partie 3 : Les pointeurs

MOOC Langage C INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 Présentation

1.1 Un pointeur est une variable

Un pointeur est une variable qui peut contenir l'adresse d'une autre variable ou, plus généralement, la référence d'un objet. Un pointeur permet donc d'accéder à cet objet de manière indirecte. La variable possède un type, un pointeur aussi : il ne peut recevoir que l'adresse d'un objet du même type. Un pointeur sur un entier ne peut recevoir que l'adresse d'un entier, à moins de corriger la chose à coup de casting, comme nous l'avons vu (en C, rien n'est sûr et tout est permis) ! En supposant qu'un entier valant `1278` réside à l'adresse `0x8bc4e2` (normalement inconnue de l'utilisateur), nous pouvons représenter par un petit schéma (figure 1) le fait qu'un pointeur ait reçu cette adresse :

FIGURE 1 – Pointage d'une variable

1.2 Conventions d'écriture

Prenons l'exemple d'une variable entière appelée `Pointee` :

```
int Pointee, Largeur;           /* Definition de deux entiers */
int * pEntier;                  /* Definition d'un pointeur */
```

`pEntier` est un pointeur pouvant pointer sur tout objet de type `int`, en particulier (mais pas forcément) sur la variable `Pointee`. Pour l'instant, la valeur qu'il contient est nulle ou quelconque, c'est-à-dire qu'il ne pointe sur rien de précis et l'on ne peut pas encore l'utiliser pour une affectation indirecte.

```
pEntier = &Pointee;
```

Cette instruction affecte à la variable `pEntier` l'adresse de la variable `Pointee`. Maintenant, `pEntier` pointe sur `Pointee`. `&` est l'opérateur unaire placé en préfixe (priorité 2), dont nous avons déjà signalé l'existence au chapitre précédent.

`pEntier` pointe sur un entier, alors :
`*pEntier`
représente l'objet pointé : une variable entière. `*` est l'opérateur unaire d'indirection (priorité 2), à ne pas confondre avec l'opérateur (binaire) de multiplication (priorité 3). On peut donc écrire :

```
Largeur = *pEntier;
Largeur = *pEntier + 4;
*pEntier = 0;
```

ici respectivement équivalent à

```
Largeur = Pointee;
Largeur = Pointee + 4;
Pointee = 0;
```

Notez déjà que `*pEntier` peut aussi bien apparaître à droite qu'à gauche du symbole d'affectation. Les affectations de valeur initiale à des pointeurs peuvent se faire durant leur définition. Si nous écrivons :

```
int   Nombre,
     *Pointeur  = &Nombre,
     *Reference = &Nombre;
```

cela est globalement équivalent à :

```
int Nombre, * Pointeur, * Reference;
Pointeur =  &Nombre;
Reference = &Nombre;         /* ou Reference = Pointeur; */
```

Dans cet exemple, `Pointeur` et `Reference` pointent sur la même variable `Nombre` : `*Pointeur` et `*Reference` sont donc équivalents. Notez bien que quand on écrit :

```
int * Pointeur = &Nombre;
```

c'est bien la variable `Pointeur` qui est initialisée.

1.3 Type généralisé

Sachons lire les définitions de pointeur :

```
int * Pointeur;
 3  2    1
```

— `Pointeur (1)` est le nom d'une variable qui pointe `(2)` sur une variable dont le type est `int (3)`.
— `*Pointeur (2 et 1)` est un objet de type `int (3)`.
— `Pointeur (1)` est de type généralisé `(int *)` c'est-à-dire `3` et `2`.

Un type généralisé est un type de base, `int` dans l'exemple précédent, altéré par un ou des modificateurs de type. L'opérateur unaire `*` est un modificateur de type. Si l'on a :

```
char Mot [12];
```

`Mot` est de type généralisé `(char[])`. Les crochets sont un autre modificateur de type. Cela signifie que `Mot` est une constante, adresse de tableau (peu importe sa taille).

Un troisième modificateur de type, que nous verrons un peu plus loin sont les parenthèses, qui indiquent une fonction retournant le type de base. Ainsi, la fonction principale `main()` est, par défaut, de type généralisé `( int() )`, c'est-à-dire une constante, adresse d'une fonction retournant un entier.

D'une manière générale, nous pouvons donner la recette simple suivante : le type généralisé d'un objet est ce qui reste de sa définition quand on en a retiré son nom !

2 Tableaux et pointeurs

2.1 Le type tableau n'existe pas

Nous avons vu qu'il est possible de relier des variables de même type en un tableau, mais ce tableau ne constitue pas un type reconnu par le langage C. Prenons l'exemple d'un tableau de 12 entiers et d'un pointeur d'entier :

```
int Salaires [12];      /* indices allant de 0 a 11 */
int Mois;
int * pActif;          /* definit un pointeur sur int */

pActif = &Salaires[0];
```

`pActif` pointe sur le premier élément du tableau `Salaires[]`. Cette écriture est autorisée car `pActif` est l'adresse de tout objet de type `int` et que `Salaires[0]` est une variable de type `int`. Le nom d'un tableau est équivalent à l'adresse de son premier élément, c'est-à-dire :

```
&Salaires [0]  est identique a   Salaires
```

On peut donc également écrire :

```
pActif = Salaires;
```

Cette notation incite le programmeur débutant à penser que `Salaires` représente un tableau. On dit abusivement que `pActif` pointe sur un tableau. Mais le type tableau n'existe pas en langage C, car on ne peut jamais manipuler globalement un tableau, affecter directement un tableau à un autre tableau, etc. Pour le C, `Salaires` est l'adresse d'un entier, ce n'est donc pas une variable. C'est le programmeur qui sait que cette adresse est suivie en mémoire par une réservation de 11 autres entiers, sans nom. Donc, plutôt que de dire "pointe sur un tableau", utilisez toujours "pointe sur le premier élément de ce que je sais être un tableau de 12 éléments" (en espérant que vous ne vous trompez pas !). Alors :

```
Mois = *pActif;
```

affecte à `Mois` le contenu du premier élément de tableau, c'est-à-dire est équivalent à :

```
Mois = Salaires [0];
```

2.2 Algèbre des déplacements

Par définition, si `pActif` pointe sur le premier élément de `Salaires[]`, alors :

```
pActif + 4
```

pointe sur le 5^e élément de `Salaires`. Donc :

```
*(pActif + 4)
```

est ici identique à :

```
Salaires [4]
```

En fait, toutes les écritures sont permises :

```
Salaires [4]
*(Salaires + 4)         /* a eviter */
pActif [4]              /* a eviter */
*(pActif + 4)
```

représentent tous la même chose ! La notation indexée par des crochets ou indirecte par l'astérisque peut donc aussi bien être utilisée avec un pointeur qu'avec le nom d'un tableau. Cependant, les notations d'un pointeur scalaire indexé par des crochets et celle du nom d'un tableau précédé de l'indirection sont à proscrire, ne serait-ce que par un souci de lisibilité. En effet :

```
pActif [4]
```

n'incite pas à croire que `pActif` est une variable scalaire. De même, l'indirection devrait être réservée à une variable pointeur, et pas appliquée à un tableau. Ainsi :

```
&Salaires[Indice]  et  Salaires + Indice
```

représentent tous deux l'adresse de l'élément `Indice + 1`. N'oublions pas qu'une telle adresse est une constante.

2.3 Nom de tableau et pointeur

Il existe une différence fondamentale entre un nom de tableau et un pointeur, même s'il semble qu'on les utilise de la même manière :

— un pointeur est une variable,

— le nom d'un tableau est une constante - l'adresse d'implantation en mémoire.

Ainsi, à partir des exemples précédents, toutes ces opérations sont autorisées :

```
pActif++;               /* pointe sur l'entier suivant */
pActif = pActif - 7;    /* le pointeur recule de 7 entiers */
pActif += 3;            /* puis il avance de 3 entiers */
```

alors qu'aucune de ces opérations n'est permise si l'on remplace `pActif` par `Salaires`.

2.4 l-value

On appelle `l-value` (abréviation de `left-value`) l'expression que l'on trouve (ou que l'on pourrait trouver) à gauche d'un signe d'affectation. Ainsi dans :

```
pActif = Salaires;
*(pActif + 4) = *(pActif + 6);
```

`pActif` et `*(pActif + 4)` sont respectivement les l-values de la première et seconde instruction. Remarquez qu'à l'inverse de bien des langages, une opération de somme peut apparaître en langage C à gauche d'un signe d'affectation. Cette somme correspond ici en fait à un calcul d'adresse avant indirection. Pour le compilateur, une l-value est toujours une variable, c'est-à-dire qu'il est capable de lui affecter une valeur, celle de la `r-value` (`right-value`, l'expression à droite du symbole d'affectation). Ainsi `Salaires`, qui est une constante, n'est donc pas une l-value. On n'a donc pas le droit d'écrire :

```
Salaires = 132;     /* INCORRECT ! * /
Salaires++;         /* ++ ne peut agir que sur une l-value */
```

Une autre écriture incorrecte :

```
pActif = &Salaires;
```

car `Salaires` est une constante qui représente déjà une adresse, pas un tableau. En revanche, voici quelques expressions correctes :

```
Mois = Salaires [7];
pActif = Salaires + 5;
Salaires[0]++;
--Salaires[3];
```

En conclusion, souvenons-nous que seul un pointeur peut recevoir une adresse. Si l'on travaille dans un petit système qui n'utilise pas la mémoire virtuelle, on peut affecter à un pointeur la valeur d'une adresse physique connue. Par exemple, la table des vecteurs d'interruption du microprocesseur 68000 commence à l'adresse 0. Si nous disons, dans un premier temps, que cette table contient des entiers, nous pouvons écrire :

```
int * TabVectInt;           /* Definit le pointeur */
TabVectInt = (int *) 0;     /* Lui attribue une valeur */
```

ou bien

```
/* Definition et initialisation */
int * TabVectInt = (int *) 0;
```

Remarquez que le forçage de type est indispensable, car un pointeur ne peut recevoir un entier. Notez aussi que la table des vecteurs d'interruptions ne contient pas des entiers, mais des adresses de fonctions [d'interruption] qui ne renvoient rien !

3 Opérateurs et pointeurs

Voici les seules opérations possibles sur un pointeur :

- l'affectation d'une adresse d'un objet de même type que le type pointé par le pointeur (il n'y a aucune conversion implicite de type),
- les six opérations relationnelles, qui servent à des comparaisons d'adresses,
- l'incrémentation et la décrémentation automatiques, généralement utilisées pour parcourir un tableau,
- l'addition et la soustraction d'un entier (variable de type entier ou constante explicite entière), pour le calcul d'adresse comme on l'a déjà vu,
- la soustraction de deux pointeurs de même type, pour déterminer un déplacement.

Les autres opérations sont interdites ou dénuées de sens. L'incrémentation, la décrémentation d'un pointeur et l'addition ou la soustraction d'un entier sont toujours interprétés en fonction de la taille de l'objet pointé, qui est déterminée lors de la définition du pointeur. Voyons donc comment travaille le compilateur quand on écrit :

```
int Jours [31];
pJournee = Jours;
pJournee += 7;
```

Pour déterminer la nouvelle valeur de `pJournee`, comme 7 a la dimension d'un indice, le compilateur doit le transformer en déplacement en calculant le nombre d'octets correspondant à 7 éléments. Il fait ce travail automatiquement puisqu'il connaît, par le type du pointeur, la taille de l'élément. Remarquez qu'en assembleur, vous auriez dû vous-même calculer ces déplacements. Rappelons que le C ne vérifie pas les indices, donc il ne vérifie pas non plus les déplacements. Écrire :

```
pJournee + 50
```

n'entraîne aucune erreur détectée à la compilation ; par contre, durant l'exécution, on risque d'avoir de mauvaises surprises si l'on altère l'entier à l'adresse `pJournee + 50`. Avec beaucoup de chance, cette adresse sera hors de la zone de mémoire allouée par l'éditeur de liens et le programme s'arrêtera sur une erreur de segmentation, sinon, on ira altérer indirectement une autre grandeur, ce dont les conséquences pourront n'apparaître que tardivement. Remonter à l'origine de l'erreur se révèle alors difficile et l'on doit faire appel, en général, à un débogueur (programme de mise au point permettant essentiellement l'exécution du programme en pas à pas tout en observant la valeur des variables).

4 Passage des arguments par adresse

Au chapitre sur les fonctions, nous avions présenté rapidement l'appel de fonctions avec des arguments passés par adresse. C'est maintenant l'occasion d'y revenir.

4.1 Tableaux

Nous avons déjà vu que lorsque l'on fournissait le nom d'un tableau en argument, c'était en fait l'adresse du premier élément de ce tableau qui était rangée dans la pile associée au programme. Vu de l'intérieur de la fonction, cet argument est une variable qui contient une adresse : c'est donc un pointeur et on peut l'utiliser comme tel, même si nous vous avons recommandé de toujours considérer les paramètres reçus comme constants (voir paragraphe 4.1.3). Cela nous amène à conclure que le paramètre représentant l'adresse d'un tableau, que nous avons toujours considéré comme de type généralisé `char[]` pouvait indifféremment être déclaré de type `(char *)`, ce qui peut paraître confus, mais provient de la logique d'écriture disant qu'un pointeur peut indifféremment être utilisé avec l'indexation `[]` ou avec l'indirection `*`. Le compilateur (pour qui n'existe pas le type tableau) voit dans tous les cas que le paramètre passé est l'adresse d'un caractère, rien de plus. Seul le programmeur sait que cette adresse est suivie d'une réservation pour un tableau. Résumons ceci dans un exemple :

```
/* Definition globale d'un tableau a 5 elements,
   avec leur initialisation */

int Entiers [] = { 5, 2, 3, 4, 1 };
int printf ( char *, ...);          /* prototype de printf() */

/************** Definition de main() *****************/
int main(void)
{
  void Par_Nom   ( int [] );     /* prototypes locaux */
  void Par_Point ( int *  );

  Par_Nom ( Entiers );           /* appels */
  Par_Point ( Entiers );
  ...
}

/************** Definition de Par_Nom() **************/
void Par_Nom ( int Table [] )
{
  printf ("\nPar_Nom : %d\n", Table [2]);
}

/************** Definition de Par_Point() ************/
void Par_Point ( int * p_Sur_Int )
```

```
{
  p_Sur_Int += 2;
  printf ("Par_Point : %d\n", *p_Sur_Int);
}
```

Ces deux fonctions sont équivalentes, excepté le message affiché. L'une et l'autre utilisent le troisième élément du tableau Entiers. Voici le résultat à l'exécution :

```
Par_Nom : 3
Par_Point : 3
```

Le programme principal fournit le même paramètre : le nom du tableau. La première fonction l'utilise comme un nom de tableau, et la seconde, comme un pointeur. En conclusion, les types généralisés `(type*)` et `(type[])` sont équivalents, en tant que paramètres formels. Voici donc une nouvelle version de `Par_Nom()` :

```
void Par_Nom ( int Table [])
{
  int * pEntier = Table;

  pEntier += 2;
  printf ("\nPar_Nom : %d\n", *pEntier);
}
```

4.2 Variables scalaires

Le passage par adresse d'une variable scalaire en tant qu'argument est tout à fait analogue au cas des tableaux, si ce n'est que l'adresse doit être passée explicitement à l'appel grâce à l'opérateur unaire `&`. Puisque la fonction a reçu l'adresse d'une variable, elle considère son paramètre comme un pointeur sur cette variable. Celle-ci peut donc être atteinte indirectement et altérée par la fonction. Prenons un exemple simple :

```
/*************** Definition de main() ***************/
int main(void)
{
  void Par_Ref ( double *);                  /* prototype */
  double Nb_Reel = 3.5;

  printf ("Avant appel de la fonction, Nb_Reel vaut %8.3lf\n",
          Nb_Reel);
  Par_Ref ( & Nb_Reel );                     /* par adresse */
  printf ("Apres appel de la fonction, Nb_Reel vaut %8.3lf\n",
          Nb_Reel);
  return 0;
}

/************* Definition de Par_Ref() ***************/
void Par_Ref ( double * ref_Double )
{
  *ref_Double += 2;     /* conversion automatique de type */
}
```

À l'exécution, le programme fournira :

```
Avant appel de la fonction, Nb_Reel vaut   3.500
Apres appel de la fonction, Nb_Reel vaut   5.500
```

Ainsi, le passage par référence permet-il d'avoir des paramètres de sortie. Nous avons déjà utilisé cela quand nous avons employé la fonction `scanf()`. Voici un appel de `scanf()` qui lit un entier au clavier et l'affecte à la variable `Prix`. Cette variable doit être passée par référence, sinon `scanf()` ne pourrait jamais en modifier la valeur :

```
int Prix = 0;

scanf ( "%d", & Prix );
Si on avait \'ecrit ? :
scanf ( "%d", Prix ) ;              /* Horribile visu ! */
```

On aurait sans doute eu une erreur à l'exécution avec le message : `segmentation fault`.

En effet, `scanf()` considère que ce qu'il reçoit est l'adresse d'un entier : il tente donc d'écrire à l'adresse 0, ce qui lui est normalement interdit (sous UNIX du moins). Bien sûr, tout ceci découle de ce que le C ne peut vérifier le paramètre passé à `scanf()` car ce type n'est pas fixe. Par exemple, voici un appel de `scanf()` où le second argument n'est plus un `int *` mais un `char *`. Il s'agit de la saisie au clavier d'une chaîne (saisie terminée par un caractère `'\r'`) :

```
char Ligne [80];

scanf ( "%s", Ligne );
```

Ici aussi, ce passage par référence d'un tableau est fait avec le nom du tableau. Remarquez encore une fois que rien ne protège le programme d'un utilisateur qui fournit au clavier une chaîne plus longue que 79 caractères - 79 car il faut compter aussi le caractère nul, automatiquement rajouté en fin de saisie par `scanf()`.

4.3 Sachons bien lire un paramètre passé par adresse

Quand on a l'en-tête :

```
void Par_Ref (double * ref_Double )
                3    2     1
```

il faut lire : `ref_Double (1)` est l'adresse `(2)` d'un double `(3)`.

Un autre exemple :

```
int Autre_fcn ( char ** ref_Point)
                  4  32     1
```

Le paramètre `ref_Point (1)` est l'adresse `(2)` d'un `(char *) (4 et 3)`. C'est donc l'adresse d'un pointeur de caractère ou, si vous préférez, il s'agit d'un pointeur passé par référence. Comme pour les tableaux, au lieu de déclarer :

```
void Par_Ref ( double * ref_Double )
```

on pouvait aussi écrire :

```
void Par_Ref ( double ref_Double [] )
```

5 Mécanismes d'évaluation des opérateurs unaires ∗ ++ −−

Dans cette partie, nous allons détailler l'utilisation de ces opérateurs avec un pointeur que nous désignerons par `Ptr`.

5.1 Opérations unaires sur des pointeurs

Les opérateurs unaires `* & ++ --` sont de même niveau de priorité (2) et l'évaluation est faite de droite à gauche. L'indirection `*` et l'adresse `&` ne peuvent que préfixer le pointeur. Par contre, incrémentation et décrémentation peuvent précéder ou suivre le pointeur. Nous avons donc les possibilités suivantes (en ne considérant que `++` ; pour `--`, c'est similaire) :

```
1) X = *++Ptr;          4) X = *(++Ptr);
2) X = *Ptr++;          5) X = *(Ptr++);
3) X = ++*Ptr;          6) X = ++(*Ptr);
                        7) X = (*Ptr)++;
```

Avec les règles d'évaluation, nous voyons que :

— Les cas 1 et 4 sont équivalents : le pointeur `Ptr` est d'abord incrémenté, il pointe alors sur l'élément suivant. `X` reçoit donc la valeur de l'élément suivant.

— Les cas 2 et 5 sont aussi équivalents : l'incrémentation de `Ptr` ne survient qu'après l'indirection. `X` reçoit la valeur de l'élément courant, puis le pointeur est incrémenté.

N.B. : ceci semble contredire la règle d'associativité telle qu'elle apparaît dans le tableau 5.1, mais il faut se souvenir que lorsque l'opérateur `++` (ou `--`) apparaît après son opérande, le résultat immédiat de l'expression est la valeur de l'opérande avant l'incrémentation (ou décrémentation). Une fois le résultat utilisé, donc attribué ici à `X`, l'opérande est incrémenté (ou décrémenté).

— Les cas 3 et 6 sont identiques : on incrémente ce qui est pointé et cette nouvelle valeur va en `X`. Le pointeur est inchangé.

— Le cas 7 est unique : les parenthèses sont indispensables. La valeur de ce qui est pointé est mise en `X`, puis ce qui est pointé est incrémenté.

Les 7 cas présentés se réduisent ainsi aux 4 cas suivants :

1. Pré-incrémenter le pointeur, puis pointer. Ce qui est pointé est inchangé.
2. Pointer, puis post-incrémenter le pointeur. Ce qui est pointé est inchangé.
3. Pré-incrémenter ce qui est pointé. Le pointeur est inchangé.
4. Post-incrémenter ce qui est pointé. Le pointeur est inchangé.

Tous ces mécanismes sont d'utilisation courante. En cas de doute, n'hésitez pas à utiliser des parenthèses ou à décomposer les expressions.

5.2 Exemples

Dans les exemples suivants, notez la similitude avec l'assembleur, surtout par l'utilisation naturelle de décompteurs, de post-incrémentation et de pré-décrémentation.

1. Voici une version de la fonction `strlen()`, qui renvoie la taille d'une chaîne de caractères. Le caractère nul final n'est pas compté :

```
int Longueur_Chn ( char * Chaine )
{
  char * Courant = Chaine;
  int Compte = 0;

  while ( *Courant++ )
    Compte++;
  return Compte;
}
```

2. Transfert d'une matrice rectangulaire réelle de taille `n x m` d'un tableau `Orig` vers un tableau `Destin`. On fournit l'adresse `Orig` de la matrice en paramètre et on la transfère sous forme linéarisée vers la matrice `Destin` dont l'adresse est aussi passée en paramètre.

```
void Copie_Mat ( double * Orig, double * Destin,
                 int N_Col, int M_Lign )
{
  int Nb_Elts;

  Nb_Elts = N_Col * M_Lign;
  while ( Nb_Elts-- )
    *Destin++ = *Orig++;
}
```

3. Incrémenter tous les éléments d'un tableau d'entiers de taille connue, à l'exception des premier et dernier éléments.

```
void Incr_Tab ( int * Vecteur, int Taille )
{
  int Rebours;

  Rebours = Taille - 2;
  while ( Rebours-- )
  {
    Vecteur++;
    (*Vecteur)++;
  }
}
```

   Dans cet exemple, le corps du `while` peut être aussi remplacé par une seule instruction (mais le programme n'est pas exécuté plus vite pour autant !) :

```
(*++Vecteur)++;
```

4. Calcul de la trace d'une matrice réelle triangulaire supérieure d'ordre donné.

```
double Trace_TSup ( double * Triang_Sup, int Ordre)
{
  double Trace = 0.0, * Courant = Triang_Sup;

  do
  {
    Trace += *Courant;
    Courant += Ordre;
  }
  while ( --Ordre );
  return Trace;
}
```

5. Exemple utilisant la comparaison de pointeurs. Il s'agit d'une fonction testant si une chaîne est un palindrome :

```
/* Verifie qu'une chaine, dont l'adresse du premier
   element est passee en argument, est un palindrome.
    Retourne 0 si faux, 1 si vrai.
*/

int Palin ( char Chaine [] )
{
  char * Debut, * Fin;
```

```
    int Retour = 1;

    Debut = Chaine; /* pointe sur le premier caractere */
    Fin = Chaine + strlen ( Chaine ) - 1;

    /* Fin pointe sur le dernier caractere car strlen()
       retourne le nombre de caracteres de la chaine,
       compte non tenu du caractere nul final */

    while (Debut < Fin && Retour) /* compare des ptrs */
    if ( *Debut++ != *Fin-- )     /* compare des car. */
      Retour = 0;
    return  Retour;
  }
```

Ces exemples permettront au lecteur de remarquer la concision du langage.

6 Indirections multiples

Considérons les définitions suivantes :

```
int Entier, * p_Entier;
int * * pp_Entier;
  3   2    1
```

en effectuant la lecture de droite à gauche comme nous l'avons déjà vu (associativité de l'opérateur unaire `*`), la variable `pp_Entier (1)` est un pointeur `(2)` sur un `(int *) (3)`, c'est-à-dire sur un pointeur d'entier. Il s'agit donc d'un pointeur de pointeur. En assembleur, nous parlerions de double indirection. Pour relier ces trois variables, nous devrons écrire, par exemple :

```
Entier = -11734;
p_Entier = &Entier;
pp_Entier = &p_Entier;
```

ce que nous pouvons représenter symboliquement par ce schéma (figure 2) :

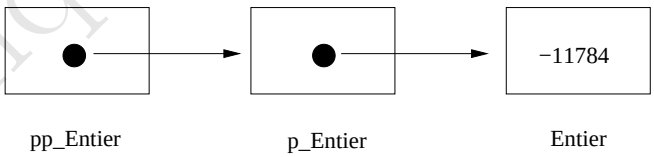

FIGURE 2 – Double indirection

On a évidemment accès à `Entier` par indirection et par double indirection :
`Entier` identique à `* p_Entier` identique à `** pp_Entier`

On peut continuer ainsi et, par exemple, avoir le schéma suivant (figure 3) :

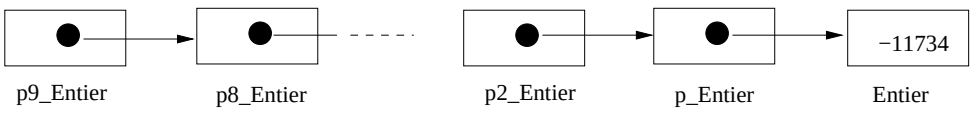

FIGURE 3 – Indirections multiples

avec ainsi :

```
int ********* p9_Entier;
```

Le type de base de tous ces pointeurs est `int`. Seul le nombre de modificateurs est différent. Mais on ne peut évidemment pas écrire pour autant :

```
p9_Entier = &Entier;             /* INTERDIT ! */
p9_Entier = &p7_Entier;          /* AUSSI !  */
```

Le compilateur décèle l'erreur et envoie un message du type : "illegal pointer combination". En effet, un pointeur n'est pas qu'une adresse. Un pointeur possède un certain nombre d'attributs, parmi lesquels il y en a un qui fixe la profondeur d'indirection :

- profondeur d'indirection,
- taille de l'objet pointé,
- classe,
- type de base,
- nom.

Les trois derniers attributs sont explicites, c'est-à-dire conservés dans une table par le compilateur durant son travail ; en revanche, les deux premiers sont implicites, car ils font partie de la sémantique du langage. Sous UNIX, il existe des attributs supplémentaires pour protéger le système contre tout pointeur abusif appelé pointeur Cheval-de-Troie (Trojan Horse). En effet, il ne faut pas que n'importe quel utilisateur puisse modifier à coup d'indirections folles des tables gérées par le système d'exploitation. En théorie, il n'y a pas de limite à la profondeur d'indirection. Comme toujours, on est en fait limité par la mémoire dont on dispose.

Plutôt que de traîner un grand nombre d'astérisques, avec le risque d'en oublier un en cours de route, on préfère souvent redéfinir des types plus lisibles par `typedef`. Par exemple, voici la définition d'un nouveau type, `p9_Sur_Int` :

```
typedef int ********* p9_Sur_Int;
```

alors la définition d'une nouvelle variable `Tres_Indirecte` s'écrit :

```
p9_Sur_Int Tres_Indirecte;
```

qui est équivalente à :

```
int ********* Tres_Indirecte;
```

7 Les tableaux de pointeurs

7.1 Tableaux de pointeurs et tableaux à deux dimensions

Puisque les pointeurs sont des variables, on peut facilement créer des tableaux dont les éléments sont des pointeurs :

```
char * Pointeurs [10];
 4  3     1      2
int * Mois [12];
```

La lecture du type exige d'utiliser les règles de préséance déjà vues au début du chapitre ??. Les crochets (priorité 1) sont prioritaires par rapport à l'indirection (priorité 2). Le type de base est toujours lu en dernier, et on commence toujours par le nom de l'objet défini. Ainsi, `Pointeurs (1)` est l'adresse d'un tableau `(2)` de 10 pointeurs `(3)` sur un caractère `(4)`. Mais pointer sur un caractère n'est pas très utile. Il est donc très possible que ces pointeurs conservent chacun l'adresse d'une chaîne de caractères, mais la syntaxe ne le dit pas. Une fois de plus, et comme en assembleur, c'est au programmeur de savoir ce qui a réellement été réservé. De même, `Mois` est le nom d'un tableau de 12 pointeurs d'entier. Ici aussi, il est possible que l'adresse conservée dans chaque pointeur soit celle d'un tableau d'entiers. Si tous ces tableaux d'entiers ont la même dimension (mais ce n'est pas obligatoire), on a alors une structure de tableau à deux dimensions. On comprend donc aisément que derrière cette notation peuvent se profiler aussi des tableaux à deux dimensions. De la même manière qu'il existait une équivalence entre

```
char *      et     char []
```

il en existe une entre

```
char [][]   ,   char *[]   ,   char []*   et   char **
```

7.2 Tableau de chaînes de caractères

Un tel tableau est intéressant à étudier pour bien montrer l'intérêt de la notation adoptée. Supposons que nous voulions avoir la configuration de la figure 4 lors de la définition (le losange noir représente le caractère nul) :

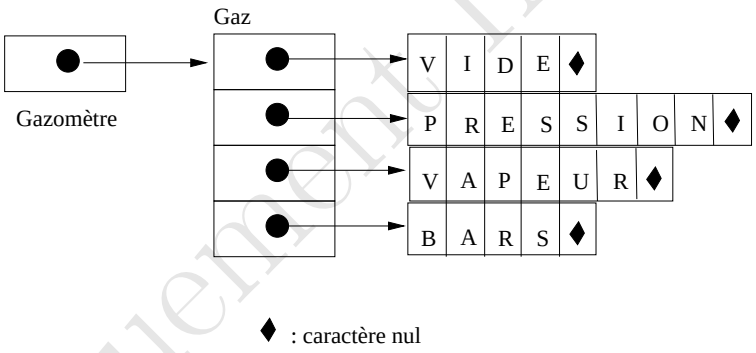

FIGURE 4 – Tableau de chaînes de caractères

Voici comment peut se faire une initialisation d'un tel tableau :

```
char * Gaz [] = { "VIDE", "PRESSION", "VAPEUR", "BARS" };
char ** Gazometre = Gaz;
```

Le compilateur réserve la place pour 4 tableaux de caractères et y place les 4 chaînes. Par exemple, le premier tableau contient `"Vide"` : il est donc de longueur 5. Chaque tableau est automatiquement ajusté à la bonne longueur. Ces tableaux n'ont pas de nom mais on connaît leur adresse car elle est placée dans le tableau `Gaz[]`. `Gaz[0]` est l'adresse du premier tableau qui contient `''Vide''`. `*Gaz [0]` est une variable de type `char` contenant la lettre `'V'`. `Gaz[0][2]` ou `*(Gaz[0] + 2)` est une variable de même type contenant la lettre `'d'`. Comme lors de la définition, on n'a pas placé de valeur entre les crochets, le compilateur ajuste automatiquement la taille du tableau `Gaz` à 4, car on a défini 4 adresses. `Gazometre` et `Gaz` sont de types généralisés analogues puisque : `char **` est équivalent à `char * []`. La seule différence, comme déjà dit, est que `Gaz` est une constante, représentant l'adresse d'un tableau de pointeurs, alors que `Gazometre` est une variable qui peut recevoir cette adresse, entre autres. Attention ! Dans l'exemple donné, si on modifie la valeur d'un élément du tableau de pointeurs, on perd inéluctablement l'accès à la chaîne sur laquelle il pointait, à moins d'avoir eu la précaution de recopier auparavant son adresse dans un pointeur libre. Ainsi, si on écrit simplement :

```
Gaz [0] = Gaz [1];
```

les deux premiers éléments du tableau contiennent la même adresse, et pointent sur la même chaîne `"Pression"`. La chaîne `"Vide"` n'est plus accessible. Ici aussi, l'utilisation de `typedef` peut alléger l'écriture en redéfinissant le type généralisé `(char *)` ainsi :

```
typedef char * Pointeur;
Pointeur Gaz [4];          /* un tableau de 4 pointeurs */
Pointeur * Gazometre = Gaz;
```

7.3 Tableau à deux dimensions

Dans l'exemple précédent, nous aurions aussi explicitement pu utiliser un tableau à deux dimensions, mais l'occupation en mémoire aurait été différente, comme le montre la figure 5.

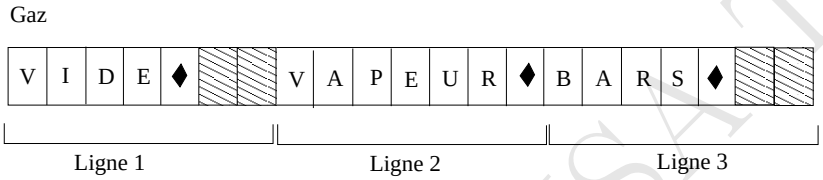

FIGURE 5 – Un tableau à deux dimensions

Voici comment se fait l'initialisation d'un tel tableau :

```
char Gaz [][7] = {"Vide", "Vapeur", "Bars"};
```

Remarquez qu'il a fallu au moins donner la taille de la seconde dimension, car le compilateur ne travaille qu'en une passe. Il est donc incapable de majorer correctement la seconde dimension. À la différence du tableau de pointeurs vu au 7.2, on note un gaspillage de la place réservée à des tableaux incomplètement remplis par les chaînes. L'utilisation en est donc différente. Si les chaînes sont invariables, l'écriture utilisée en 7.2 est bien préférable. En annexe **??**, vous trouverez une présentation générale des tableaux à `n` dimensions.

8 Allocation et libération de mémoire dynamique

Un programme écrit en langage C peut gérer, durant son exécution, des allocations et des libérations dynamiques de la mémoire.

Pour l'allocation, on utilise principalement les fonctions `malloc()` et `calloc()`. Il est nécessaire d'inclure la fichier `stdlib.h` pour utiliser ces fonctions à l'aide de la directive `#include <stdlib.h>`.

```
void * malloc (int Taille);
```

À l'exécution d'un appel de cette fonction, le système d'exploitation attribue `Taille` octets contigus (par l'algorithme du premier accord, first fit) et la fonction renvoie l'adresse du premier octet. C'est donc un tableau d'octets qui est alloué, et `malloc()` en retourne l'adresse. A priori, ce tableau n'est pas initialisé. Un pointeur sur `void` veut simplement signifier un pointeur sur n'importe quel type, mais l'organisation demeure en fait une organisation d'octets.

```
void * calloc (int Nbre, int Taille);
```

Cette fonction réserve en mémoire un tableau de `Nbre` éléments de `Taille` octets chacun. L'adresse renvoyée est celle du tableau, mais, encore une fois, considérée comme un tableau de `Nbre x Taille` octets ! Tous les éléments sont initialisés à zéro. En général, la zone allouée par une de ces deux fonctions est utilisée à autre chose qu'un tableau de caractères ou à une chaîne, aussi est-il nécessaire d'avoir recours à l'opérateur de coercition (forçage de type). Supposez que nous voulions obtenir la place pour un tableau de 40 entiers, on peut écrire :

```
int * Table_Dyn;

Table_Dyn = (int *) malloc ( 40 * sizeof (int) );
```

Le pointeur `Table_Dyn` recueille l'adresse du tableau. L'utilisation de l'opérateur `sizeof` appliqué au type `int` permet d'obtenir un programme portable. On peut alors attribuer une valeur au 13^e élément, par exemple :

```
*(Table_Dyn + 12) = -23456;
```

Pour libérer une zone de mémoire pointée par `pZone`, il suffit d'écrire :

```
free (pZone);
```

Le système d'allocation n'alloue pas la mémoire à l'octet près, car la gestion en serait bien trop lourde. La zone réellement réservée contient toujours un multiple de la plus petite taille directement allouable (par exemple 4 octets). Chaque bloc alloué est précédé d'un en-tête (4 octets pour UNIX) contenant la taille du bloc, si bien que pour le libérer, la fonction `free()` n'a besoin que de l'adresse du bloc. Pour diminuer la fragmentation de la mémoire lors de la libération d'un bloc, le système de gestion de la mémoire dynamique effectue le regroupement des zones libres adjacentes pour former des zones libres de la plus grande taille possible.