Les pointeurs du C et du C++.

Par CGi

Le 10 mars 2005

Introduction :

Les pointeurs vous harcèlent, vous hantent, vous terrorisent, ce document est fait pour vous. Il a pour but d'aider les débutants en C/C++ à aborder les pointeurs avec le moins d'appréhension possible. Avant de rentrer dans le vif du sujet, nous ferons un rappel sur la définition d'une variable.

Rappel :

Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée. Physiquement cette valeur se situe en mémoire.
Prenons comme exemple un entier nommé x :

```
int x;    // Réserve un emplacement pour un entier en mémoire.
```

```
x = 10;  // Ecrit la valeur 10 dans l'emplacement réservé.
```

Décomposons notre exemple :
Nous allons représenter la mémoire de l'ordinateur par des cases numérotées en ordre croissant. On considérera qu'une variable utilise une case, même si dans la réalité elles ne prennent pas la même quantité de mémoire selon leur type.
Voyons-en une partie sous forme d'un schéma :

Quand j'écris : **int x;**

Je réserve une case pour la variable x dans la mémoire, case numéro 62 dans le cas du schéma.

Quand j'écris : **x=10;**

J'écris la valeur 10 dans l'emplacement réservé pour x. On dit que x a pour valeur 10. Cette valeur est située physiquement à l'emplacement &x (adresse de x) dans la mémoire (62 dans le contexte du schéma).
Pour obtenir l'adresse d'une variable on fait précéder son nom avec l'opérateur '&' (adresse de) :

```
printf("%p",&x);
```

Ce qui dans le cas du schéma ci-dessus, afficherait 62.

Le pointeur :

Un pointeur est aussi une variable, il est destiné à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire. Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom du signe '*' lors de sa déclaration.
Poursuivons notre exemple :

```
int *px;   // Réserve un emplacement pour stocker une adresse mémoire.
```

px = &x; // Ecrit l'adresse de x dans cet emplacement.
Décomposons :

Quand j'écris : **int *px;**

Je réserve un emplacement en mémoire pour le pointeur px (case numéro 96 dans le cas du schéma). Jusqu'a ici il n'y a donc pas de différence avec une variable ordinaire.

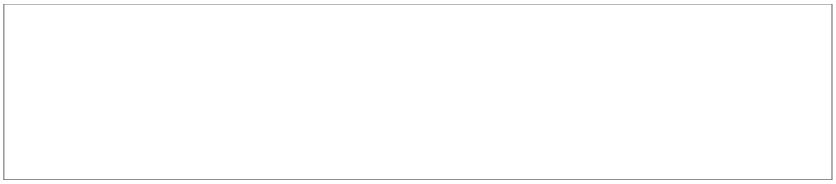
Quand j'écris : **px = &x;**

J'écris l'adresse de x à l'emplacement réservé pour le pointeur px. Je rappelle qu'un pointeur est destiné à mémoriser une adresse mémoire.

A l'emplacement réservé pour le pointeur px, nous avons maintenant l'adresse de x. Ce pointeur ayant comme valeur cette adresse, nous pouvons donc utiliser ce pointeur pour aller chercher (lire ou écrire) la valeur de x. Pour cela on fait précéder le nom du pointeur de l'opérateur de déréférencement '*'.
Donc l'instruction suivante :

printf("%d",*px);

Affiche la valeur de x par pointeur déréférencé (10 dans le cas du schéma).



On peut donc de la même façon modifier la valeur de x :

```
*px = 20; // Maintenant x est égal à 20.
```

On se rend vite compte qu'un pointeur doit être initialisé avec une adresse valide, c'est à dire qui a été réservée en mémoire (allouée) par le programme pour être utilisé. Imaginez-vous l'instruction précédente, si nous n'avions pas initialisé le pointeur avec l'adresse de x, l'écriture se ferait en un lieu indéterminé de la mémoire.

Dans l'exemple précédent vous avez dû remarquer que nous avons donné un type au pointeur (int *), même si dans un système donné un pointeur a toujours la même taille (4

octets pour un système à adressage sur 32 bits), le langage impose de leur donner un type. Si vous ne savez pas à l'avance sur quel type de données il va pointer vous pouvez lui donner le type void.

Il est d'usage de préfixer le nom des variables de type pointeur de la lettre "p" ceci pour une meilleure lisibilité du code.

Pointeurs et tableaux :

L'utilisation du pointeur de l'exemple précédent n'a qu'un intérêt pédagogique. Mais en C/C++, il y a des cas ou on ne peut pas se passer de leur utilisation. Prenons le cas des tableaux : le nom d'un tableau est un pointeur sur son premier élément.
Soit le tableau Tab suivant :

```
int Tab[10]={5,8,4,3,9,6,5,4,3,8};
```

L'instruction suivante affiche bien la valeur du premier élément du tableau par pointeur déréférencé.

```
printf("%d",*Tab);
```

Ceci démontre que le nom d'un tableau est bien un pointeur sur son premier élément. On peut alors tout à fait déclarer un pointeur et l'initialiser avec le nom du tableau puisque c'est un pointeur. A condition bien sûr qu'il soit du même type, pointeur sur des entiers dans notre cas :

```
int *pTab;
pTab = Tab;
```

On peut donc se servir de ce pointeur comme s'il était un tableau et des opérateurs crochets [] pour accéder à ses éléments :

```
printf("%d",pTab[0]);          // Affiche 5.
```

Mais ce n'est pas un autre tableau c'est le même que Tab, il référence le même emplacement en mémoire :

```
pTab[0]++;
printf("%d",Tab[0]);           // Affiche 6.
```

Quand j'incrémente le premier élément du tableau en utilisant le pointeur c'est bien le premier élément du tableau d'origine qui est incrémenté.
Que ce passe t'il si j'écris :

```
printf("%d",Tab[10]);
```

J'ai l'affichage d'une valeur ne faisant pas partie de mon tableau. Ceci est dû au fait que je vais lire une valeur en dehors des limites du tableau sans que le système signale une erreur. L'opérateur crochet n'étant qu'une écriture simplifiée du pointeur déréférencé, ceci :

```
printf("%d",Tab[10]);
```

est équivalent à cela :

```
printf("%d",*(Tab+10));
```

Un pointeur et donc un tableau peuvent facilement accéder à n'importe quel lieu en mémoire sans la moindre alerte, jusqu'au plantage de l'application.
Cette dernière écriture ne vous est peut-être pas familière pour accéder au contenu d'un tableau. Elle nous amènera donc à parler de l'arithmétique des pointeurs.

Arithmétique des pointeurs.

Reprenons le cas de notre tableau Tab. De même que pTab, puisqu'initialisé avec Tab, C'est un pointeur sur son premier élément. Si j'incrémente le pointeur pTab il ne contiendra pas l'adresse du premier élément du tableau + 1, mais l'adresse de l'élément suivant. La valeur de l'adresse qu'il contient sera donc incrémentée de la taille du type qu'il référence. Ceci est l'une des raisons pour lesquelles il faut donner un type à un pointeur.
Donc si j'écris :

```
printf("%d",*(Tab+1));        // Affiche 8.
```

J'ai bien l'affichage du deuxième élément du tableau.
De même si j'écris :

```
pTab++;
printf("%d\n",pTab[0]);
```

J'ai aussi l'affichage du deuxième élément du tableau d'origine, puisque mon pointeur ayant été incrémenté d'une unité, il contient maintenant l'adresse du deuxième élément du tableau (par contre on ne peut pas incrémenter le pointeur Tab car il a été déclaré en tant que tableau et, par conséquent, c'est un pointeur constant).
On peut de la même façon décrémenter un pointeur, lui ajouter ou lui soustraire une valeur numérique entière (attention toutefois à ne pas sortir des zones de mémoire allouées). Donc si j'écris :

```
pTab = Tab+4;
printf("%d\n",pTab[0]);       // Affiche 9.
```

J'ai l'affichage du 5ème élément du tableau car pTab est maintenant un pointeur sur son 5ème élément.

Pointeurs et allocation dynamique de la mémoire.

Une utilisation très courante des pointeurs est l'allocation dynamique de la mémoire. Quand on fait une allocation dynamique de mémoire on obtient en retour un pointeur sur la zone mémoire allouée.
Exemple en C :

```
int *p = malloc(10*sizeof(int));
if(p)
  {
    p[0] = 5;
    printf("%d",p[0]);
    free(p);
  }
```

Dans cet exemple l'instruction malloc nous retourne un pointeur sur une zone mémoire de la taille de 10 entiers (en C++, le malloc retournant un pointeur de type void (void*), nous devons le transtyper pour l'utiliser avec des entiers). Nous pouvons maintenant utiliser notre pointeur comme s'il était un tableau de 10 entiers (malloc retourne la valeur NULL si l'allocation à échouée). Les emplacements mémoire alloués dynamiquement

doivent être libérés explicitement, à l'aide de l'instruction free dans le cas d'une allocation par malloc.

En C++ le principe est identique sauf que l'on préferera l'utilisation de l'opérateur new pour l'allocation et delete (delete[] dans le cas d'un tableau) pour la libération.

```
int *p = new int[10];
p[0] = 5;
std::cout << p[0];
delete[] p;
```

Attention toutefois à ne pas modifier ce pointeur car il est nécessaire pour la libération de la mémoire. Vous pouvez en ce cas le déclarer comme pointeur constant pour éviter de le modifier:

```
int *const p = new int[10];
```

Pointeurs comme paramètres de fonctions.

Une autre utilité des pointeurs est de permettre à des fonctions d'accéder aux données elles même et non à des copies. Prenons pour exemple une fonction qui échange la valeur de deux entiers :

```
void exchange(int *x, int *y)
{
 int tmp;
 tmp = *x;
 *x = *y;
 *y = tmp;
}
```

Pour que la fonction puisse affecter leurs nouvelles valeurs à chaque variable elle doit y avoir accès, le passage des variables par pointeur permet donc à la fonction d'accéder aux variables par pointeur déréférencé.
Voyons l'exemple d'utilisation de la fonction :

```
int a;
int b;
a = 5;
b = 10;
printf("a = %d\nb = %d\n",a,b);
exchange(&a, &b);
printf("a = %d\nb = %d\n",a,b);
```

On passe donc l'adresse des variables a et b comme paramètres puisque la fonction attend des pointeurs comme paramètres. Pour le pointeur les paramètres sont en fait passés par valeur, puisque l'adresse de la variable est bien copiée dans le pointeur créé afin d'être utilisé par la fonction (à savoir qu'en C++ on aurait pu utiliser les références pour cet exemple).

Le passage de paramètres sous forme de pointeur est aussi utilisé pour passer un tableau en tant que paramètre de fonction, c'est d'ailleurs la seule solution possible dans le cas d'un tableau. La fonction reçoit donc un pointeur sur le premier élément du tableau, mais

la fonction ne devant pas accéder en dehors des limites du tableau, elle doit pouvoir en contrôler le traitement dans ce cas. Pour une chaîne de caractères, on peut tester la présence du caractère de fin de chaîne '\0', mais dans la majorité des autres cas il faudra fournir à la fonction la taille du tableau. Prenons comme exemple une fonction qui retourne la plus grande valeur d'un tableau d'entier :

```
int max(int *tab, int n)
{
   int x;
   int nmax;
   nmax = 0;
   for(x=0; x<n; x++)
            if(tab[x]>nmax) nmax=tab[x];
   return nmax;
}
```

Nous lui fournissons donc l'adresse du premier élément du tableau comme premier paramètre et la taille du tableau en second paramètre. Dans l'implémentation de la fonction nous utilisons le pointeur comme s'il était un tableau d'entier (utilisation de l'opérateur crochet "[]" pour accéder à ses éléments). Voici un code d'utilisation de la fonction max :

```
int Tab[] = {12,5,16,7,3,11,14,6,11,4};
printf("%d",max(Tab,10));
```

On aurait pu écrire l'entête de la fonction comme ceci :

```
int max(int tab[], int n)
```

ou

```
int max(int tab[10], int n)
```

Ces écritures étant équivalentes à la première, la fonction recevra dans tous les cas une copie du pointeur sur le tableau.

Les pointeurs de fonction :

La valeur renvoyée par le nom (seul) d'une fonction étant l'adresse de son code en mémoire, nous pouvons l'affecter à un pointeur.
Dans l'exemple ci-dessous ou nous créons un pointeur de fonction sur la fonction "max" vu précédemment.

```
int(*pmax)(int*, int);
pmax = max;

printf("%d",pmax(Tab,10));
```

Le pointeur doit être déclaré avec la signature de la fonction, c'est-à-dire dans le cas de notre exemple le pointeur pmax est un pointeur sur des fonctions recevant en paramètre un pointeur sur un entier puis un entier et retournant un entier. Au pointeur ainsi déclaré doit ensuite être affectée l'adresse d'une fonction (max dans notre exemple) ayant la même signature. Le pointeur s'utilise alors avec la même syntaxe que la fonction.

Conclusion :

Ce document n'ayant pour but que d'aborder les pointeurs, j'espère tout de même qu'il vous aura un peu aidé à leurs compréhensions. Il faut bien sûr ne les utiliser que quand cela est nécessaire. En C++ d'autre type de données permettent d'en faire abstraction. Signalons aussi que l'API Windows utilise abondamment les pointeurs comme paramètres de fonctions. Les pointeurs de fonctions y sont utilisés pour aller chercher des fonctions incluses dans les librairies dynamiques (dll).

Pour continuer vous pouvez aller consulter le tutoriel sur **les listes chaînées** où les pointeurs sont abondamment utilisés.

Bonne lecture,
CGi.

Avec la contribution de gege2061 et Taxol pour la relecture.