Programmation 1, leçon 3

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

14 janvier 2015

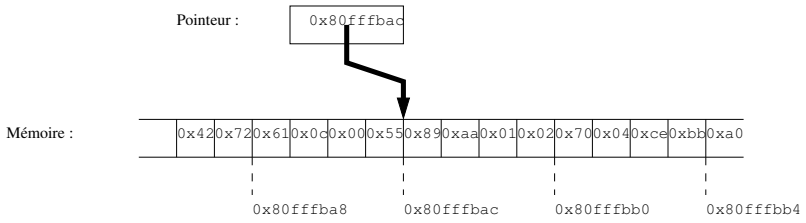
Résumé

Ceci est la version 2 datant du 14 janvier 2014. La version 1 datait du 15 octobre 2013. Merci à Glen Mével.

1 Pointeurs

La notion d'*adresse mémoire* est fondamentale en assembleur. Une donnée entière que l'on peut interpréter comme une adresse est usuellement appelée un *pointeur*. Par exemple, en leçon 2 nous avons abondamment parlé du pointeur de pile : c'est la donnée stockée dans le registre `%esp` (ou `%rsp`), que l'on voit comme l'adresse du sommet de pile. On dit que le registre `%esp` lui-même *pointe* vers le sommet de pile.

L'expression « pointer » suggère une représentation graphique des pointeurs, que l'on utilisera abondamment :



La flèche (en gras ici) va d'une case mémoire de 4 octets (8 octets sur une architecture 64 bits) vers la case *pointée*. Cette dernière contient une donnée, ici l'octet `0x89`. Le pointeur est la case du dessus, qui contient l'adresse `%0x80fffbac` où est stockée la donnée `0x89`.

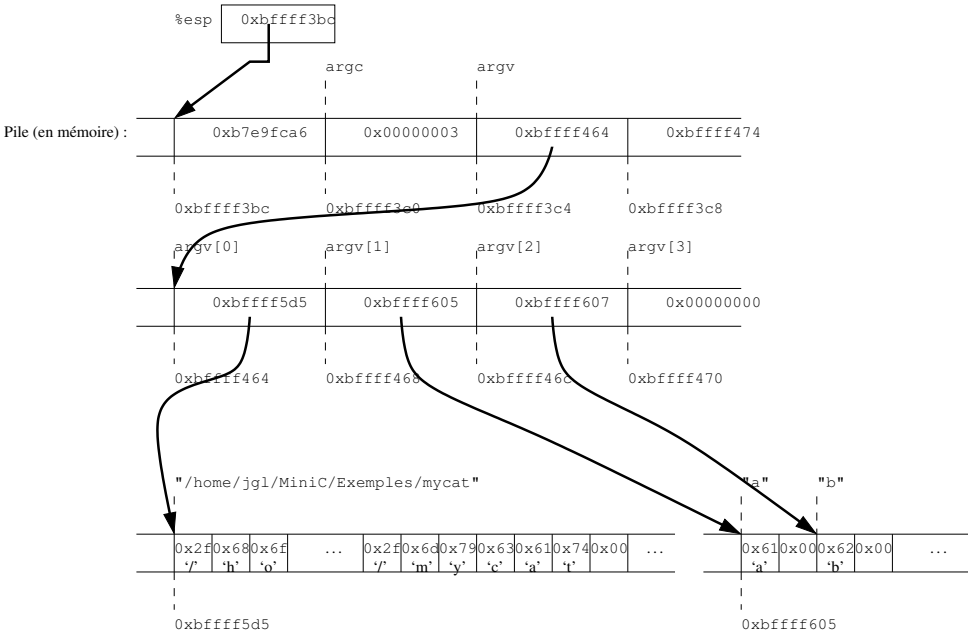
Le pointeur lui-même peut être un registre, comme le registre `%esp`. N'importe quel autre registre peut aussi être vu comme un pointeur, par exemple `%eax`, du moment qu'il contient une adresse valide en mémoire. Mais on peut aussi avoir des pointeurs *en mémoire*. Il suffit de stocker le pointeur lui-même à une autre adresse, sur 4 octets consécutifs.

Mentionnons au passage une petite subtilité. Dans l'exemple ci-dessus, les quatre octets démarrant à l'adresse `0x80fffbb0` peuvent être interprétés comme un entier (ou une adresse).

Un processeur *gros-boutiste* (« big endian » en anglais) considérera que la suite des quatre octets `0x70 0x04 0xce 0xbb` dénote l'entier 32 bits `0x7004cebb`, ce qui peut sembler naturel. Il existe aussi des processeurs *petit-boutistes*, qui considèrent que ces mêmes quatre octets représenteront plutôt l'entier (ou l'adresse) `0xbbce0470`... et c'est le cas des processeurs Intel.

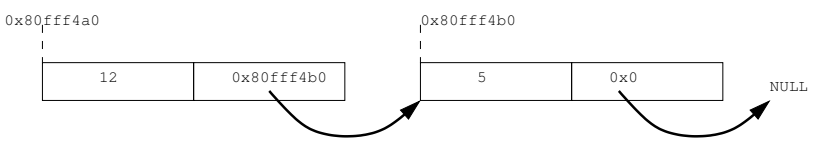
Un *tableau*, par exemple d'entiers, est juste une zone composée d'entiers mis à la queue leu leu. Pareil pour les tableaux de caractères, les tableaux de nombres à virgule flottante, ... et même les tableaux de pointeurs. On peut ainsi imbriquer des structures de données. Dans l'exemple ci-dessus, la zone commençant à `0x80fffba6` peut être vue comme un tableau d'octets. En lisant ces octets en ASCII, et en prenant la convention du langage C qu'une chaîne de caractères est juste un tableau se terminant par l'octet `0x0`, on y lit la chaîne `"Bra\n"`—j'ai pris la convention C d'écrire `\n` pour le caractère de code `0xc`, qui est la fin de ligne (« new line »).

Les tableaux peuvent eux-mêmes contenir des pointeurs vers des données de bases (entiers, caractères), ou même d'autres tableaux. Explorez sous `ddd` ce vers quoi pointe le tableau `argv`, lorsque vous déboguez le programme `mycat`, avec comme arguments les chaînes de caractères `"a"` et `"b"`. Vous devriez obtenir quelque chose comme ceci :



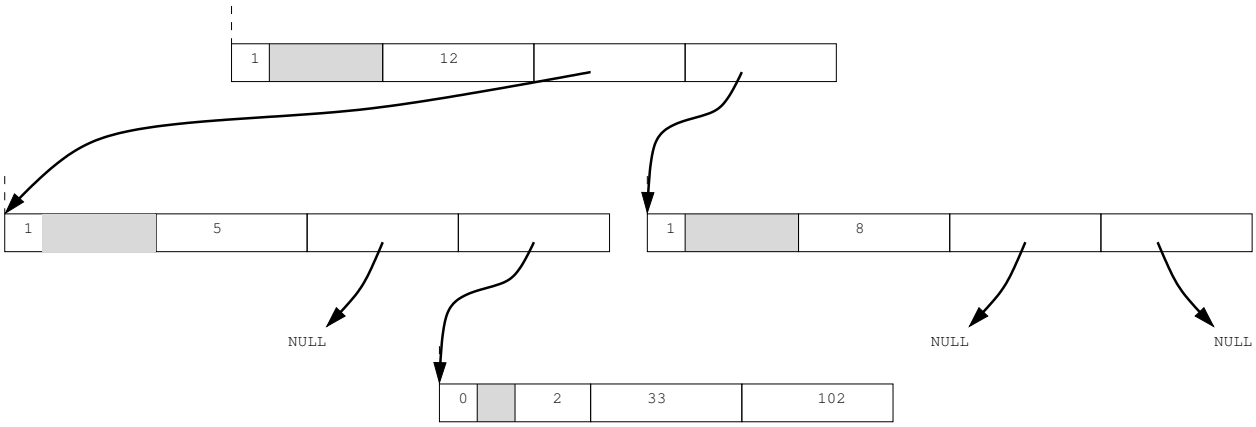
Notez que j'ai représenté certaines zones de mémoire comme des zones où des entiers/adresses 32 bits sont stockées, sur de larges cases, et certaines autres comme des zones d'octets (chaînes de caractères).

On utilisera cette capacité des pointeurs à pointer sur des zones mémoire contenant elles-mêmes des pointeurs à peu près systématiquement. Voici par exemple une représentation typique d'une liste dite *simplement chaînée*, ici à deux éléments, et que l'on écrirait `[12;5]` en Caml.



La liste est stockée sous forme de *cellules* à deux cases, la première contenant la valeur (12 pour la première, 5 pour la seconde), la second contenant un pointeur sur le reste de la liste. Ce dernier pointeur vaut une adresse convenue, et appelée traditionnellement `NULL` lorsqu'on atteint la fin de la liste. Cette adresse `NULL` est une adresse où ne sera jamais rien stocké, pour éviter toute ambiguïté. En général (mais ce n'est pas obligé), l'adresse `NULL` est l'adresse égale à `0x0` [1]. L'intérêt d'une telle structure par rapport à un tableau est qu'on peut l'étendre, ou enlever des éléments en tête de liste, par de simples manipulations de pointeurs.

Et on peut faire bien plus compliqué encore ! La figure suivante montre par exemple un arbre binaire. Chaque sommet interne est repéré par l'octet `1` en début de cellule, et contient une valeur entière (`12` pour la cellule du haut) et deux pointeurs vers les deux sous-arbres fils. Certains de ces sous-arbres sont absents (`NULL`), certains autres sont des feuilles (la cellule du bas). Nous distinguons les sommets internes des feuilles par le fait que l'octet initial vaut `2`. (Ce genre d'octets servant à distinguer plusieurs sortes de zones mémoire s'appelle des *tags*.) On ne donne plus les adresses réelles stockées dans les cases pointeurs : on aura compris qu'elles ne sont pas d'une utilité fondamentale. Les zones en grisé sont des zones inutilisées, contenant bien sûr des octets, mais dont la valeur nous est indifférente. (Ce sont des zones de *rembourrage*, ou *padding*, qui servent à assurer que les données 32 bits sont stockées à des adresses multiples de 4. On parle de contrainte d'*alignement*. Elles sont nécessaires pour accéder aux mots 32 bits sur la plupart des processeurs ; pas sur les Intel, mais accéder sur Intel à un mot 32 bits à une adresse non multiple de 4 est plus lent...)



1.1 Les pointeurs (et les tableaux) en C

Le langage C est une notation de haut niveau pour des structures de données de très bas niveau. La notion d'adresse mémoire est y notamment représentée par les données de type *pointeur*.

1. Et, pour ajouter à la confusion, caster le nombre 0 à un type pointeur en C doit produire le pointeur `NULL`, même si ce dernier n'est pas l'adresse 0. Ceci est un message destiné aux geeks. Pour les autres, faites comme si je n'avais rien dit.

De même que `int` est le type des entiers machines, et `char` le type des octets, le type `int *` est le type des pointeurs sur des entiers. La lettre « `*` » signifie « pointeur ». Noter qu'elle vient après le type vers lequel on pointe.

Si l'on déclare une variable `p` de type `int *`, alors `p` est une case mémoire contenant une adresse, et à cette adresse on trouvera un entier machine.

- On déclare une variable de type pointeur vers `int` en écrivant :
  ```
  int *p;
  ```
- Pour lire l'entier stocké à l'adresse `p`, on écrit `*p`
- Pour écrire un entier $\langle n \rangle$ à l'adresse `p`, on écrit `*p =` $\langle n \rangle$.

On peut remplacer `int` par n'importe quel type. Par exemple, `char *` est le type des pointeurs vers un octet, dont nous verrons plus bas qu'il est le type des chaînes de caractères en C. Le type `char **` est le type des pointeurs vers des pointeurs vers des caractères (!). Aussi ésotérique soit-il, c'est le type de la variable `argv` dans l'exemple compliqué des tableaux de chaînes de caractères vu plus haut.

Types pointeurs. La syntaxe des types pointeurs peut sembler étrange. Il s'agit d'une syntaxe *imitant l'usage*. Par exemple, la déclaration `int *p;` est sans doute plus facile à comprendre si on l'interprète comme « `*p` est de type `int` ». Ca fonctionne pour tous les types : déclarer `char **argv;` permet d'accéder à un caractère par l'expression `**argv`, équivalente à `*(*argv)`. Cela va très loin, même si peu de gens osent s'aventurer sur ce terrain. Par exemple, pour déclarer que `f` est une variable qui doit contenir l'adresse d'un morceau de code (assembleur) qui implémente une fonction prenant une chaîne de caractères `s` en entrée et renvoie un entier en sortie, on écrira :

```
int (*f) (char *s);
```

et pour appeler la fonction dont l'adresse est stockée dans `f`, sur la chaîne `argv[0]` par exemple, et récupérer son résultat dans `i`, on pourra écrire :

```
i = (*f) (argv[0]);
```

où `i` est de type `int`.

▸ **EXERCICE 1.1**

Un truc pour obtenir le type d'une variable C est de regarder sa déclaration, par exemple `int i`, et d'enlever le nom de la variable (ici, `i`) : le type de `i` dans ce cas est juste `int`. Quel est le type de `p`, de `f` dans les exemples ci-dessus ? (Non, ne cherchez pas à enlever des parenthèses ou quoi que ce soit d'autre, juste le nom de la variable. Et non, vous ne rêvez pas.)

Chaînes de caractères, tableaux. Une caractéristique étonnante du langage C est que déclarer une variable de type pointeur vers, disons `int`, la déclare automatiquement comme un pointeur vers, non pas juste un entier, mais une suite d'entiers (de longueur indéfinie : il faudra se donner un moyen par ailleurs de connaître la longueur de ce tableau ; dans l'exemple de `argv`, c'est une autre variable, `argc`, que l'on utilisera pour contenir le nombre d'éléments du tableau).

C'est pourquoi `char *` est aussi le type des (pointeurs vers des) *tableaux d'octets*, et est utilisé comme type des chaînes de caractères. Additionnellement, on convient de représenter les chaînes de caractères dans de tels tableaux, et la fin de la chaîne est identifiée par l'octet `0x0`. C'est pourquoi la chaîne de caractères verb :"a" :, par exemple, est représentée par la suite des octets `0x61` (code de `a`) et `0x0`.

▸ **EXERCICE 1.2**
Peut-on représenter une chaîne de caractères en C qui contient le caractère de code ASCII 0 ? (Pour les fans : ce caractère porte le nom de `NUL`... et ne doit pas être confondu avec le pointeur `NULL`.)

Toute chaîne de caractères est stockée à une certaine adresse, déterminée par le compilateur C pour les chaînes constantes comme `"a"` (par opposition aux chaînes calculées par programme). Cette adresse est de type `char *`.

A noter qu'une même chaîne peut être représentée à plusieurs adresses différentes. Le test d'égalité s'applique aux pointeurs, mais n'opère qu'une *comparaison d'adresses*. Il est donc tout à fait possible que le test `s==t` retourne faux (`0`) alors même que les chaînes de caractères `s` et `t` sont identiques ; il suffit qu'elles soient stockées à des adresses différentes.

Pour comparer deux chaînes de caractères, on utilisera donc la fonction standard `strcmp`. Sa déclaration est :

```
int strcmp (char *s, char *t);
```

Elle prend deux chaînes deux caractères en entrée `s` et `t`, et retourne 0 si elles sont égales, un nombre strictement négatif si `s` vient avant `t` dans l'ordre lexicographique, et un nombre strictement positif.

Sous la ligne de commande, tapez :

```
man strcmp
```

pour en savoir plus sur cette fonction. La bibliothèque standard de C contient bien d'autres fonctions sur les chaînes : `strdup` pour dupliquer une chaîne, `strstr` pour trouver une sous-chaîne dans une chaîne, etc.

Puisqu'un pointeur est un tableau (du moins en ce qui concerne le type, pas la classe de stockage... voir plus loin), il est naturel d'utiliser une notation d'accès à un tableau : pour un objet `p` de type $\langle t \rangle$ `*`, et un entier machine i, `p[`i`]` désignera l'élément (de type $\langle t \rangle$) numéro i du tableau à l'adresse pointée par `p`.

Les indices commencent à 0. Ceci a au moins deux conséquences remarquables :
- Un tableau `p` à n éléments aura comme entrées `p[0]`, `p[1]`, ..., `p[`$n-1$`]`. Attention : `p[`n`]` est *en-dehors* de la plage du tableau, s'il n'a que n éléments !
- La notation `p[0]` est tout simplement *synonyme* de `*p`.

On peut donc par exemple implémenter `strcmp` comme suit :

```
int strcmp (char *s, char *t)
{
  int i;
```

```
  char cs, ct;

  for (i=0;; i++) /* pas de test d'arret, on sortira par 'break' */
    {
      cs = s[i];
      ct = t[i];
      if (cs==0) /* fin de chaine s */
        if (ct==0) /* fin de chaine t aussi => chaines egales */
          break; /* on sort de la boucle */
        else return -1; /* chaine s plus courte */
      else if (ct==0) /* chaine t plus courte */
        return 1;
      else if (cs < ct)
        return -1;
      else if (cs > ct)
        return 1;
      /* sinon cs==ct et on continue. */
    }
  return 0; /* chaines egales */
}
```

▸ **EXERCICE 1.3**
Proposez une implémentation de `strstr`.

Arithmétique des pointeurs. Les inventeurs de C sont allés beaucoup plus loin, et ont inventé l'*arithmétique des pointeurs*. Supposons que l'on ait déclaré $\langle t \rangle$ `*p`. Pour chaque entier machine i, l'élément `p[`i`]` sera stocké à l'adresse obtenue comme suit. Soit a l'adresse, valeur de `p`, où démarre le tableau. Chaque élément du tableau occupe un certain nombre d'octets, disons k. L'élément `p[0]` se trouve à l'adresse a, l'élément `p[1]` à l'adresse $a + k$, ..., l'élément `p[`i`]` à l'adresse $a + ki$.

Cette adresse peut être calculée, en C...simplement par l'opération d'addition : `p+`i va calculer l'adresse $a + ki$ de l'élément `p[`i`]`. Attention : l'adresse où `p[`i`]` est stocké n'est *pas* $a + i$, mais $a + ki$. Par exemple, dans un tableau d'entiers sur une architecture 32 bits, $k = 4$, et `p[`i`]` sera stocké en $a + 4i$!

Au passage, la taille k des éléments de type $\langle t \rangle$ s'obtient en C par l'expression `sizeof(`$\langle t \rangle$`)`. Par exemple, sur une architecture 32 bits, normalement, `sizeof(int)` vaut 4 (octets).

Il existe aussi en C un opérator `&`, qui prend l'adresse de ce qui suit (si ceci a un sens). On aurait donc pu calculer l'adresse de l'élément `p[`i`]` en écrivant `&p[`i`]`. C'est un synonyme exact de l'expression plus étrange `p+`i.

En fait, c'est à l'envers que ça fonctionne : les notations d'accès aux éléments des tableaux `p[`i`]` sont du *sucre syntaxique* (des constructions non indispensables aux langages, et définissables en termes de constructions plus primitives). Les compilateurs C, en voyant `p[`i`]`, traduisent cette expression en l'expression synonyme `*(p+`i`)`.

▸ **EXERCICE 1.4**

Ceci a des conséquences troublantes. Notamment, l'addition est toujours considérée comme commutative en C. Quelle est donc la valeur de `13["Ce n'est pas une blague"]` ? Indication : commencez par les valeurs de `&"Ce n'est pas une blague"[13]`, puis de `"Ce n'est pas une blague"[13]`.

▸ **EXERCICE 1.5**

Justifier les égalités suivantes : `&*p=p`, `*&x=x`, `p[i]=i[p]`, `(&p[i])[j]=p[i+j]`.

On peut donc aussi incrémenter un pointeur, comme on pouvait incrémenter un entier, par l'opérateur `++`, ou le décrémenter par l'opérateur `--`. On notera que :

- `p++` incrémente `p` (de 1, donc l'adresse augmente de la taille du type de l'objet pointé), et retourne l'ancienne valeur de `p` ;
- `p--` décrémente `p` (de 1, donc l'adresse augmente de la taille du type de l'objet pointé), et retourne l'ancienne valeur de `p` ;
- `++p` incrémente `p` (de 1, donc l'adresse augmente de la taille du type de l'objet pointé), et retourne la nouvelle valeur de `p` ;
- `--p` décrémente `p` (de 1, donc l'adresse augmente de la taille du type de l'objet pointé), et retourne la nouvelle valeur de `p`.

On pourrait ainsi réécrire `strcmp` comme suit :

```
int strcmp (char *s, char *t)
{
  char cs, ct;

  for (;;) /* boucle infinie, on sortira par 'break' */
    {
      cs = *s++; /* lit le caractere en *s et incremente s */
      ct = *t++; /* pareil avec t */
      /* le reste du code est comme dans l'implementation precedente */
      if (cs==0) /* fin de chaine s */
        if (ct==0) /* fin de chaine t aussi => chaines egales */
          break; /* on sort de la boucle */
        else return -1; /* chaine s plus courte */
      else if (ct==0) /* chaine t plus courte */
        return 1;
      else if (cs < ct)
        return -1;
      else if (cs > ct)
        return 1;
      /* sinon cs==ct et on continue. */
    }
  return 0; /* chaines egales */
}
```

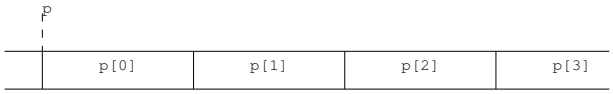
Cette nouvelle façon d'écrire est plus efficace dans un compilateur non optimisant : on se contente d'incrémenter des pointeurs, plutôt que de calculer des adresses de la forme $a + k \times i$ à chaque tour de boucle. De nos jours, les techniques d'optimisation des compilateurs sont tellement au point qu'il n'y a aucun intérêt spécial à écrire nos boucles par incrémentation de pointeur plutôt qu'à accéder aux éléments de tableaux par `s[i]`, `t[i]`.

Tableaux. Il y a deux façons d'allouer de la place pour un tableau. Supposons que nous voulons allouer de la place pour un tableau `p` de 20 entiers. On peut déclarer `p` directement comme un tel tableau :

```
int p[20];
```

Ceci a pour effet de réserver $20\times$`sizeof(int)` octets, soit sur la pile si `p` est une variable locale à une fonction, soit dans une zone de données statiques sinon (nous expliquerons les notions de variable *globale*, *locale*, et *statique* dans la leçon prochaine)

Ce que l'on a en mémoire après cette déclaration se représente comme suit :

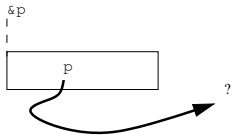


et ainsi de suite sur 20 éléments.

On peut aussi déclarer `p` comme un pointeur sur `int` :

```
int *p;
```

Mais attention ! Ceci n'alloue pas le tableau. Si l'on s'en tenait là, la mémoire ressemblerait à ça :



où la case mémoire de `p` contiendrait un entier arbitraire. (On dit que `p` est un pointeur indéfini, ou « dangling pointer » en anglais).

Pour allouer effectivement le tableau, on peut utiliser la fonction d'allocation mémoire standard `malloc` :

```
p = malloc (sizeof(int [20]));
```

(Eh oui ! Rappelez-vous : on obtient un type en imaginant sa déclaration, par exemple `int p[20]`, et en supprimant le nom de la variable déclarée... donc `int [20]` est le type des tableaux de 20 entiers machine. Il est plus traditionnel d'écrire `p = malloc(20*sizeof(int))`, mais ceci ignore le rembourrage éventuel nécessaire pour maintenir les contraintes d'alignement.)

La fonction `malloc` prend un entier n en paramètre, et retourne l'adresse d'une nouvelle zone mémoire de n octets ; du moins si c'est possible, c'est-à-dire s'il reste suffisamment de mémoire à allouer, sinon `malloc` renvoie `NULL`. On doit donc en général compléter l'appel à `malloc` ci-dessus par :

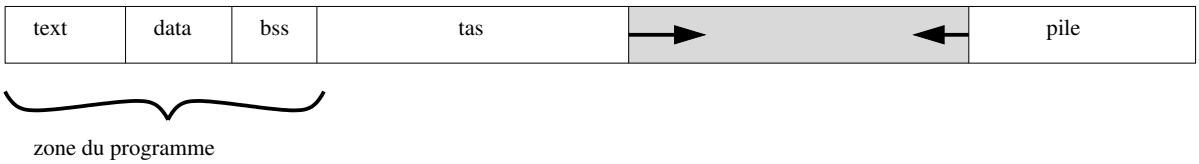
```
if (p==NULL) /* pas assez de memoire */
  {
    [...] /* faire quelque chose!
             'fprintf(stderr,"Plus de memoire: stop.\n"); abort();'
             est la solution la plus brutale.
             Liberer de la memoire non essentielle,
             ou appeler un garbage collector est plus sympa,
             mais plus complique. */
  }
```

Une fois qu'on aura fini d'utiliser la zone mémoire pointée par `p`, on devra la libérer en faisant :

```
free (p);
```

Mais il faut bien faire attention à ne libérer que des pointeurs alloués par `malloc`, et pas déjà libérés. Sinon, le programme peut planter ou se comporter bizarrement.

La fonction `malloc` effectue une allocation mémoire *dans le tas*. Pour comprendre ce que ça veut dire, il est bon de comprendre que la mémoire d'un ordinateur exécutant un programme est découpée en zones, typiquement :



où :

- la zone où est stockée le programme est divisée en trois sous-zones, « text » (les instructions du programme), « data » (les données initialisées, typiquement les constantes chaînes de caractères comme `"r"` ou `"Ce n'est pas une blague"` écrites explicitement dans le programme, ou bien les déclarations globales de la forme `int x=20;` par exemple), et « bss » (les données non initialisées, par exemple obtenues par des déclarations de variables globales telles que `int x;` ou `int p[20];`) ; j'expliquerai la notion de variable *globale* et *locale* dans la prochaine leçon) ;
- le *tas* (« heap » en anglais), qui est en zone de taille variable où s'opèrent les allocations effectuées par `malloc`, et permettant d'obtenir ainsi des zones supplémentaires *dynamiquement*, c'est-à-dire au fur et à mesure que le programme s'exécute ; c'est une zone extensible, comme l'indique la flèche pointant vers la droite sur le graphique ;
- la *pile*, où l'on trouve notamment les variables locales, c'est-à-dire par exemple les variables déclarées à l'intérieur des fonctions (sauf variables dites statiques) ; on y trouve aussi les adresses de retour (voir la section **??**).

En attendant, ce qu'on aura en mémoire après `p = malloc(sizeof(int [20]))` diffère un peu de ce qu'on avait en mémoire en faisant `int p[20]` :

et ainsi de suite sur 20 éléments. Noter que contrairement au cas précédent, il y a une vraie variable `p`, contenant un pointeur. De plus, cette variable a elle-même une adresse `&p`. (... Plus ou moins. Rien n'interdit que le compilateur préfère ne pas allouer de la place pour `p`, mais décide de le garder dans un registre, lorsque `p` est une variable locale. Ecrire `&p` va cependant forcer le compilateur à lui allouer de la place en pile.)

Ce qui change entre les deux façons d'allouer 20 entiers machine, ce n'est pas le type : dans les deux cas, `p` a le type `int *` (ou `int []`, qui est synonyme). Ce qui change, c'est la *classe de stockage*, c'est-à-dire la stratégie d'allocation en mémoire : dans le premier cas, `p` a une classe de stockage « tableau », dans le second cas, une classe de stockage « pointeur ».

▸ **EXERCICE 1.6**

On peut créer des tableaux à plusieurs dimensions en C. Voici comment créer un tableau 30×40 d'entiers :

```
int p[30][40];
```

Dessiner schématiquement la liste des cases du tableau `p` en mémoire. On réalisera qu'il s'agit juste d'un tableau de 30 éléments, chaque élément étant un tableau de 40 éléments. Quelle est la différence avec les déclarations suivantes ?

1. ```
   int *p[30];
   int i;
   for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
   ```
2. ```
   int **p;
   int i;
   p = malloc(sizeof(int *[30])); /* des types de plus en plus fous! */
   if (p==NULL) abort();
   for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
   ```
3. ```
   int p[30*40];
   ```
4. ```
   int *p;
   p = malloc(sizeof(int [30*40]));
   ```

▸ **EXERCICE 1.7**

Pourquoi d'après vous la plupart des compilateurs C conformes à la norme ANSI C 89 (pas `gcc`, ni les compilateurs plus récents, conformes à la norme ANSI C 99) refusent-t-ils le code suivant ?

```
int f (int n)
```

```
{
  int p[n];

  [...] /* reste non pertinent */
}
```

1.2 Les pointeurs en Pascal

Pascal, qui est un langage légèrement antérieur à C, utiliser des conventions superficiellement similaires à celles de C. Les principales différences sont :

- pas d'arithmétique des pointeurs, les pointeurs, les tableaux et les chaînes de caractères sont des choses bien différentes, et en particulier de types distincts ; chaque tableau et chaque chaîne est munie d'un intervalle d'indices permis, et accéder aux éléments du tableau ou de la chaîne en-dehors de l'intervalle provoque une erreur ;
- l'allocation dans le tas s'effectue avec `new`, qui contrairement à `malloc` ne prend pas une taille d'objet à allouer, mais le pointeur à allouer lui-même ; ceci évite les erreurs sur les longueurs de zones à allouer.

Le reste est essentiellement une différence de syntaxe, donc peu importante. On écrira par exemple :

`var i : integer;`		`int i;`
`var p : integer^;`	pour	`int *p;`
`var p : array [0..19] of integer;`		`int p[20];`
`var s : string[50];:`		`char s[51];`

Dans le troisième cas, on notera que Pascal permet aux indices d'un tableau de varier dans un intervalle : on pourrait écrire `var p : array [1..20] of integer;`, mais ses éléments seraient alors `p[1]` jusqu'à `p[20]` plutôt que (comme en C) `p[0]` jusqu'à `p[19]`. On notera aussi que l'on doit déclarer `int p[20]` pas `int p[19]` en C pour obtenir autant d'éléments (soit 20).

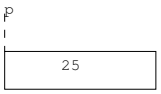
Dans le quatrième cas, noter qu'une chaîne de 50 caractères (au plus) en C sera codé sous forme d'une zone de 51, pas 50, octets (en supposant un codage représentant un caractère par un octet, comme en ASCII ou en ISO-latin1). En C, on doit aussi stocker le caractère `'\0'` (de code `0x00`) de fin de chaîne.

1.3 Les pointeurs en Caml

La syntaxe de Caml laisserait à penser que le type pointeur en Caml est le type `ref`.

Ce n'est qu'une approximation de la réalité. Donc τ `ref` est le type des *références* vers un objet de type τ, et référence est (presque) un synonyme de pointeur.

Si on écrit `let p = ref 25 in ...`, ceci créera une case mémoire (une *référence*) contenant l'entier `25` :



Jusqu'ici, ceci ressemble à ce qu'on obtiendrait si l'on écrivait `int p=25;` en C. Les différences sont les suivantes :

- la case mémoire contenant la valeur (`25` ici) est allouée *dans le tas* ;
- on ne peut pas allouer de référence sans dire quelle valeur on veut y stocker initialement (`25` ici) ; ceci évite tous les problèmes de pointeurs non définis ;
- les variables en Caml ne sont *pas* des cases mémoires que l'on peut changer, comme dans les langages impératif (C, Pascal, Java, etc.) : une variable n'est qu'un nom pour une valeur, qui ne changera jamais ; ici, cette valeur est l'adresse de la référence ;
- pour lire la valeur stockée dans la référence `p`, on utilise la fonction `! : α ref -> α` (α est une variable de type, qui dénote n'importe quel type : c'est ce qu'on appelle le *polymorphisme* paramétrique) ; donc `!p` retourne la valeur `25`
- c'est la référence dont on peut modifier le contenu, pas la variable. Pour ceci, on utilise la fonction `:= : α ref -> α -> unit`. Le type `unit` est le type qui ne contient qu'une valeur, notée `()`, est est utilisé pour dénoter le type de retour des fonctions qui ne retournent rien de signifiant. La fonction `:=` est écrite en *infixe*, c'est-à-dire entre ses deux arguments. Donc, par exemple, `p := 16` remplace le contenu de la référence nommée `p` par la valeur `16`.

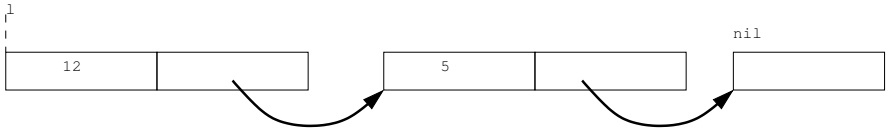
Une traduction intuitive de ces constructions Caml en C serait :

`let p = ref 25 in`	`int *p = malloc (sizeof(int *));` `*p = 25;`
`!p`	`*p`
`p := 16`	`*p = 16;`

On notera que dans la traduction vers C, la valeur de `p` n'est jamais modifiée. (L'appel à `malloc` n'est qu'une approximation, de nouveau ; d'abord parce que Caml alloue une zone mémoire plus grande, contenant certains autres gens ; ensuite parce que Caml a une politique d'allocation mémoire spécifique, où les données non utilisées sont automatiquement libérées par un programme auxiliaire appelé *garbage collector* : il n'y a pas d'appel à `free` en Caml.)

Mais Caml contient aussi de nombreux pointeurs *cachés*.

Lorsqu'on écrit, par exemple `let l = [12; 5] in`..., la variable `l` contient réellement un pointeur vers une liste chaînée comme suit :



En fait, tout objet Caml est alloué en mémoire, et repéré par son adresse. Le type des listes est :

```
type 'a list = nil
             | :: of 'a * 'a list
```

disant qu'une liste est soit la liste vide `nil` (aussi abrégée `[]`), soit l'ajout en tête d'un élément x à une liste ℓ, notée x`::`ℓ (en infixe). La notation `[12; 5]` est une abréviation de `12::(5::nil)`. Or :

– `nil` est alloué en mémoire (la case de droite, ci-dessus) ;
– chaque fois que l'on écrit x`::`ℓ, Caml alloue une case mémoire à deux champs, stocke x dans le premier et ℓ dans le second, et retourne l'adresse de la case mémoire,

ce qui explique la représentation de la liste `[12; 5]` ci-dessus.

Les données Caml sont donc représentées à l'aide de quantité de pointeurs, mais en quoi ceci nous touche-t-il ? On pourrait par exemple décider de représenter les listes autrement, sans pointeurs, sous forme de tableaux que l'on réallouerait progressivement au fur et à mesure qu'ils grossissent, par exemple, et ceci serait a priori invisible à l'utilisateur, non ? (Une telle technique de représentation des listes par des tableaux existe, et a été utilisée en Lisp dans les années 1960-70 : c'est le *CDR coding*, je vous laisse la découvrir sur Wikipedia par exemple.)

Ce n'est *pas* invisible à l'utilisateur, en fait, lorsque l'on inclut des références à l'intérieur des structures de données, comme le démontre l'exercice suivant.

▸ **EXERCICE 1.8**

Sous le toplevel Caml (`ocaml`), tapez :

```
let l1 = [ref 1; ref 2];;
```

Ceci crée une liste de deux références. Ecrivez la fonction suivante, qui (en principe) n'est rien d'autre qu'un codage un peu étrange de la fonction identité sur les listes.

```
let list_copy l = match l with
    [] -> []
  | ar::l' -> ref (!ar)::l';;
```

Ecrire :

```
let l2 = list_copy l1;;
```

Visiblement, `l2` a les mêmes éléments que `l1`, mais elles ne sont pas égales (pas à la même adresse), comme on peut le vérifier en tapant :

```
l1==l2;;
```

alors même qu'elles ont le même contenu, comme on peut le vérifier en tapant :

```
l1=l2;;
```

(Vu la différence entre les deux prédicats d'égalité = et == ?) Maintenant, modifiez les deux références de `l2` en écrivant :

```
match l2 with
    r1::r2::_ -> begin r1:=25; r2:=47 end;;
```

(et ignorez l'avertissement "`pattern-matching is not exhaustive`"). Vérifiez que `l2` est maintenant une liste vers deux références vers `25` et vers `47` respectivement. En regardant `l1`, vérifiez que `l1` est maintenant une liste vers deux références, vers `1` et `47`. Autrement dit,

la première référence n'a pas été modifiée, mais la seconde oui. Comment expliquez-vous ça ? Faites un dessin des cases mémoire allouées et des pointeurs entre elles.

Dans l'exercice précédent, vous aurez sans doute remarqué que Caml affiche une référence, disons vers l'entier `47`, avec la syntaxe `{contents = 47}`.

La raison en est que le type `ref` n'est pas un type primitif. Il est défini comme :

```
type 'a ref == {contents : mutable 'a};;
```

où == dans une déclaration de type dit que l'on définit une abréviation pour un type préexistant, alors que = annonce la création d'un nouveau type (voir le type des listes plus haut).

Une référence est donc un *enregistrement* (notion que nous verrons la prochaine fois) avec un unique champ `contents`, lequel est déclaré `mutable`, c'est-à-dire modifiable.

1.4 Les pointeurs en Java

Java est un langage orienté objet, impératif, et au premier abord très proche de C, dont il apparaît comme une extension. Leur proximité est surtout due à une ressemblance des syntaxes relatives de C et de Java... le modèle mémoire étant en revanche beaucoup plus proche de Caml.

On y trouve donc des pointeurs (cachés) partout ! Et il n'y a pas de type référence... ils sont inutiles, car tous les champs sont tous implicitement mutables. La différence avec C sera sans doute plus claire sur un exemple.

Reprenons les listes vues ci-dessus pour Caml. On écrirait typiquement le type des listes d'entiers en C comme :

```
typedef struct liste {
  int valeur;
  struct liste *suivant;
} liste;
```

où `suivant` est un pointeur vers la case suivante, ou `NULL` s'il n'y en a pas (j'expliquerai les `typedef` et les `struct` une prochaine fois ; ce sont des *enregistrements*, que l'on se contentera de voir ici comme des zones contenant un champ entier `valeur` et un champ pointeur `suivant`). La fonction `::` s'écrirait par exemple :

```
liste *cons(int i, liste *l)
{
  liste *ll;

  ll = malloc(sizeof(liste));
  if (ll==NULL) abort();
  ll->valeur = i;
  ll->suivant = l;
  return ll;
}
```

A noter que, pour lire ou écrire dans le champ `valeur` de la liste pointée par `l`, il faut d'abord lire le pointeur (en écrivant `*l`, normalement), puis aller lire le champ `valeur`, ce qui s'écrirait `(*l).valeur`. La notation `l->valeur` est justement une abréviation de cette dernière expression, et lit donc le champ `valeur` de l'objet pointé par `l`.

Récupérer le premier élément d'une liste non vide se ferait donc typiquement par la fonction :

```
int hd(liste *l)
{
  return (*l).valeur;
}
```

où j'écris `(*l).valeur` plutôt que le plus habituel `l->valeur` pour qu'on voie bien la différence avec ce qui va suivre.

En Java en revanche, il n'y a pas de pointeur visible. En trichant un peu, on écrirait donc en Java quelque chose comme :

```
int hd (liste l)
{
  return l.valeur;
}
```

d'où l'étoile a disparu partout (deux occurrences).

J'ai dit « en trichant un peu ». J'ai même beaucoup triché, car il n'y a pas de `struct` en Java, mais des déclarations de classes, et pas de fonction, mais des déclarations de méthodes. On écrirait donc typiquement le code suivant, que je ne tenterai pas d'expliquer (vous verrez l'orienté objet au second semestre).

```
private static class Node<AnyType>
{
   private AnyType valeur;
   private Node<AnyType> suivant;

   public Node(AnyType i, Node<AnyType> l)
   { /* remplace cons, ou :: */
     this.valeur = i;
     this.suivant = l;
   }

   public AnyType hd()
   {
     return this.valeur; /* ou 'return valeur', aussi. */
   }
}
```