stockée en `&a`,
- *i.e.* `*p` vaut la valeur de `a`,
- *i.e.* `*p` vaut `42`.

De même, l'expression `*q` vaut la valeur de `b` c'est-à-dire `5`, et la valeur prise par `c` est 47. Autre exemple :

```
int tab[4] = {1,2,3,4};
int *p;
int *q;
int a;
p = &tab[1];
q = &tab[3];
a = (*p - *q) + (*p);
```

Ici, `p` et `q` recoivent les valeurs d'adresses de deux éléments d'un vecteur. Pour les même raisons, la dernière affectation est opératoirement équivalente à

```
a = (tab[1] - tab[3]) + (tab[1]);
```

Autre exemple, dans lequel l'étoile encadrée par `(*p - *q)` et `(*p)` est la multitplication ordinaire, et ne doit pas être visuellement confondue avec un déréférencement :

```
int tab[4] = {1,2,3,4};
int *p;
int *q;
int a;
p = &tab[1];
q = &tab[3];
a = (*p - *q) * (*p);
```

Enfin, le même exemple sous une forme plus condensée via des initialisations immédiates :

```
int tab[4] = {1,2,3,4};
int *p = &tab[1];
int *q = &tab[3];
int a = (*p - *q) * (*p);
```

2.2 L'opérateur * à gauche d'une affectation

Prenons l'exemple suivant :

```
int a;
int *p ;
p = &a;
*p = 42;     // a vaut a present 42
```

Dans la norme de langage C on appelle *l-value* toute expression qui peut apparaître à gauche d'un symbole d'affectation.

> *Règle de déréférencement à gauche.* Si une variable `p` est de type pointeur vers `data`, alors `*p` est une l-value. Pour toute expression `expr` de type `data`,
>
> ```
> *p = expr;
> ```
>
> stocke la valeur de `expr` à l'adresse stockée dans `p`.

En particulier, si `p` pointe vers une variable, l'affectation portera sur cette variable – sans modifier, bien sûr, la valeur de `p`. Dans la dernière instruction,

- la valeur `42` est stockée à l'adresse stockée dans `p`,
- *i.e.* `42` est stockée en `&a`,
- *i.e.* la valeur `42` est stockée dans `a`.

Autre exemple :

```
int a;
int *p ;
p = &a;
*p = 42;         // a vaut a present 42
*p = *p + 1;     // a vaut a present 43
```

La première affectation fait pointer `p` sur `a`. La seconde écrit `42` dans `a` par l'intermédiaire de son adresse stockée dans `p`. La troisième relit la valeur de `a` via l'expression `*p`, ajoute `1` à cette valeur, et stocke la valeur résultante dans `a` par l'intermédiaire de `p`. Autre exemple :

```
int tab[4]={1,2,3,4};
int *p = &tab[1];
int *q = &tab[3];
*p = *p + 2 * (*q) + 1;
```

pour les même raisons, l'expression à droite de la dernière affectation vaut

```
tab[1] + 2 * (tab[3]) + 1;
```

En outre, `p` contient l'adresse de `tab[1]` donc l'affectation a le même effet que :

```
tab[1] = tab[1] + 2 * (tab[3]) + 1;
```

3 La valeur NULL

Un pointeur peut prendre une valeur particulière, la valeur de *pointeur nul*. Cette valeur est celle de la constante symbolique `NULL` définie dans `stddef.h`.
Le langage C garantit que `NULL` n'est jamais une adresse valide en lecture ou en écriture [5] : toute tentative pour déréférencer `NULL` provoquera une exception et la terminaison immédiate du programme. Si par exemple `p` vaut `NULL` les instructions `*p = 42;` ou `a = *p;` provoqueront une terminaison du programme dans les deux cas.

Déréférencement d'un pointeur non-initialisé. Supposons que l'on commette l'erreur d'écrire :

```
1 int *p;
2 *p = 5;
```

ou `p` est non-initialisé, donc de valeur indéfinie. Deux cas peuvent se produire : ou bien cette valeur d'adresse n'est pas valide, ce qui provoquera la terminaison du programme ; ou bien, par pur hasard, il s'agit d'une adresse valide.
Dans le second cas, l'affectation modifiera la mémoire à cette adresse, en y écrivant la valeur `5`. Il est possible que ce code erroné s'exécute sans problèmes, et que l'on découvre seulement bien plus tard que la mémoire a été alterée de façon imprévue [6]. Il est souvent extrêmement difficile dans ce type de situations de comprendre d'où vient une telle erreur.

Pour éviter ce problème il est très fortement recommandé de suivre la consigne suivante :

Consigne. Ne déclarez **jamais** un pointeur sans l'initialiser. S'il n'y a pas d'autre valeur initiale appropriée pour un pointeur, initialisez-le avec `NULL`.

4 Les pointeurs comme arguments de fonctions

Comme vous le savez [7], les paramètres d'une fonction sont tout simplement des variables locales à cette fonction, initialisées avec les valeurs des expressions utilisées lors d'un appel. Puisque ces paramètres sont locaux, toute modification de leurs valeurs dans le corps la fonction n'affectent qu'eux-mêmes : ils sont pas « visibles » à l'extérieur de celle-ci.

Cependant, à l'aide des pointeurs et de l'opérateur `*`, une fonction *peut* modifier de manière indirecte les valeurs de variables ou d'éléments de vecteurs déclarés à l'extérieur de celle-ci. Il suffit de lui donner en arguments *l'adresse* de ces données : elle pourra alors les modifier par déréférencement.

5. La norme spécifie également que `NULL` est effectivement de valeur numérique nulle, c'est-à-dire vaut `0`. Les pointeurs et les entiers ne sont pas interchangeables, et `0` est l'unique exception. Mais le fait d'écrire `NULL` souligne dans le code le fait qu'il s'agit d'une valeur singulière de pointeur, pas d'un entier : il vaut mieux par exemple écrire `if (p != NULL)` plutôt que `if (p != 0)` ou `if (p)`.

6. Ou pire encore, que l'on ne découvre pas cette erreur, en obtenant des résultats erronés que l'on croira corrects

7. Je l'espère.

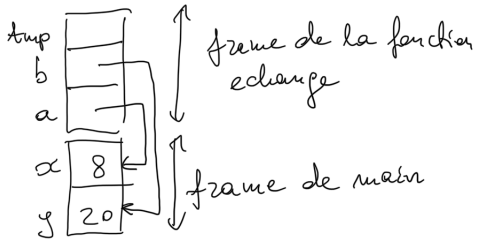
4.1 Modifier des données externes via leur adresse

```
1  void echange( int *a, int *b ){
2    int tmp = *a;
3    *a = *b;
4    *b = tmp;
5    return;
6  }
7  int main(){
8    int x = 8, y = 20;
9    echange( &x, &y );
10   //...
11 }
```

Lorsque l'on imbrique des appels de fonctions (`main` qui appelle `f`, qui appelle `g`, qui appelle `h`, etc.), les informations relatives à chaque appel s'accumulent dans une zone de la mémoire appelée la *pile d'appels* (ou simplement : la pile). A chaque appel en cours d'exécution est associée un élément de la pile appelé le *frame* de cet appel. Chaque frame contient :

- les variables stockant les valeurs données aux paramètres de la fonction pour cet appel [8],
- les variables correspondant aux variables locales déclarées dans la fonction [9],
- l'adresse de retour de l'appel (l'adresse de l'instruction où le contrôle reviendra au retour d'appel.

Dans notre exemple, après le lancement du programme, arrivé à la ligne 9 et immédiatement avant l'appel de la fonction `echange`, la pile contient l'unique frame correspondant à l'appel de la fonction `main()`, deux variables `x` et`y` contenant les valeurs `8` et `20`.
L'instruction `echange(&x, &y)` ajoute à la pile un nouveau frame pour l'éxecution de cet appel d'`echange`. Ce frame contient : deux variables `a` et `b`, respectivement initialisées avec les adresses de `x` et de `y` ; une variable `tmp` (non encore initialisée).
Le dessin suivant représente la pile au tout début de l'exécution de la fonction `echange` :



La fonction effectue l'échange par déréférencement des deux données `int` dont les adresses sont stockées dans `a` et `b` – `a` pointe vers `x`, et `b` pointe vers `y` : au retour d'appel, on retrouve `x` et `y` dans `main` avec leur valeurs échangées.

8. Il peut y avoir plusieurs exemplaires d'une même fonction en cours d'appel : lorsqu'elle est récursive. Dans ce cas, la pile contient un frame par appel en cours d'exécution, chacun disposant de ses propres exemplaires de variables distincts de tous les autres, même s'ils portent les mêmes noms dans le code source.

9. Mais pas les variables `static` déclarées dans la fonction. Les variables `static` seront traitées ailleurs dans ce cours.

Le principe illustré par cet exemple s'exprime ainsi :

> **Pour qu'une fonction puisse modifier les valeurs de variables extérieures à celle-ci, il suffit de lui passer comme valeurs de paramètres les adresses de ces variables.**

Cette règle ne concerne pas les vecteurs – en tout cas, pas exactement de cette manière. Les vecteurs comme paramètres de fonctions seront traités à la section 7.

4.2 Les paramètres de type pointeurs pour récupérer les valeurs calculées par une fonction

Une fonction ne peut renvoyer par un `return` qu'une unique valeur. Mais les paramètres de type pointeur permettent de contourner cette limitation : il est possible – et même courant – de définir une fonction calculant plusieurs valeurs, toutes récupérables par l'appelant.

```
1  void minmax(int n, int tab[], int *mn, int *mx){
2    int a = 0, b = 0;
3    for(int i = 1; i < n; i++){
4      if(tab[i] < a)
5      a = i;
6      else if(tab[i] > b)
7      b = i;
8    }
9    if(mn != NULL)
10   *mn = a;
11   if(mx != NULL)
12   *mx = b;
13   return;
14 }
```

La fonction `minmax` calcule les indices du plus petit et du plus grand élément de `tab` (respectivement `a` et `b` en fin de boucle). Ces indices ne sont pas renvoyés par le `return` final (la fonction est de toute manière en `void`) : ils sont stockés aux adresses `mn` et `mx` – et, pour chaque adresse, uniquement si elle ne vaut pas `NULL`. La fonction `minmax` peut s'utiliser de la façon suivante :

```
1 int tab[10];
2 /* remplissage de tab[] */
3 int s, t;
4 minmax(10,tab, &s, &t);
```

Après l'appel de `minmax` les variables `s` et `t` contiendront les indices de plus petit et de plus grand éléments de `tab`.
Noter que le test de nullité des pointeurs `mn` et `mx` n'est pas ici un principe de précaution, mais d'une utilité pratique : l'appelant peut librement choisir d'extraire une seule information plutôt que les deux, il lui suffit de fournir à la fonction un pointeur `NULL` pour celle qu'il souhaite ignorer.

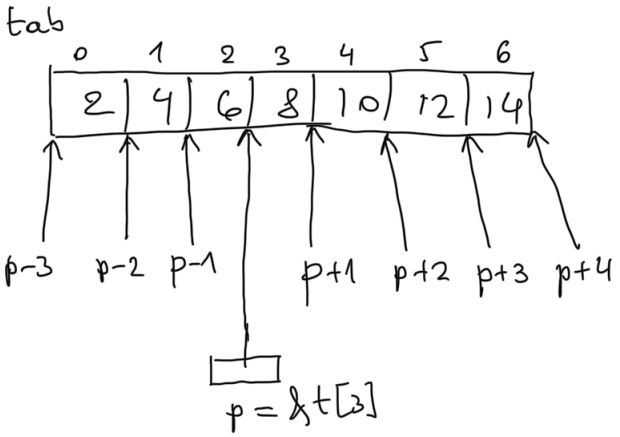
5 Arithmétique des pointeurs

Lorsque `p` est un pointeur et `n` une expression entière, `p + n` et `p - n` sont valeurs d'adresses de même type que `p`. On les appelle des *expressions de type pointeur*. Nous allons préciser ici le sens de ces deux formes, dont les valeurs dépendent de la valeur de `p`, de la valeur entière de `n` et du type de `p`.

5.1 Pointeur + entier, pointeur - entier

```
1 int tab[] = {2, 4, 6, 8, 10, 12, 14};
2 int *p = &tab[3];
```

Le dessin suivant montre les valeurs de `p - 1`, `p - 2`, `p - 3` et, dans l'autre direction, les valeurs de `p + 1`, `p + 2`, `p + 3`, `p + 4`.



La règle générale est la suivante :

> Si `p` est un pointeur vers `data` et `n` est entier, l'expression `p + n` est une valeur d'adresse de type pointeur vers `data`, obtenue en se décalant en mémoire, à partir de `p`, de `n` objets de type `data`, soit dans la direction des adresses supérieures si `n > 0`, soit dans la direction des adresses inférieures si `n < 0` [10].

Dans notre exemple `p` est un pointeur vers `int`, de valeur `&tab[3]`. On a donc :

- `p + 1` est l'adresse de l'entier qui suit celui pointé par `p`, *i.e* `&tab[4]`,

10. Autre formulation plus technique de cette règle, exprimée en termes de calculs d'adresses : si `p` est de type pointeur vers `data` et `n` une expression entière, alors `pi + n` et `pi - n` sont des valeurs d'adresse de même type que `p`, et,
   - la valeur de `pi + n` se calcule en ajoutant `(n * sizeof(data))` à la valeur numérique de `p`,
   - la valeur de `pi - n` se calcule en soustrayant `(n * sizeof(data))` à la valeur numérique de `p`.

- `p + 2` est l'adresse de l'entier qui suit celui d'adresse `p + 1`, *i.e* `&tab[5]`, etc.
- `p - 1` est l'adresse de l'entier qui précède celui pointé par `p`, *i.e* `&tab[2]`,
- `p - 2` est l'adresse de l'entier qui précède celui d'adresse `p - 1`, *i.e* `&tab[1]`, etc.

Remarque sur l'adresse de l'extrémité droite d'un vecteur. Sur le diagramme, `p + 4` est visuellement l'adresse d'un élément fictif `tab[7]`. En fait, on a bien `p + 4 == &tab[7]` : les deux expressions sont correctes, et égales. Toutes deux valent la valeur de l'adresse située en mémoire immédiatement après le dernier élément de `tab`. La norme du langage spécifie que cette adresse est toujours valide, elle peut être utilisée dans le programme, stockée dans un pointeur, comparée à une autre, etc.
Il ne faut par contre jamais déréférencer un pointeur contenant une adresse calculée de cette manière : d'une part, la valeur qui y est stockée est indéfinie ; d'autre part, toute tentative pour y stocker une donnée risque d'altérer la mémoire et de mener le programme vers un état imprévisible.

5.2 Déréférencement d'une expression de type pointeur.

L'opérateur de déréférencement `*` introduit à la section 2 est applicable à une expression de type pointeur. L'effet de cet opérateur est alors exactement le même que s'il était appliqué à un pointeur dont la valeur d'adresse serait celle de l'expression.

Déréférencement à droite de =

```
int tab[] = {2, 4, 6, 8, 10, 12, 14};
int *p = &tab[3];
int l = *(p - 1) + *(p + 2);
```

Comme indiqué ci-dessus, on a `p - 1 == &tab[2]` et `p + 2 == &tab[5]`. L'effet de

```
int l = *(p - 1) + *(p + 2);
```

est donc exactement le même que s'il on écrivait, avec deux pointeurs de plus :

```
int *q2, *q5;
q2 = p - 1;          // q2 pointe vers tab[2]
q5 = p + 2;          // q5 pointe vers tab[5]
int l = *q2 + *q5;   // l recoit la valeur de tab[2] + tab [5]
```

ou encore, en écrivant de manière directe :

```
int l = tab[2] + tab[5];
```

Déréférencement à gauche de =. Comme pour les pointeurs, le déréférencement d'une expression de type pointeur produit une l-value :

```
int tab[] = {2, 4, 6, 8, 10, 12, 14};
int *p = &tab[3];
*(p - 2) = 42;
```

où `p + 2 == &tab[5]`, et où l'effet de l'affectation est le même que s'il on écrivait :

```
int *q2;
q2 = p - 1;          // q2 pointe vers tab[2]
*q2 = 42             // tab[2] recoit la valeur 42.
```

Autre exemple, en combinant des déréférencements à gauche et à droite :

```
int tab[] = {2, 4, 6, 8, 10, 12, 14};
int *p = &tab[3];
*(p - 2) =  *p  - *(p + 2);
```

où la dernière instruction a le même effet que :

```
tab[1] = tab[3] - tab[5];
```

5.3 Notation indexée avec les pointeurs.

La notation indexée des vecteurs est utilisable avec les pointeurs (nous allons voir dans un instant qu'en réalité, ces deux notations ont le même sens qu'il s'agisse d'un vecteur ou d'un pointeur) : si `p` est un pointeur et `n` un entier ou une expression entière, alors :

`*(p + n)` peut aussi s'écrire `p[n]`

`*(p - n)` peut aussi s'écrire `p[-n]`

au sens : partir de la valeur d'adresse de `p` ; calculer la valeur d'adresse `p + n` ou `p - n` ; accéder par déréférencement à l'adresse calculée (en lecture si l'on est dans une expression, en écriture si l'on est à gauche d'une affectation). On peut par exemple écrire, ou bien :

```
int tab[] = {2, 4, 6, 8, 10, 12, 14};
int *p = &tab[3];
int l = *(p - 1) + *(p + 2);
*(p - 2) =  *p  - *(p + 2);
```

ou bien, de manière équivalente – et sachant que `*p` peut aussi s'écrire `p[0]` :

```
int tab[] = {2, 4, 6, 8, 10, 12, 14};
int *p = &tab[3];
int l = p[-1] + p[2];
p[-2] =  p[0] - p[2];
```

Remarque : sur le sens réel de la notation indexée pour les vecteurs. La possiblité de se servir de la notation indexée pour les vecteurs comme pour les pointeurs est due au phénomène suivant :

- utilisé dans une expression, le nom d'un vecteur `tab` d'éléments de type `data` *est immédiatement converti* en une expression de type pointeur vers `data`, de valeur égale à l'adresse de départ du vecteur,
- le sens réel de la notation `tab[n]` est en fait `*(tab + n)`, où dans la seconde, le nom `tab` est automatiquement converti en une valeur de pointeur – les deux notations sont utilisables sur un vecteur.

Les notations `p[n]` ou `*(p + n)` d'une part, `tab[n]` ou `*(tab + n)` d'autre part, ont en fait le même sens : dans le premier cas, `p` est déjà un pointeur ; dans le second, `tab` devient un pointeur après conversion automatique du nom de vecteur `tab`. Pour les mêmes raisons,

- `tab` vaut l'adresse de départ du vecteur après conversion du nom `tab` en une valeur de type pointeur : `tab == &tab[0]`,
- `tab + i` vaut l'adresse de l'élément de positon `i` du vecteur `tab`, calculée après conversion du nom `tab`, on a donc : `tab + i == &tab[i]`,
- `*(tab + i)` est une l-value après conversion – comme toute l-value, on peut lui appliquer l'opérateur de prise d'adresse : `&(*(tab + i)) == &tab[i] == tab + i`.

Remarque : sur l'impossiblité de réaffecter un nom de vecteur. Il y a des exceptions à la règle de conversion automatique des noms de vecteurs. Si `tab` a été déclaré par `int tab[7]`, alors `tab` est bien un *vecteur* et non un pointeur (même si *le nom* `tab` est en général converti en pointeur), et le type de `tab` est bien "vecteur d'entiers à `7` éléments" (ce type se note `int[7]`). Deux des cas où un nom de vecteur *n'est pas* converti en une valeur de pointeur sont :

- lorsque ce nom est soumis à l'opérateur `sizeof`,
- lorsqu'il se trouve à gauche d'une affectation.

Dans les deux cas, le nom `tab` conserve son type réel. Dans le premier, `sizeof(tab)` vaut bien `7 * sizeof(int)`, et non la taille en mémoire d'un pointeur. Le second explique, de manière exotique, pourquoi il est impossible de réaffecter un nom de vecteur. Si l'on tente de compiler :

```
int t[7], u[7] = {2, 4, 6, 8, 10, 12, 14};
t = u;
```

la compilation échouera par simple erreur de typage : le nom `u` sera converti à droite de l'affectation en une valeur de type pointeur vers `int` ; le nom `t` conservera à gauche son type réel, et ces deux types sont incompatibles. Plus généralement, il est impossible de placer à droite une expression du bon type, à cause des conversions automatiques.

Curiosité. Si `p` est un pointeur (ou un vecteur dont le nom a été converti) et `n` un entier, alors le compilateur transformera l'expression `p[n]` en `*(p + n)`. Ce qui est bien moins évident c'est que `n[p]` est aussi traduit en `*(p + n)` : les notations `p[n]` et `n[p]` sont équivalents. Bien sûr, la deuxième notation est à bannir de vos programmes, à moins que vous ne desiriez participer à l'International Obfuscated C Code Contest : `https://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest`

5.4 pointeur++, pointeur−

```
int t[]= {1,2,3,4,5,6};
int *px = &t[2];
int *py = &t[5];
px = px + 1;
py = py - 1;
```

Par analogie avec les variables entières, les deux dernières instructions sont respectivement appelées *incrémentation* et *décrémentation* de pointeur – mais ces termes sont à prendre, bien sûr, au sens de l'arithmétique des pointeurs. Après ces instructions, `px` contient l'adresse de l'élément qui suit `t[2]` en mémoire c'est-à-dire `t[3]`, et `py` contient l'adresse

de l'élément qui précède `t[5]` en mémoire c'est-à-dire `t[4]`. Autrement dit, lorsqu'un pointeur pointe vers un élément d'un vecteur :

- incrémenter ce pointeur de `1` revient à passer à l'élément suivant,
- décrémenter ce pointeur de `1` revient à passer à l'élément précédent.

Il se trouve que l'on peut utiliser pour tout pointeur `p`, les notations `p++`, `++p`, `p--`, `--p`, avec le sens que l'on imagine – par exemple, la valeur de `p++` est la valeur courante de `p` (avant son incrémentation), et son effet de bord consiste à incrémenter `p` (toujours, bien sûr, au sens de l'arithmétique des pointeurs).

Exemple. Voici une manière de calculer la somme des éléments d'un vecteur en se servant d'une boucle qui incrémente un pointeur plutot qu'une valeur d'indice :

```
1 double tab[] = {1.13, -2.23, 3.54 -4.65, 55.78};
2 double *f = sizeof(t)/sizeof(t[0]) + tab;   //f vaut l'adresse qui suit tab[4]
3 double s = 0;
4 for(double *p = tab; p < f; p++){
5   s += *p;
6 }
```

Noter la comparaison `(p < f)` entre deux pointeurs. Cette comparaison est vraie (c'est-à-dire s'évalue en `1`) si l'adresse stockée dans `p` est (numériquement) inférieure à l'adresse stockée dans `f`, c'est-à-dire si la valeur d'adresse de `p` précède (strictement) celle de `f` en mémoire.

Exemple. Cet exemple illustre le rôle essentiel joué par le typage dans l'arithmétique des pointeurs (*c.f.* également la remarque à la fin de la section 6).

```
1 int t[] = {1, 3, 5};
2 int *pi = &t[0];
3 char *pc = (char *) pi;
4 pi++;
5 pc++;
```

Après l'instruction de la ligne 3, les pointeurs `pi` et `pc` contiennent des valeurs d'adresses *numériquement* égales : ils pointent tous les deux vers le premier octet de `t[0]`. Cependant, ces deux pointeurs n'ont pas les mêmes *types*. L'incrémentation de `pi` décale `pi` en mémoire d'une fois la taille d'un `int`, celle de `pc` d'une fois la taille d'un `char` : après ces incrémentations, `pi` pointe vers `t[1]`, tandis que `pc` pointe vers le deuxième octet de `t[0]`.

5.5 Différence de pointeurs

```
1 double t[]={0.1, -1.5, 2.2, -3.3, 4.4, -5.5, 6.6, -7.7, 8.8, -9.9, -11.1 };
2 double *pa = &t[2];       // pa pointe vers t[2]
3 double *pb = &t[8];       // pb pointe vers t[8]
4 ptrdiff_t db = pb - pa;    // db prend la valeur 6
5 ptrdiff_t da = pa - pb;    // da prend la valeur -6
```

Le type `ptrdiff_t` défini dans `<stddef.h>` est le type des entiers signés permettant d'exprimer la différence de deux pointeurs de même type[11]. La règle mise en jeu dans cet exemple est la suivante :

> Si `p` et `q` sont deux pointeurs vers `data`, l'expression `(p - q)` vaut : le nombre d'objets de type `data` intercalables entre les valeurs d'adresses de `p` et `q` si `(p >= q)` ; l'opposé de ce nombre si `(p < q)` [12].

Dans notre exemple, `pa` pointe vers `tab[2]` et `pb` vers `t[8]`. Il y a `(8 - 2) == 6` objets de type `double` intercalés entre les deux adresses (`t[2]`, `t[3]`, `t[4]`, `t[5]`, `t[6]`, `t[7]`). De plus `(pb > pa)`, donc `(pb - pa) == 6`. Symétriquement `(pa < pb)`, donc `(pa - pb) == -6`

A noter que la norme du langage garantit que la différence de deux pointeurs est bien définie dans seulement deux cas : lorsque ces deux pointeurs pointent vers deux éléments d'un même vecteur ; lorsqu'ils pointent vers deux éléments d'une même zone mémoire allouée par `malloc`.

6 Les pointeurs génériques

Un pointeur générique un pointeur vers `void`, c'est-à-dire de type `void *`. Le type `void` (vide) est un type ne contenant aucune valeur – on ne peut pas construire une valeur de type `void`, mais il est possible de déclarer des *pointeurs vers* `void` :

```
1 void *pg;
```

Un pointeur générique peut recevoir la valeur d'adresse de toute donnée, quel que soit son type. Inversement, tout pointeur peut recevoir la valeur d'un pointeur vers `void`. Dans les deux cas, il n'est même pas nécessaire d'écrire des conversions (cast) :

```
1 int tab[] = {1, 2, 3, 4};
2 int *p, *q;
3 void *r, *s;
4
5 p = &t[1];              // int *  vers int *
6 r = p;                  // int *  vers void *
7 q = r;                  // void * vers int *
8 s = &t[2];              // int *  vers void *
```

L'écriture de conversions explicites reste possible, et nous pourrions écrire les quatre dernières instructions comme :

```
1 p = &t[1];              // int *  vers int *
2 r = (void *) p;         // int *  vers void *
3 q = (int *)  r;         // void * vers int *
4 s = (void *) &t[2];     // int *  vers void *
```

mais cela n'apporte rien, et est inutile.

11. Le type `ptrdiff_t` est défini par un `typedef` à partir d'un autre type entier, souvent `long` ou `long long`

12. Là encore, on peut exprimer cette règle de manière plus technique, par un calcul : `(p - q)` vaut la différence entre les valeurs numériques de `p` et `q`, divisée par `sizeof(data)`.

Pointeurs génériques et préservation des valeurs d'adresses. La norme du langage C garantit que, après l'exécution du code suivant, `p` et `q` contiennent la même adresse :

```c
int tab[] = {1, 2, 3, 4};
int *p = &t[1], *q;
void *r;

r = p;                  // int *  vers void *
q = r;                  // void * vers int *
```

Autrement dit, on ne perd aucune information sur l'adresse d'une donnée lorsque l'on transfère cette adresse d'abord depuis un pointeur vers un pointeur générique, puis inversement du pointeur générique vers un pointeur de même type que le pointeur de départ.

Examinons les conséquences de cette règle, à l'aide d'un exemple minimal. On souhaite écrire une fonction capable d'effectuer un traitement sur plusieurs sortes de données de types distincts : par exemple, tirer un élément au hasard parmi deux choix possibles. On pourrait songer à écrire une fonction pour les `int`, une autre pour les `char`, etc. Les pointeurs génériques permettent de se limiter à l'écriture d'une unique fonction renvoyant une adresse générique choisie aléatoirement parmi deux :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void *draw(void *p, void *q){
  if (rand() % 2)
    return p;
  return q;
}

int main(){
  // initialisation du generateur de nombres aleatoires.
  srand(time(NULL));

  int a = 42;
  int b = 17;

  int *p =  draw(&a, &b);            // conversion implicite en (int *)
  printf("%d\n", *p);                // affichage de 42 ou de 17

  char *s = "abc";
  char *t = "def";

  char *q =  draw(s, t);           // conversion implicite en (char *)
  printf("%s\n", q);                 // affichage de abc ou de def

}
```

La première instruction n'est qu'un simple détail technique : il s'agit d'une des manières d'initialiser le générateur de nombres pseudo-aléatoires du C. La fonction `rand` tire un

nombre au hasard entre `0` et la plus grande valeur du type `int` : en prenant le modulo `2` de ce nombre, on obtient un nombre tiré au hasard égal soit à `0`, soit à `1`.
La fonction `draw` reçoit comme arguments deux valeurs de pointeurs génériques, et renvoie l'une de ces deux valeurs tirée au hasard. Le premier appel se fait sur les adresses de `a` et `b`, de type pointeur vers `int`. Ces valeurs sont converties en valeurs de pointeurs génériques stockées dans deux nouvelles variable `p` et `q`. Le pointeur `pab` récupère la valeur de retour – générique – de `draw` en la convertissant à nouveau pointeur vers `int` (à nouveau, la conversion est implicite, il est inutile de l'écrire explicitement).
La règle précédente garantit qu'il s'agit toujours de l'adresse de `a` si la fonction a renvoyé `p`, et de celle de `b` si elle a renvoyé `q`. Le second appel est similaire, avec cette fois des adresses de chaînes.

Restrictions sur l'usage des pointeurs génériques. Même lorsqu'il contient l'adresse d'une donnée, le type d'un pointeur générique (`void *`) ne fournit aucune information sur la *taille* de cette donnée, ni sur la manière de d'*interpréter* le contenu de la zone-mémoire occupée par cette donnée. Pour ces deux raisons :

- **l'arithmétique des pointeurs ne s'applique pas aux pointeurs génériques,**
- **il est impossible de déréférencer (`*`) un pointeur générique.**

Remarque : conversions entre pointeurs non-génériques. Il est possible de transférer la valeur d'un pointeur non-générique vers un pointeur générique, puis de transférer cette valeur depuis le pointeur générique vers un pointeur de type différent du premier :

```
1 int n = 42;
2 int *p = &n;
3 void *q = p;
4 char *r = q;
```

Il est aussi possible de transférer la valeur d'un pointeur non-générique vers un pointeur non-générique d'un type différent – il faut dans ce cas des conversions explicites :

```
1 int n = 42;
2 int *p = &n;
3 char *r = (char *) p;
```

Il y a cependant deux problèmes dans chacun de ces exemples :

- d'une part, la norme du langage C **ne garantit pas** que ces opérations préservent les adresses : la valeur finale de `r` n'est pas nécessairement celle de `p` [13],
- d'autre part, même en cas d'égalité, la donnée pointée par `p` et `r` est un `int`, et non un `char` : toute tentative pour lire cette valeur en déréférençant `r` produira une valeur indéfinie.

13. Avec l'architecture Intel, ceci dit, il n'y a pas de problème : les pointeurs `p` et `r` auront la même valeur. Autrement dit, les conversions entre pointeurs non-génériques sont possibles en C sur certaines architectures, mais le résultat n'est pas défini par la norme du langage. Notons aussi que sur certaines architectures, des pointeurs de types différents peuvent avoir des formats différents, par exemples les adresses de `int` peuvent avoir une longueur différente des adreeses de `char`.

6.1 Exemple : pointeurs génériques pour les algorithmes génériques

Nous terminerons cette section avec un exemple qui illustre l'utilisation de pointeurs génériques pour écrire des algorithmes génériques, qui agissent sur des structures de données de n'importe quel type. Dans notre exemple, il s'agira de vecteurs pouvant contenir des données de n'importe quel type.

- Appelons *shift droit* d'un vecteur l'opération consistant à décaler d'une position vers la droite tous ses éléments sauf le dernier, qui est déplacé au début du vecteur. Si par exemple la suite des éléments du vecteur est a, b, c, d, e, un shift droit transformera cette suite en e, a, b, c, d.
- Symétriquement, appeleons *shift gauche* d'un vecteur l'opération consistant à décaler d'une position vers la gauche tous ses éléments sauf le premier, qui est déplacé à la fin du vecteur. Si la suite des éléments du vecteur est a, b, c, d, e, un shift gauche transformera cette suite en b, c, d, e, a.
- Pour tout entier k, appelons k-*shift* d'un vecteur l'opération consistant à lui appliquer k-fois : un shift droit si $k \geqslant 0$, un shift gauche si $k < 0$.

L'observation suivante est cruciale pour la suite : le k-shift d'un vecteur `t` donne le même résultat qu'un $k + (L \times n)$-shift, où n le nombre d'éléments de `t` et L un entier quelconque (positif ou négatif).
Par exemple, appliqué à un vecteur `t` à 5 élément a, b, c, d, e, un 6-shift [14] produira le même résultat qu'un unique shift droit. Plus généralement, pour tout entier k positif ou négatif, il existe un unique entier $l \in [0, \ldots, n[$ tel qu'un k-shift de `t` produise le même résultat qu'un `l`-shift de `t`, où n est le nombre d'éléments de `t`.
Notre but est d'écrire une fonction :

```
1 void shift(size_t count, size_t size, void *t, int k)
```

Cette fonction suppose que `t` est l'adresse (générique) du premier élément d'un vecteur contenant `count` éléments, chaque éléments étant de taille `size` en octets. Le type des éléments du vecteur est supposé quelconque, d'où le choix d'un type de pointeur générique pour `t`. La fonction doit appliquer un `k`-shift au vecteur.

Par exemple si l'on écrit :

```
1 int t[] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
2 size_t count = sizeof(t)/sizeof(t[0]);
3 size_t s = sizeof(t[0]);
4 shift(count, s, t, 2);
```

et si l'on affiche le contenu de `t` après le shift, on obtiendra : $10, 11, 1, 2, 3, 4, 5, 6, 7, 8, 9$.
Autre exemple, le 3-shift d'un string :

```
1 char txt[] = "W Szczebrzeszynie chrzaszcz brzmi w trzcinie";
2 size_t q = strlen(txt);
3 shift(q, 1, txt, 3);
4 printf("\"%s\"\n", txt);
```

qui affichera :

14. c'est-à-dire l'application de 6 fois un shift droit

```
"nieW Szczebrzeszynie chrzaszcz brzmi w trzci"
```

Notez que le premier paramètre `q` de shift dans le denier exemple *est le nombre de caractères du string, sans prendre en compte marquant la fin de u string en mémoire.* Le caractère nul n'est donc jamais déplacé, le shift ne concernant que les caractères qui précédent `'\0'`.
La fonction `shift` peut s'implémenter avec très peu de lignes de code, en se servant de la fonction `memmove` de la libraire standard [15] :

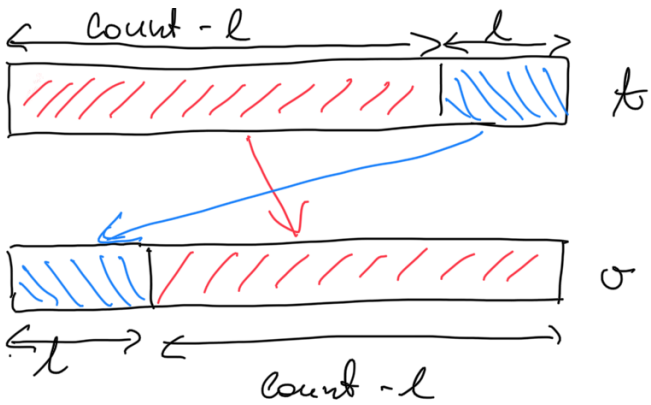
```
void shift(size_t count, size_t size, void *t, int k){
  int l = k % (int)count;
  if(l < 0)
    l += count;
  char v[count * size];
  memmove(v + (l * size), t, (count - l) * size);
  memmove(v, (char *) t + (count - l) * size, l * size);
  memmove(t, v, count * size);
  return;
}
```

La fonction commence par calculer le `l` tel que $0 \leqslant$ `l` $<$ `count` et tel que le `l`-shift du vecteur d'adresse `t` soit équivalent à un `k`-shift de ce vecteur.
Vient ensuite la déclaration d'un vecteur `v` de même taille en octets que le vecteur d'adresse `t` (la taille d'un `char` est d'un octet).
Les deux premiers `memmove` déplacent respectivement :

- tous les octets des `(count - l)` premiers éléments de `t` à la fin de `v`, et,
- tous les octets des `l` derniers éléments de `t` au début de `v`.

Notez que toute l'arithmétique se fait en nombre d'octets, donc pour obtenir le nombre d'octets à déplacer, il faut multiplier le nombre d'éléments à déplacer par la taille `size` en octet d'un élément. Le dessin suivant illustre l'effet de ces deux appels de `memmove` :

15. voir la section 11



Après ces deux appels, le vecteur `v` contient la suite des octets du `k`-shift du vecteur d'adresse `t`. Il ne reste plus qu'à recopier ces octets à l'adresse `t`, ce qui est effectué par le troisième `memmove`.

7 Vecteurs et paramètres des fonctions

Partons de l'exemple suivant :

```
1  int somme(size_t len, int t[]){
2    int s = 0;
3    for( size_t i=0; i < len; i++)
4       s += t[i];
5    return s;
6  }
7  int main(void){
8    int v[] = {9, 7, 1, -2, -4, -6, -8};
9    size_t l = sizeof(v)/sizeof(v[0]);
10   int k = somme(l, v);
11   //....
12 }
```

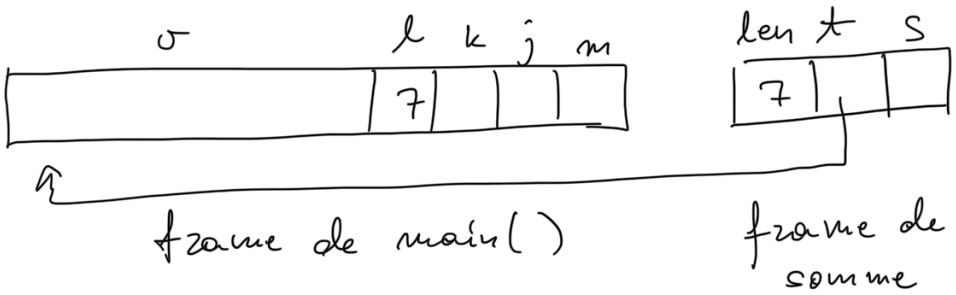
A la ligne 9, on calcule le nombre d'éléments du vecteur `v`. et sans suprise, l'appel de la fonction `somme` depuis `main` aura pour effet, au retour d'appel, de stocker dans `k` la somme des éléments du vecteur `v`. Deux vraies questions demeurent, cependant : quelle est l'information transmise à la fonction `somme` par le nom `v` dans l'appel de cette fonction depuis `main` ? Et quel est le sens exact de la notation `int t[]` dans la déclaration de la fonction `somme` ? Les éléments présentés à la section 5.3 devraient déjà vous fournir un indice sur les réponses.

Sur les appels de fonctions sur des noms de vecteurs. Comme indiqué à la section 5.3, un nom de vecteur dans une expression est immédiatement converti en une expression de type pointeur vers le type de ses éléments, valant l'adresse de départ du vecteur – l'exception à cette règle pour l'opérateur `sizeof` est ce qui permet de calculer la taille de `v` à la ligne 8, mais il ne s'agit que d'une exception. Autrement dit, la seconde valeur passée à la fonction `somme` est un pointeur vers `int`, pointant vers `v[0]`.

Sur le sens de la notation `t[]` pour les paramètres. La notation de paramètres de la fonction `somme` semble indiquer que `t` est un vecteur, mais il s'agit d'une illusion. A la compilation, la première ligne du code sera traduite en :

```
1 int somme(size_t len, int *t)
```

Autrement dit, `t` *n'est pas* un vecteur mais *un pointeur vers* `int`, ce qui est compatible avec le passage du nom `v` converti en pointeur dans l'appel. Ecrire `int t[]` ou `int *t` n'est qu'une question de goût : pour le compilateur, les deux notations sont équivalentes, puisqu'il traduira la première en la seconde.
Nous avons vu par ailleurs à la section 5.3 que la notation indexée est utilisable avec les pointeurs. C'est exactement ce que fait la fonction avec la variable `t` : pendant l'appel, `t` pointe vers `v[0]`, et `t[i] == *(t + i)` permet d'accéder à chaque élément du vecteur `v` par décalage et déréférencement. Le dessin ci-dessous représente l'état de la pile au début de l'appel :



Examinons à présent deux autres formes d'appels :

```
1 int j = somme(4, &v[2]);
2 int m = somme(4, v + 2);
```

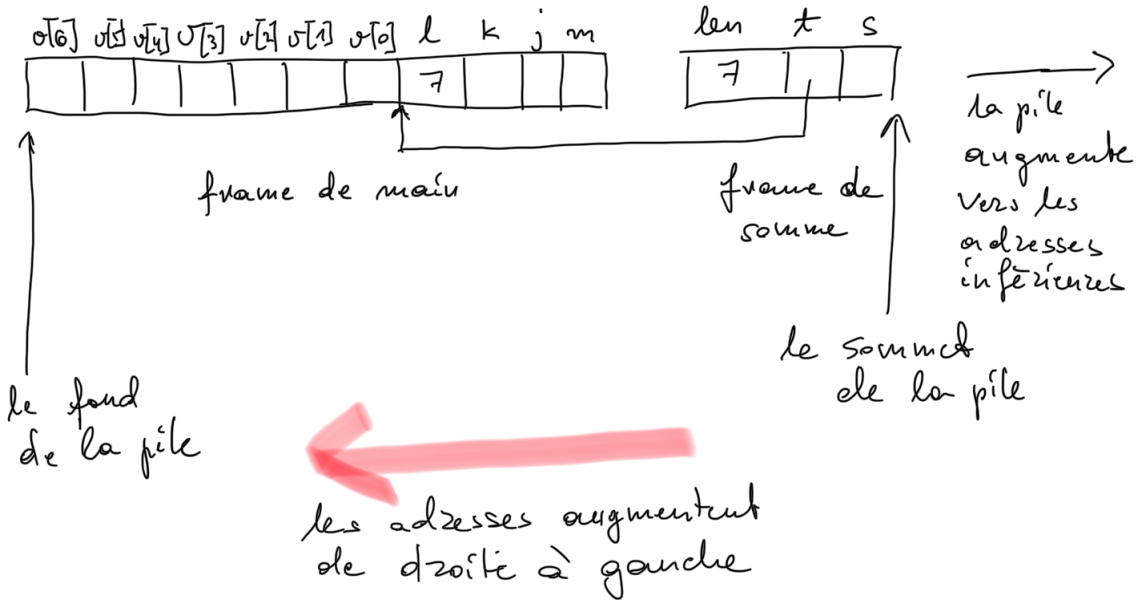
Les deux formes sont en fait équivalentes, et calculeront toutes les deux `v[2] + v[3] + v[4] + v[5]`. Dans la première, on passe à la fonction l'adresse de `v[2]`, de type pointeur vers `int`, qui sera stockée dans une nouvelle variable `t` de même type. Dans la seconde, la conversion du nom `v` en pointeur permet de calculer un décalage de l'adresse du départ du vecteur, à savoir l'adresse de `v[2]`.

Insistons sur le fait que le paramètre `t` de `somme` est un pointeur, donc aussi une *variable* pouvant changer de valeur. La fonction `somme` pourrait aussi s'écrire :

```
1 int somme(size_t len, int t[]){
2   int s = 0;
3   int *fin = t + len ; /* ou int *fin = &t[len] ; */
4
5   for(  ; t < fin ; t++){
6     s += *t; /* ajouter la valeur int pointée par t */
7   }
8   return s;
9 }
```

où dans la boucle, l'incrémentation `t++` permet de balayer l'ensemble du vecteur dont `t` contient intialement l'adresse.

Pour aller plus loin. Comme nous le verrons à la section 12, la pile croît vers les adresses inférieures – le sommet de la pile est à une adresse inférieure à celle du fond de la pile. En revanche, lorsqu'un vecteur `v` se trouve dans l'un des frames de la pile, les adresses de ses éléments augmentent avec les indices : on a `&v[i] < &v[j]` si et seulement si `i < j`. Le dessin suivant représente la pile d'appels lorsque l'on interrompt le programme (en utilisant un breakpoint dans `gdb`) à la ligne 2, au début de `somme(l, v)`.



8 Objets anonymes

Cette section est une digression : ce sujet n'a pas été abordé en cours, et ne sera pas traité à l'examen. Les objets anonymes (*compound literals* dans la norme C99 et C11) sont un ajout au langage relativement récent.

8.1 Vecteurs anonymes

Dans le fragment de code ci-dessous, `(int []){4, 7, 9}` est un vecteur anonyme. Sa taille est calculée à la compilation : elle correspond à la longueur de la suite de trois valeurs `{4, 7, 9}` qui sert à initialiser le vecteur. Le type `(int [])` de cet objet est le type "vecteur d'entiers", et **doit être écrit entre parenthèses** :

```
int *q = (int []){4, 7, 9};
```

Lorsqu'une telle instruction est rencontrée dans une fonction, le vecteur anonyme correspondant est placé sur la pile – cela implique en particulier que son contenu sera perdu au retour d'appel. Ce vecteur est sans nom, mais l'adresse du vecteur est ici stockée dans une variable `q`. Bien sûr, l'accès aux éléments du vecteurse fait de la manière habituelle, via la notation indexée. Mais `q` reste une *variable* et non un vecteur, on peut donc librement changer sa valeur :

```
q++; // on a à présent q[0] == 7, q[-1] == 4 et q[1] == 9
```

Noter que `q` est aussi un pointeur, donc `sizeof(q)` vaut la taille d'un pointeur en octets, pas celle du vecteur anonyme : il n'y a aucun moyen d'extraire depuis le code la taille d'un vecteur construit de cette façon.

Les vecteurs anonymes peuvent être passés en arguments lors d'un appel de fonction. Par exemple, la fonction `somme` de la section 7 peut s'appeler de la manière suivante, la somme renvoyée sera bien sûr celle des éléments du vecteur :

```
int t = somme(5, (int []){5,9,-9,7,4});
```

Il est possible de créer un vecteur anonyme avec une taille fixe. Dans l'exemple suivant, la variable `q` contient l'adresse (du premier) élément d'un vecteur anonyme à 6 éléments qui ne sont pas initialisés :

```
int *q = (int [6]){  };
```

Il est en revanche *impossible* de créer un vecteur anonyme dont la taille ne peut pas être calculée à la compilation :

```
int n = ... ;
int *q = (int [n]){  };
/* ne compile pas, le compilateur ne sachant pas quelle
* est la valeur de n a la compilation
*/
```

8.2 Structures anonymes

Rappelons que les champs d'une variable de type structuré peuvent être initialisés dès sa déclaration, comme par exemple la variable `a` dans le code ci-dessous :

```
typedef struct point{
  double x;
  double y;
} point;

// ...
  point a = {.x = 1.0, .y = 2.0};
  point b, c;
  /* b = { .x = 5.0, .y = 9.8 }; <-- incorrect, une fois b déclaré, b ne peut
    pas être initialisé de cette façon */
  b.x = 5.0;
  b.y = 9.8;
  c = b;  /* OK affectation entre deux variables de type structure est possible
     */
// ...
```

Les valeur des champs peuvent être aussi modifiées via les noms des champs, par exemple comme la variable `b`. Il est aussi possible de faire d'affecter une telle variable, par exemple comme `c`. Il est aussi possible de modifier simultanément tous les champs d'une variable, même après sa déclaration, à l'aide de structures anonymes :

```
//...
c = (point) {.x = -2.0, .y = -3.0};
```

Ici, `(point){.x = -2.0, .y = -3.0}` est une structure anonyme, et l'affectation est une simple affectation entre structures, ce qui est tout à fait possible.

9 Allocation mémoire

```
#include <stdlib.h>
void *malloc(size_t len )
void *calloc( size_t count, size_t len )
void *realloc( void *ptr, size_t len )
void free( void *ptr )
```

1. La fonction `malloc` alloue une tranche de mémoire de taille `len` en octets, et renvoie l'adresse du premier octet de la tranche allouée, ou `NULL` en cas d'échec. La zone mémoire n'est pas initialisée, et contiendra donc initialement des valeurs quelconques.
2. La fonction `calloc` alloue une tranche de mémoire de taille `count * len` en octets. Tous ces octets sont initialisés à 0. L'appel renvoie l'adresse de départ de la zone allouée, ou `NULL` en cas d'échec.
3. La fonction `realloc` permet de modifier la taille d'allocation d'une zone mémoire allouée par `malloc`, `calloc` ou `realloc` : le paramètre `ptr` est l'adresse de cette zone,

`len` est la nouvelle taille souhaitée en octets. La nouvelle taille peut-être supérieure ou inférieure à la taille de la zone pointée par `ptr`.

La fonction renvoie l'adresse d'une nouvelle zone d'allocation, ou `NULL` en cas d'échec. La nouvelle adresse peut être différente de la valeur de `ptr`, auquel cas la zone pointée par `ptr` est automatiquement libérée et ne doit plus être utilisée.

Si la nouvelle taille est supérieure à l'ancienne, les données présentes dans l'ancienne zone sont recopiées au début de la nouvelle - la partie de la nouvelle zone qui suit la dernière donnée n'est pas initialisée.

Si la nouvelle taille est inférieure à l'ancienne, les `len` premiers octets de l'ancienne zone sont recopiés dans la nouvelle, les autres devant être considérés comme perdus.

4. La fonction `free` libère la zone mémoire pointée par `ptr`, qui doit avoir été renvoyé par `malloc`, `calloc` ou `realloc`.

L'exemple suivant illustre la maniere dont on peut se servir de `malloc` et de `realloc` :

```
//...
size_t n = ...  ;
//allocation d'un vecteur de n  double
double *tab = malloc(n * sizeof(double));
assert(tab != NULL);

for(int i = 0; i < n; i++){
  tab[i] = i*i + 0,5 ;
}

//...

// on double le nombre d'elements de tab.
n *= 2;
tab = realloc(tab, n * sizeof(double));
assert(tab != NULL);
//le n premiers éléments de tab ont ete recopies,
//le n suivants non.
```

Noter que la premiere allocation peut aussi s'ecrire :

```
double  *tab = malloc(sizeof(double[n]));
```

où littéralement, `double[n]` est le type des vecteurs de `double` à `n` éléments. On peut aussi écrire, si l'on souhaite intialiser la zone mémoire allouée par des zéros :

```
double *tab = calloc(n, sizeof(int));
```

10 Les pointeurs de structures

Les pointeurs de structures s'utilisent de la même manière que les pointeurs vers les autres types de données. Supposons déclaré le type de structure suivant :

```
struct toto{
  signed char a;
  int b;
};
```

Dans le code ci-dessous, `p` est un pointeur vers `struct toto` initialisé avec l'adresse de la variable `x`.

```
struct toto x = {.a = 2, .b = 6};
struct toto *p = &x;

p -> a = 3;   // raccourci d'ecriture pour (*p).a = 3;
p -> b = 5;   // raccourci d'ecriture pour (*p).b = 5;
```

Puisque `p` pointe vers `x`, le déférencement de `*p` désigne la structure d'adresse `&x`, c'est-à-dire la structure `x` elle-même. On peut donc accéder (ici, en écriture) aux champs de cette structure en écrivant par exemple : `(*p).a = 3;`
Mais la lourdeur de cette notation fait qu'elle n'est presque jamais utilisée : le C propose le raccourci d'écriture `p -> a = 3;` qui, avec le même sens, est celui couramment utilisé.

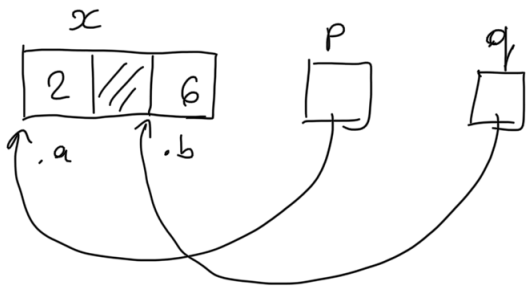
Il est possible d'appliquer l'opérateur `&` aux champs d'une structure, puisque ce sont des l-values. On peut par exemple écrire, après les lignes de code ci-dessus :

```
int *q = &(x.b) ;
```

ou, de manière équivalente :

```
int *q = &(p -> b) ;
```

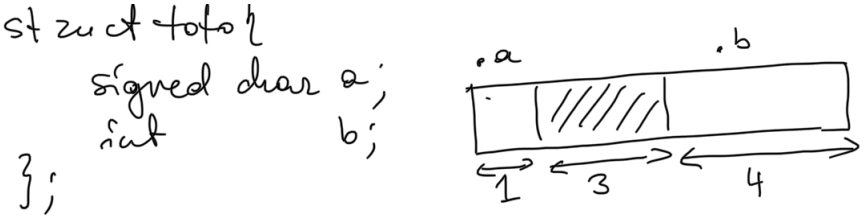
Le dessin suivant illustre l'état de la mémoire après l'une ou l'autre de ces instructions :



Allocations de structures Dans le code suivant, la fonction `malloc` effectue l'allocation d'une zone mémoire dont la taille en octets est celle d'une variable de type `struct toto`. L'adresse d'allocation est récupérée par un pointeur vers `struct toto` :

```
struct toto *p = malloc(sizeof(struct toto));
p -> a = 3;
p -> b = -10;
```

Il est important de noter que la taille d'allocation fournie à `malloc` *doit* être calculée avec l'opérateur `sizeof` : elle *n'est pas* de manière général égale à la somme des tailles des différents champs de la structure.
Par exemple, sur ma machine, `sizeof(struct toto) == 8` tandis que `sizeof(unsigned char) == 1` et `sizeof(int) == 4`. Si l'on observe la zone mémoire occupée par une variable de type `struct toto`, on constate que le champs `b` ne suit pas immédiatement le champ `a`, comme le montre le dessin ci-dessous :



Le champ `a` occupe un octet, suivi de trois octets non-utilisés. Viennent ensuite les 4 octets du champ `b`. Ces octets inutilisés sont ce qu'on appelle des octets de *rembourrage*.
Pourquoi les octets de rembourrage ? Il s'avère que l'architecture d'un processeur impose traditionnellement certaines contraintes sur le placement des données en mémoire, contraintes qui dépendent du type de ces données.
Par exemple si `sizeof(int) == 4`, une contrainte possible sera que les `int` soient placés uniquement à des adresses dont la valeur numérique est multiple de 4. Et si l'on ne respecte pas ces contraintes ? Tout dépend du processeur. Il y a trois cas possibles lorsque l'on essaie d'accéder à une donnée non correctement alignée :

1. sur certains processeur cela provoquera une interruption qui terminera l'exécution du programme,
2. sur d'autres processeurs, il est possible d'accéder à des données non-alignées, mais cela prendra beaucoup plus de temps que l'accès à des données alignées,
3. enfin, il y a des processeurs accédant aux données alignées et non-alignées avec la même vitesse, donc, en théorie, sans aucune règles d'alignement.

Peu importe la situation : en règle générale un compilateur produit du code qui respecte les conventions standards d'alignement, même pour des processeurs pour lesquelles ce n'est pas vraiment nécessaire.
On peut se poser la question de l'alignement des adresses renvoyées par `malloc`. Après tout `malloc` ne sait pas si la mémoire allouée l'est pour un vecteur de `char` (qui peut commencer n'importe où en mémoire), ou pour un vecteur d'`int` (qui doit commencer à une adresse multiple de `sizeof(int)`) ou encore un vecteur de `double`. La fonction `malloc` résout ce problème de façon très simple : elle retourne une adresse d'alignement maximal, de telle sorte que tout type de données puisse être placé à cette adresse.

11 Copier une zone de mémoire avec `memmove`

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t len);
void *memset(void *dst, int c, size_t len);
int  memcmp(const void *s1, const void *s2, size_t n);
```

`memmove` copie `len` octets de la zone mémoire pointé par `src` vers la zone mémoire pointé par `dst`. Les deux zones peuvent chevaucher [16]. Nous avons vu une application de `memmove` dans la section 6.1.

`memset` écrit la valeur `c` transformée en `unsigned char` dans `len` octets à partir de l'adresse `dst`.
Dans l'exemple suivant on initialise tous les éléments de vecteur `tab` à 0.

```
int tab[n];
memset( tab, '\0', sizeof( int[n] ) );
```

`memset` initialise chaque octet de la mémoire avec la même valeur. Ceci implique on ne pourra pas, par exemple, initialiser avec `memset` tous les élément d'un vecteur de `int` avec la valeur 1. En pratique `memset` est utile uniquement pour mettre 0 partout ou pour initialiser un vecteur de caractères avec le même caractère partout.

`memcmp` compare un par un les `n` octets pointées respectivement par `s1` et `s2`. Ces octets sont traités comme les valeurs `unsigned char`. Soit `c1` le ième octet à l'adresse `s1` et `c2` le ième octet à l'adresse `c2`, `c1 <> c2` et i est la première position oùles octets sont différents. Le signe de la valeur retournée par `memcpy` est le signe de la différence `c1-c2`. Si tous les `n` octets sont égaux un à un la fonction retournera 0.
Les pointeurs `s1` et `s2` ne sont pas forcement des pointeurs vers des strings et, contrairement à `strcmp`, la fonction `memcmp` traite le caractère nul comme n'importe quel d'autres caractères.

Comme un exemple d'utilisations de différentes fonctions nous présenterons une implémentation de vecteur dynamique générique. A la création de vecteur on spécifie le nombre d'éléments et la taille en octets d'un élément. Le vecteur est représenté par un pointeur `vecteur` :

```
struct vect{:
  size_t size_elem;     /* taille d'un element en octets */
  size_t nb_elem;       /* nombre d'éléménts*/
  char *v;              /* pointeur vers le premier octet du vecteur */
};
typedef struct vect *vecteur;
```

Le champs `v` de `struct vect` pointe vers le vecteur lui-même. Le fait que `v` est de type `char *` ne veut pas dire que dans le vecteur on stocke les caractères. En fait l'implémentation ne sait rien sur le type d'éléments stockés sauf que chaque élément occupe `size_elem`

16. Il existe une fonction `memcpy` qui fait la même chose mais qui est applicable uniquement quand les deux zones de mémoire ne chevauchent pas. Puisque `memmove` peut être utilisée sans aucune contrainte nous pouvons oublier `memcpy` et toujours utiliser `memmove`. Pourquoi deux fonctions qui font la même chose ? `memcpy` peut-être plus efficace.

octets de la mémoire et que le vecteur contient `nb_elem` d'éléments. Aussi bien on pouvait définir le champ `v` comme le pointeur générique `void *v` mais dans ce cas on ne pouvait pas faire l'arithmétique de pointeurs. Par contre `char *v` permet de faire le calcul d'adresse en octets.
Le vecteur est créer par la fonction :

```
/* creer un vecteur dynamique de nb_elem  places
* size_elem - taille d'un élément en octets
* retourne NULL si la création échoue       */
vecteur vect_new(size_t nb_elem, size_t size_elem){
  vecteur p = malloc(sizeof(struct vect));
  if( p == NULL)
  return NULL;
  p->size_elem = size_elem;
  p->nb_elem = nb_elem;
  /* allouer le vecteiur lui même */
  p->v = calloc( nb_elem, size_elem );
  if(p->v == NULL){
    free(p);
    return NULL;
  }
  return p;
}
```

Pour ajouter et recupérer un élément du vecteur on utilise les fonctions `vect_put` et `vect_get` :

```
/* :
 *  x - pointeur vers la valeur à mettre dans le vecteur
 *  i - l'indice, la nouvelle valeur sera mise à la position i dans le vecteur
 */
int vect_put(vecteur t, size_t i, void  *x){
  if( i >= t->nb_elem )
    return -1;
  /* copier la valeur pointé par x dans le ième élément du vecteur */
  memmove(t->v + i*t->size_elem, x, t->size_elem);
  return 0;
}

/* copier la valeur de i-ème élément du vecteur à l'adresse x
 */
int vect_get(vecteur t, size_t i, void  *x){
  if( i >= t->nb_elem )
    return -1;
  assert( x );
  memmove(x, t->v + i * t->size_elem, t->size_elem);
  return 0;
}
```

Notez que le i-ème élément du vecteur `v` commence à l'adresse `t->v + i * t->size_elem` et que le calcul de l'adresse s'effectue en octets. Les deux fonctions utilisent `memmove` pour

transférer les données entre l'adresse `x` et le ième élément du vecteur pointé par `t->v`. Les fonctions retournent 0 s'il existe un élément à l'indice `i` et -1 sinon.
La fonction suivante change le nombre d'éléments dans le vecteur. Si le vecteur est agrandi les nouveaux éléments sont initialisés à 0. Si le vecteur est raccourci les éléments à la fin du vecteur sont perdus.

```
int vect_change_len(vecteur t, size_t n){
if( n == t->nb_elem ) /* rien à faire */
  return 0;
void *x = realloc( t->v, n * t->size_elem );
if( x == NULL )
  return -1;
t->v = x;

/* mettre 0 dans le nouveaux éléments */
if( n > t->nb_elem )
  memset(t->v + t->nb_elem * t-> size_elem, '\0',
    (n - t->nb_elem)*t->size_elem);
t->nb_elem = n;
return 0;
```

La fonction `vect_find` cherche la valeur pointé par `val` dans le vecteur et retourne l'indice du premier élément qui contient cette valeur ou -1 si la valeur n'est pas retrouvée :

```
long long vect_find( vecteur t, void *val){
  size_t nb = vect_len( t );;
  char value[ t->size_elem ];
  for( size_t i = 0; i < nb ; i++ ){
    vect_get( t, i, value ); /* recuperer ième valeurs de t */
    if( memcmp( val, value, t->size_elem) == 0 )
      return (long long ) i;
  }
  return -1;
}
```

Notez que l'utilisation de `vect_get` pour récupérer le $i$ème élément du vecteur. Comme nous ne savons rien sur le type de cet élément la valeur du ième élément est mise dans le vecteur `char value[ t->size_elem ]` qui contient suffisamment d'octets. La comparaison entre le ième élément du vecteur et la valeur recherchée s'effectue à l'aide de `memcmp`.

Il reste deux fonctions dont le rôle est évident :

```
/* supprimer le vecteur */
void vect_delete(vecteur t){
  free(t->v);
  free(t);
}

size_t vect_len(vecteur t){
  return t->nb_elem;
}
```

Exemple d'utilisation avec un vecteur de `double`s :

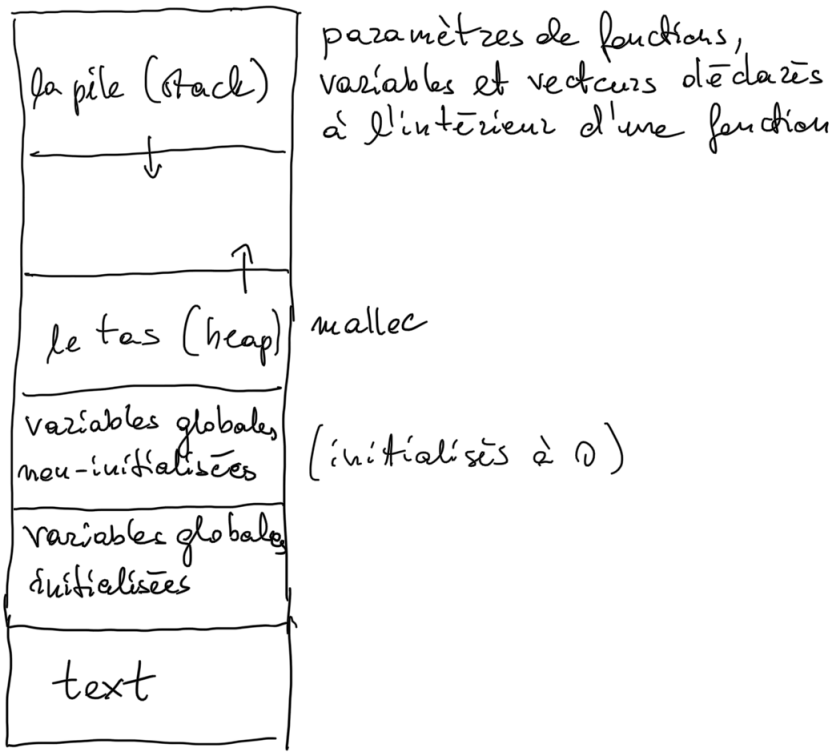
```c
size_t nb = 10;
double d;
vecteur t = vect_new(nb, sizeof(double));
for(size_t i=0;  i < nb; i++){
  d = i*i+1;
  vect_put(t, i, &d);
}
.......
  /* ajouter 5 éléments */
if( vect_change_len( t, vect_len( t ) + 5) == -1){
  fprintf(stderr, "chage_len");
}
size_t n = vect_len( t );
for(size_t i = n-1; i >= nb+1 ; i--){
  d = - (int)(i*i*i);
  vect_put( t, i,  &d);
}
/* chercher 50.00 dans le vecteur */
double q = 50 ;
long long l = vect_find( t, &q);
printf( "indice = %lld\n", l);
```

Pour afficher le vecteur composé de nombre `double`s on pourra utiliser la fonction :

```c
void vect_print_double(vecteur t){
  size_t n = vect_len( t );
  double d;
  for(size_t i=0 ; i < n ; i++){
    vect_get( t, i,  &d);
    printf("v[%i]=%7.2f\n", (int)i, d );
  }
}
```

12 Disposition d'un programme C en mémoire

Pour comprendre l'utilité de la fonction `malloc`, il faut comprendre la disposition d'un programme C en mémoire. Sur ce dessin les adresses augmentent vers le haut :



Le premier segment `text`, en bas, contient le code compilé du programme. Le segment suivant vers le haut contient toutes les variables globales initialisées. Rappelons qu'une variable globale est une variable déclarée à l'extérieur de toute fonction, donc si

```
int i = 7;
```

apparaît dans le code du programme comme variable globale, la variable `i` est allouée dans ce segment et reçoit la valeur 7. Ces deux éléments sont les seuls segments du programme existant après sa compilation et avant son lancement. Tous les autres segments sont alloués au moment du lancement.

Le segment des variables globales non-initialisées contient, comme son nom l'indique, toutes les variables globales qui ne sont pas explicitement initialisées dans le code. Par exemple, si

```
double x ;
```

est une variable globale, elle sera allouée dans ce segment au lancement du programme. Tous les octets des variables de ce segment sont initialisés à 0.

Vient ensuite le **tas** (ou **heap** en anglais), le segment contenant toute la mémoire allouée par les appels des fonctions d'allocation mémoire (*e.g.* `malloc`). Les zones mémoire allouées dans le tas restent allouées jusqu'à ce que le programme les libèrent par `free` (ou jusqu'à sa terminaison). Le tas grandit vers les adresses hautes.

Enfin, tout en haut, se trouve le dernier segment : la **pile** (ou **stack** en anglais). La pile commence à la toute dernière adresse de son segment, et grossit vers le bas. La gestion de la pile est automatique : à chaque nouvel appel de fonction, le système place un nouveau frame contenant les valeurs des paramètres pour cet appel, les variables locales, etc. A chaque retour d'appel, son frame est supprimé de la pile [17].

13 Avec ou sans allocation ?

Quand est-il nécessaire d'allouer de la mémoire avec `malloc` ?

13.1 Vecteurs utilisés localement

Examinons les deux fonctions suivantes :

```
1  void f(size_t n){
2    double *tab = malloc(n * sizeof(double));
3    h(n, tab);       /* faire des calculs */
4    free(tab);       /* liberer tab */
5    return;
6  }
7
8  void g(size_t n){
9    double tab[n];
10   h(n, tab);       /* faire des calculs */
11   return;
12 }
```

Les deux fonctions `f` et `g` effectuent le même traitement : un appel de la fonction `h` sur l'adresse d'un vecteur alloué par `malloc` pour `f`, sur un nom de vecteur local comverti en en pointeur pour `g`.

Dans le cas de `f`, le vecteur alloué se trouve dans le tas (heap), et il ne faut pas oublier de libérer explicitement cette mémoire avec `free` pour éviter les fuites de mémoire.

Dans le cas de `g`, le vecteur `tab` est ce qu'on appelle un VLA (variable length array). Les VLA sont différents des vecteurs de taille constante (comme `int tf[100];`). La taille de ces vecteurs ne peut pas être déterminée à la compilation : elle est calculée dynamiquement au moment de l'exécution [18]. La mémoire pour `tab` lors d'un appel de `g` est automatiquement réservée sur la pile (stack) et libérée quand la fonction termine son exéction par `return`.

Le code de `g` est effectivement plus simple que celui de `f` : il est inutile de se rappeler qu'il faut un `free`. Pour les programmes simples, les VAL sont sans doute plus faciles à utiliser que `malloc` [19]

17. En fait *de facto* rien dans ce cas n'est supprimé physiquement de la pile : un registre spécial pointe sur le sommet de la pile, et un retour d'appel ne fait que modifier la valeur de ce registre. Ce sont les *nouveaux* appels de fonctions qui effacent l'ancien contenu de la pile.
18. Les anciennes normes du C ne connaissaient que les vecteurs de taille constantes et les VLC sont une addition récente à la norme (C99).
19. Mais ils peuvent avoir des inconvénients qui ne nous concernent pas ici.

13.2 Créer des objets persistants dans une fonction

Si la mémoire d'un objet (vecteur, structure) construit par un appel de fonction doit être préservée en retour d'appel, nous n'avons pas le choix : les fonctions de la famille `malloc` deviennent indispensables, comme dans cet exemple :

```
int *f(size_t n){
  int *v = malloc(sizeof(int[n]));
  //  équivalent à
  //  int *v = malloc(n * sizeof(int));
  return v; //OK, v pointe sur la mémoire dans le tas
}
```

Dans l'exemple ci-dessus, on ne *peut pas* remplacer `v` par un VLA, **le code suivant est incorrect** :

```
int *g(size_t n){
  int v[n];
  return v;
  /* incorrect, v est vecteur VAL (variable length array)
  * sur la pile. Ce vecteur est dépilé au moment de return
  * donc au retour de la fonction l'adresse retournée
  * n'est plus valable */
}
```

Le problème de la fonction `g` est qu'elle renvoie l'adresse du vecteur `v` *qui est sur la pile*. Au moment où `return v` est atteint, tout ce qui est local à la fonction (en particulier `v`) est dépilé et doit être considéré comme perdu : tout nouvel appel de fonction effacera presque à coup sûr le contenu de `v`. Noter que le compilateur `gcc` (avec l'option `-Wall`) affichera dans ce cas un message d'avertissement (warning) :

```
warning : function returns address of local variable
```

Pour ceux qui ont lu la section 8, le retour de l'adresse d'un vecteur anonyme posera le même problème, pour les mêmes raisons :

```
int *fun(...){
  int a,b,c;
  //...
  int *v = (int []){a, b, c};
  return v;
  /* incorrect, v pointe sur le vecteur anonyme
  * placé sur la pile, donc  dépilé au retour de
  * la fonction fun
  */
}
```

Cette erreur est cependant plus insidieuse : *le code compile sans erreurs et sans avertissements*. Toutes sortes de problèmes pourront donc survenir de façon inopinée pendant l'exécution où, pire encore, le programme s'exécutera sans problèmes mais produira des résultats erronées. Le même problème se pose avec les structure anonymes :

```c
struct point{double x; double y;};

struct point *fun(...){
  double a = ... ;
  double b = ... ;

  struct point *v = &(struct point){ .x = a, .y = b};

  return v;
  /* incorrect, v pointe sur une structure anonyme
  * placée sur la pile, donc dépilée au retour de
  * la fonction fun
  */
}
```

Il faut évidemment, pour résoudre ce problème, se servir de `malloc`. Dans l'exemple suivant une structure anonyme (sur la pile) sert seulement à initialiser `*v` (dans le tas).

```c
struct point *fun(...){

  double a = ... ;
  double b = ... ;

  struct point *v = malloc( sizof(struct point) );

  *v = (struct point){.x = a, .y = b};

  return v;
  //OK, on retourne l'adresse de mémoire
  //allouée par malloc
}
```