Petit rappel sur les pointeurs

Version 1
29 mai 2009

Axel Berardino <axel.berardino@gmail.com>

Table des matières

1 Préface 1

2 Introduction 2

3 Déclaration de type pointeur simple. 3

4 Exemple récapitulatif commenté: 4

5 Les erreurs classiques. 5

6 Déclaration de type pointeur plus évoluée. 7

7 Le pointeur neutre. 8

8 Les pointeurs sur fonctions/procédures. 9

9 Les listes chainées. 10

10 Manipulation des listes chainées: 11

10.1 Accéder à l'élement n: 11
10.2 Insertion d'élément 12
10.3 Suppression d'élément 13

1 Préface

Auteur : Axel BERARDINO
<axel.berardino@gmail.com>

2 Introduction

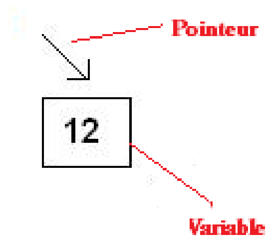
Les pointeurs sont très utiles. Ils permettent d'accélerer ou d'alléger certains processus.

Comme vous le savez, à chaque fois que vous déclarez une variable, une taille est automatiquement allouée dans la mémoire. Chaque variable peut être considérée comme une "boite" qui contient l'information que l'on a stocké. Ces petites "boites" sont placées quelquepart dans la mémoire, elles ont une adresses.

Le principe du pointeur c'est de gérer soit même la mémoire, au lieu de laisser le compilateur le faire. Lorsque vous déclariez votre variable, vous mettiez votre information dans une "boite" sans vous souciez ou elle se trouvait dans la mémoire. Cette fois, vous allez gérer son emplacement et sa taille.

Un pointeur est une variable spéciale qui contient une adresse. On pourrait comparer un pointeur à un raccourci. Imaginez que vous ayez une liste de 40 divx, faisant 1 go chacun, que vous voudrier trier. La première solution consiste à trier naturellement ces 40 fichiers, ce qui mettrait un temps fou. La deuxième serait de créer un raccourci pour chaque divx, et de trier, non pas les divx directement, mais les raccourcis de ceux-ci. Ainsi en très peu de temps, on a trié notre liste (même si on n'a pas trié directement la liste, le résultat est le même).

On pourrait donc voir les pointeurs comme des "flèches" qui pointent sur des "boites".



3 Déclaration de type pointeur simple.

Pour déclarer un pointeur rien de plus simple. Il suffit de mettre une étoile après le type pointé.

```
int* a;
mon_type* b;
etc...
```

Ce type de pointeur ne peux pointer que devant le type qui lui est associé. Lorsque vous allez créer votre pointeur, celui-ci ne pointera sur rien. Vous devez donc lui dire sur quoi pointer.

```
Exemple:

int main(void)
{
  int* chiffre1;
  int  i;

  i = 12;
  chiffre1 = NULL; // ne pointe sur rien
  chiffre1 = &i;
  printf("%i\n", *chiffre1); // Affichera 12
  printf("%i\n", i); // Affichera 12
}
```

Analysons l'exemple précédent. On commence par déclarer un pointeur sur entier (chiffre1) et un entier (i). Puis on attribue la valeur de 12 à i. chiffre1 ne pointe au début sur rien (NULL). Puis on récupere l'adresse de i, grâce à l'opérateur "&". Maintenant notre "flèche" chiffre1 pointe sur la "boite" i. Enfin, on aimerait se servir du joli pointeur que l'on a crée, alors on va afficher la valeur de i, indirectement. On déréférence le pointeur, c'est à dire que le pointeur va se comporter comme s'il était la "boite" pointée (avec le signe "*"). A présent, on sait utiliser les pointeurs comme raccourci. Maintenant, on va voir comment allouer de la mémoire, c'est à dire se servir des pointeurs sans utiliser de variables déjà existantes. Vous allez me dire, s'il n'y a pas de variable à pointer, comment faire ? Et bien c'est simple, on peut utiliser un pointeur pour créer une "boite" dans la mémoire. Pour allouer de la mémoire plusieurs solutions. On utilise la fonction "malloc". Vous devez alors indiquer la taille de la "boite".

```
Exemple: chiffre1 = malloc(50);
```

Voila vous venez de créer un espace pour mettre une valeur. Maintenant, on sait que toute les variables ne prennent pas toute la même place en mémoire. On va donc allouer de la mémoire en fonction de la taille de la variable.

```
Exemple: chiffre1 = malloc(sizeof (int));
```

Voila, on est sur d'occuper exactement la taille nécessaire.

Notre "boite" est crée. Il ne reste qu'à la remplir. Pour cela, on déréference le pointeur, et on fait comme si le pointeur était une variable normale.

```
Exemple: *chiffre1 = 52;
```

Il reste une notion importante, si on alloue un espace mémoire, il faut **ABSOLUMENT** le libérer quand on a finit de s'en servir. On utilise pour cela la fonction "free".

```
Exemple: free(chiffre1);
```

4 Exemple récapitulatif commenté:

```
int main(void)
{
  int* chiffre1;
  float* chiffre2;
  int i;

  chiffre1 = malloc(sizeof (int));// Allocation mémoire
  chiffre2 = malloc(sizeof (float));

  i = 7;
  *chiffre1 = 5; // On place 5 dans la "boite" que pointe chiffre1
  printf("%i\n", *chiffre1);// Affichera 5
  *chiffre1 = i; // On place l valeur de i dans la "boite" que pointe chiffre1
  printf("%i\n", *chiffre1);// Affichera 7
  *chiffre2 = i; // On place l valeur de i dans la "boite" que pointe chiffre2
  printf("%i\n", *chiffre2);// Affichera 7
  free(chiffre1);// Désallocation mémoire
  chiffre1 = chiffre2; // On demande a chiffre1 de pointer
                       //sur ce que pointe chiffre2
  printf("%i\n", *chiffre1);// Affichera 7

  chiffre1 = NULL;
  free(chiffre1);// Ici, chiffre1 vaut NULL, donc free ne fera rien
  free(chiffre2);// Désallocation mémoire
}
```

5 Les erreurs classiques.

Il y a certaines erreurs qui reviennent souvent. Par exemple, n'essayez jamais de désallouer un pointeur nul.

```
Exemple:

int main(void)
{
  int* chiffre2;
  int  i;

  chiffre2 = malloc(sizeof (int));// Allocation mémoire
  chiffre2 = NULL;// Ne pointe sur rien

  i = 7;
  printf("%i\n", i);// Affichera 7
  printf("%i\n", *chiffre2);// PLANTE
  free(chiffre2);// chiffre2 vaut NULL, donc ne fait rien
}
```

En passant, lorsque l'on a fait un "chiffre2 = NULL;" on a perdu dans la mémoire la "boite" que l'on pointait. Comme on a perdu l'adresse, on ne pourra plus jamais la retrouver. Une partie de la mémoire sera donc occupée pour rien. (Pas de panique, les "boites" perdues sont détruites au sortir de l'application).

Deuxieme erreur fréquente, n'oubliez pas d'allouer avant de manier un pointeur.

```
Exemple:

int main(void)
{
  int* chiffre2;
  int  i;

  i = 7;
  printf("%i\n", i);// Affichera 7
  printf("%i\n", *chiffre2);// PLANTE
  free(chiffre2);// chiffre2 vaut une valeur aléatoire, Donc ca plante !
}
```

Maintenant vous allez me dire, oui mais dans ton premier exemple tu n'alloues pas.

```
Rappel (1er exemple):

int main(void)
{
  int* chiffre1;
  int  i;

  i = 12;
  chiffre1 = NULL; // Ne pointe sur rien
  chiffre1 = &i;
  printf("%i\n", *chiffre1); // Affichera 12
  printf("%i\n", i); // Affichera 12
}
```

Ce n'est pas tout à fait pareil, dans mon premier exemple, je n'ai rien à allouer parce que les variables existent déjà. Ici, la variable i existe, je veux stocker dans la "boite" i, je n'ai donc pas besoin de créer une nouvelle "boite".

Enfin dernière erreur (la plus fréquente), n'oubliez pas de désallouer un pointeur.

```
Exemple:

int main(void)
{
  int* chiffre1 = NULL;
  int  i;

  i = 12;
  chiffre1 = malloc(sizeof (int));
  chiffre1 = &i;
  printf("%i\n", *chiffre1); // Affichera 12
  printf("%i\n", i); // Affichera 12
  // Ici, un free est manquant !
}
```

Le pire dans cette erreur, c'est que ca ne fait pas planter l'application. Mais votre programme "fuira". C'est à dire qu'il pompera de plus en plus de mémoire pour rien.

6 Déclaration de type pointeur plus évoluée.

On peut bien évidemment pointer sur des structures plus évoluées (Tableau, enregistrement, classe, etc..).

```
Exemple:
struct  s_list
{
  int   a;
  float b;
};

struct s_list* p_ma_structure; // Pointeur sur un struct s_list*
```

Créer un tableau de pointeur:

```
int** tab;// Tableau de pointeur pointant sur des entiers
```

Ou même pointer sur un pointeur qui pointe sur un pointeur, etc...

```
int******* ptr;

******ptr = malloc(sizeof (int));
*******ptr = 5;
free(******ptr);
```

ATTENTION: ne pas confondre "pointeur sur tableau" et "tableau de pointeur". Un pointeur sur tableau est un pointeur unique qui pointe sur un tableau. Il se déréférence comme ceci: (*tab)[0]

Un tableau de pointeur est un tableau qui contient des pointeurs. Il se déréférence comme ceci: *(tab[0]) On peut évidemmnent déclarer un type "Pointeur sur tableau de pointeur". Il se déférence comme ceci: *((*tab)[0])

7 Le pointeur neutre.

Dernier type de pointeur: le pointeur neutre. Ce type de pointeur, peut pointer sur tout. On le déclare comme ceci: void *ptr;

Toutefois, ce type de pointeur ne peut être déréférencé.

```
void* ptr;
int   i = 5;
int*  truc;
int*  tmp;

truc = &i;
ptr = truc; // Valide
*ptr = 5;   // Invalide
```

En effet, il faut "caster" ce type.

```
void* ptr;
int   i = 5;
int*  truc;
int*  tmp;

truc = &i;
ptr = truc; // Valide
tmp = ptr;
printf("%i\n", *tmp); // Affichera bien 5
```

8 Les pointeurs sur fonctions/procédures.

On peut pointer aussi sur des fonctions ou des procédures.

```
int (*pfunc)(void);// Pointe sur des fonctions qui ne prennent
                   //rien en paramètre mais retourne un type integer
void (*pfunc)(void);// Pointe sur des procédure qui ne prennent
                    // rien en paramètre
int (*pfunc)(char* s);// Pointe sur des fonctions qui prennent une
                      // chaine en paramètre et retourne un type integer
void (*pfunc)(int a);// Pointe sur des procedure qui prennent un
                     // entier en parametre
```

Pour utiliser nos procedures pointées, il faut faire ceci:

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

int dbl_strlen(char* s)
{
  return (2 * strlen(s));
}

int main(void)
{
  pfunc my_func;

  my_func = strlen;
  my_func("coucou"); // Renvoie 6
  my_func = dbl_strlen;
  my_func("test"); // Renvoie 8

  return (0);
}
```

Cela permet aussi d'avoir indirectement un tableau de procédures ou de fonctions.

```
pfunc* tab_func;
```

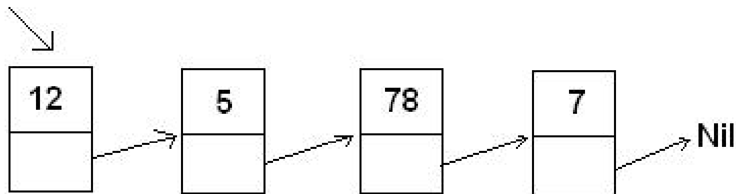
9 Les listes chainées.

Enfin, je terminerai sur les listes chainées. Une liste chainée est comparable à un tableau. Elle se construit avec un enregistrement. Le principe est celui ci: chaque "case" de ce "tableau" contient l'adresse de la prochaine "case". D'une certaine manière cela revient à gérer soit même son tableau.

On la déclare comme ceci:

```
typedef struct      s_list
{
   int              val; // Ici on prend un int, mais ca aurait pu
                       //   être n'importe quoi d'autre
   struct s_list*   suivant;
}                  s_list;
```

Pour savoir quand on atteint la fin du tableau on regarde si la variable "suivant" vaut "NULL". Il existe des variantes comme les listes doublement chainées (chaque case possède l'adresse de la case suivante et de la case précédente), les listes circulaires (la dernière case pointe sur la première), et les listes circulaires doublement chainées.



10 Manipulation des listes chainées:

10.1 Accéder à l'élement n:

Pour lire un élément, il suffit de le déréférencer. Mais avant il faut déjà placer le pointeur dessus.

1er élément: pointeur->val

2ème élément: pointeur->suivant->val

3ème élément: pointeur->suivant->suivant->val

Vous imaginez bien que si la liste comporte 120 éléments, on ne va pas écrire "suivant->suivant...". On va donc utiliser une boucle.

```
Exemple:

for (i = 0; i < 3; i++)
 pointeur = pointeur->suivant;

 // Accès 3ème élément
 pointeur->val;
```

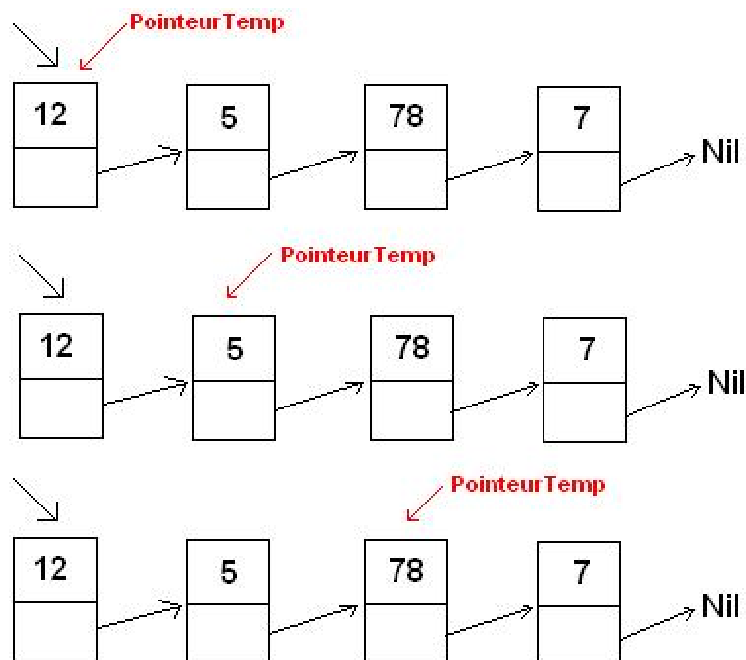
Ainsi on atteint le 3ème élément du tableau, en contrepartie, on perdu toute les informations précédentes (on ne peut plus retourner en arrière).

ATTENTION: le "danger" avec les listes chainées c'est de perdre la "tête", c'est à dire perdre les précédents éléments de la liste. Cela arrive fréquemment. Pour parcourir votre liste chainée, créez un pointeur temporaire.

Il faut donc toujours l'écrire comme ceci:

```
Exemple:

 s_list* tmpPointeur;
 tmpPointeur = pointeur;
 for (i = 0; i < 3; i++)
   tmpPointeur = tmpPointeur->suivant;

 // Accès 3ème élément
```

```
tmpPointeur->val;
```



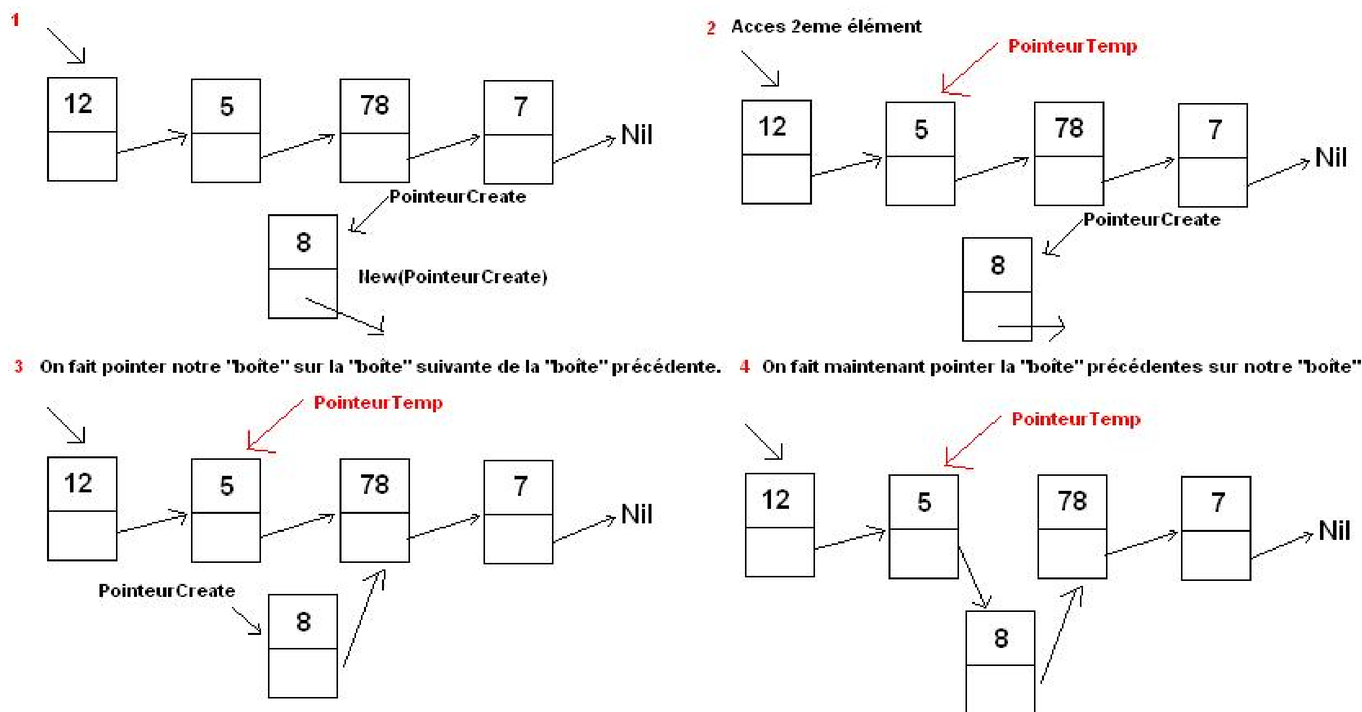
Là, c'est parfait, on accède au 3ème élément, sans perdre des informations.

10.2 Insertion d'élément

Pour insérer un élément le principe est le suivant: On créer une "boite" pointée par un pointeur temporaire. On fait pointer notre "boite" sur la "boite" suivante de la "boite" précédente. On fait maintenant pointer la "boite" précédente sur notre "boite". Et voila, on a ajouté notre élément. Vous comprenez maintenant l'utilité des listes chainées par rapport au tableau. Dans un tableau il aurait fallu décaler certains éléments vers la droite, en liste chainée, c'est immédiat.

```
Exemple d'ajout en 3ème position:

 s_list* tmpPointeur;
 s_list* pointeurCreation;

 pointeurCreation = malloc(sizeof (s_list));
 tmpPointeur = pointeur;
 // accès 2ème élément
 for (i = 0; i < 2; i++)
   tmpPointeur = tmpPointeur->suivant;
 // On fait pointer notre "boite" sur la "boite" suivante de la "boite" précédente.
 pointeurCreation->suivant = tmpPointeur->suivant;
 // On fait maintenant pointer la "boite" précédente sur notre "boite".
```

```
tmpPointeur->suivant = pointeurCreation;
```

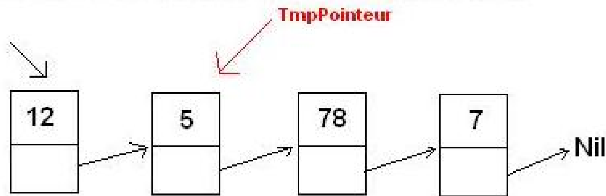


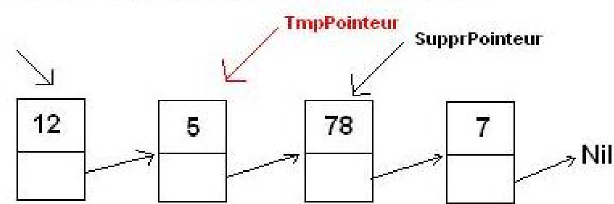
10.3 Suppression d'élément

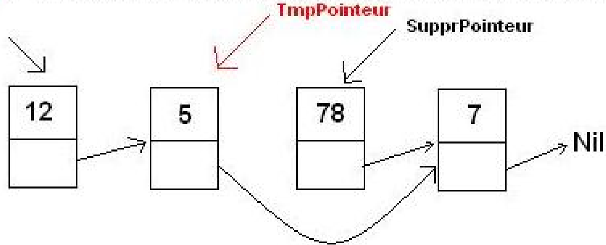
Pour supprimer un élément le principe est proche de l'insertion d'élément. On se place sur l'élément précédent l'élément à supprimer. Puis on pose un pointeur temporaire sur cet élément. On fait pointer l'élément précédent sur l'élément suivant de l'élément à suppprimer. Enfin, on détruit l'élément à supprimer.

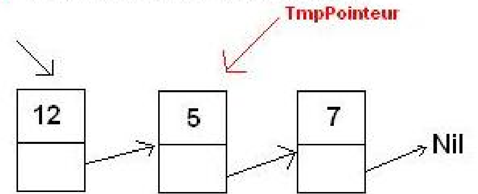
```
Exemple de suppression de la 3ème position:

 s_list* tmpPointeur;
 s_list* supprPointeur;

 tmpPointeur = pointeur;
 // On se place sur l'élément précédent l'élément à supprimer.
 for (i = 0; i < 2; i++)
   tmpPointeur = tmpPointeur->suivant;
 // Puis on pose un pointeur temporaire sur cet élément.
 supprPointeur = tmpPointeur->suivant;
 // On fait pointer l'élément précédent sur l'élément suivant l'élément à suppprimer
 tmpPointeur->suivant = supprPointeur->suivant;
 // Enfin, on détruit l'élément à supprimer.
```

```
free(supprPointeur);
```









11 Conclusion

Les pointeurs sont un point difficile du C, seul une pratique régulière de ce concept vous permettra de réellement vous familiariser avec cette notion.