Principe et usage des pointeurs en langage C

1. Adressage de la mémoire de l'ordinateur

Lorsque l'on lance l'exécution d'un programme, toutes les instructions de ce programme et ses données sont chargées dans la mémoire de l'ordinateur. De façon très approximative, on peut considérer cette dernière comme une suite de cases, chacune de ces cases pouvant contenir une variable. Chacune de ces cases a un numéro d'ordre (adresse) dans la mémoire de l'ordinateur et un contenu (valeur de la variable).

Soit le programme suivant :

```
main demo(){
char premier='a',
…..
```

Une vue partielle de la mémoire au lancement de ce programme donnerait ("approximativement")

Adresse case mémoire	Contenu case mémoire	Nom variable stockée
Case 0	peu importe !	….peu importe !
Case 1		
……..		
Case **10345**	**'a'**	*….premier*

2. Définition d'un pointeur

Une variable de type pointeur est une variable qui contient l'adresse mémoire où est stockée une autre variable.

Par exemple en reprenant l'exemple précédent et en supposant que la variable panneau est un pointeur qui pointe vers la variable premier on aurait:

Adresse case mémoire	Contenu case mémoire	Nom variable stockée
Case 0	peu importe !	….peu importe !
Case 1		
……..		
Case **10345**	**'a'**	*….premier*
Case **108945**	**10345**	*….panneau*

panneau contient l'adresse à laquelle est stockée la variable premier, c'est donc est un pointeur vers premier.

3. Déclaration d'un pointeur

On déclare une variable de type pointeur en précisant vers quel type de variable elle pointe, et en la précédent

par * ; ainsi par exemple:

char * verschar; verschar est une variable pointeur vers une variable de type char

int * versint; ; versint est une variable pointeur vers une variable de type int

float *versfloat; versfloat est une variable pointeur vers une variable de type float
FILE * versfile; versfile est une variable pointeur vers un descripteur de fichier
int (* versfonc) (int, float); versfonc est une variable pointeur vers une fonction retournant un entier et ayant pour paramètres une variable entière et une variable de type float

4. Opérateur & et * , initialisation d'un pointeur

Un pointeur, pour pouvoir être utilisé, doit être initialisé (juste après sa déclaration le pointeur contient une adresse aléatoire c'est à dire pointant vers une adresse invalide).
Pour obtenir l'adresse de la variable vers laquelle on doit pointer on peut utiliser l'opérateur &
Ainsi l'exemple ci dessous:

```
main(){
int aaa=0;       /* la variable entière aaa est déclarée */
int * versint;   /* la variable versint est déclarée comme pointeur vers un entier, et contient
                    initialement une valeur aléatoire */
aaa=5;           /* aaa est initialisée à la valeur 5 */
versint=&aaaa; /* l'adresse de la variable aaa est mise dans versint qui est maintenant un pointeur
                    valide */
```

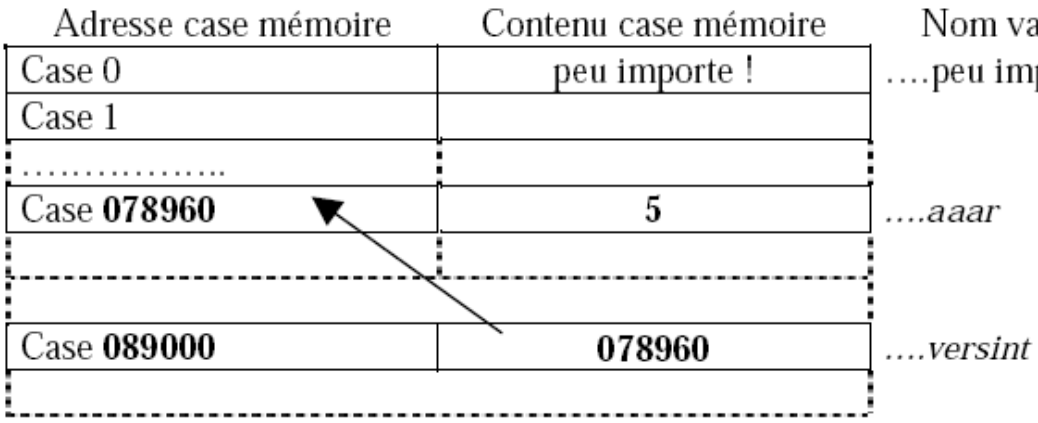
Reprenons l'exemple ci-dessous en détaillant l'état de la mémoire à différentes étapes:
Etape 1: le programme est chargé en mémoire et n'a pas commencé son exécution

Adresse case mémoire	Contenu case mémoire	Nom variable stockée
Case 0	peu importe !	….peu importe !
Case 1		
……..		
Case **078960**	**0**	*….aaa*
Case **089000**	**VALEUR ALEATOIRE**	*….versint*

Contenu de la mémoire après exécution de la ligne aaa=5 ;

Adresse case mémoire	Contenu case mémoire	Nom variable stockée
Case 0	peu importe !	….peu importe !
Case 1		
……..		
Case **078960**	**5**	*….aaa*
Case **089000**	**VALEUR ALEATOIRE**	*….versint*

Contenu de la mémoire après exécution de la ligne versint=&aaa;



L'opérateur * mis devant un pointeur permet d'obtenir le contenu de la variable pointée par le pointeur; ainsi après exécution du code précédent, un printf("%d",*verint) afficherait 5

Autre exemple:

```
main(){
int aaa,bbb;  /* les variables entière aaa et bbb sont déclarées */
int * versint;          /* la variable versint est déclarée comme pointeur vers un entier,
                        et contient initialement une valeur aléatoire */
aaa=5;        /* aaa est initialisée à la valeur 5 */
versint=&aaaa;  /* l'adresse de la variable aaa est mise dans versint qui est
maintenant un                   pointeur valide */
bbb=*versint;  /* bbb prend pour valeur le contenu de la variable pointée par versint
                c.a.d la valeur contenue dans aaa */
```

5. Les opérations élémentaires sur pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

- Les opérateurs ***** et **&** ont la même priorité que les autres opérateurs unaires (la négation **!**, l'incrémentation **++**, la décrémentation **--**). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.
- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

```
Y = *P+1      ⇔ Y = X+1
*P = *P+10    ⇔ X = X+10
*P += 2       ⇔ X += 2
++*P          ⇔ ++X
(*P)++        ⇔ X++
```

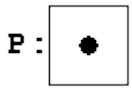
Dans le dernier cas, les parenthèses sont nécessaires:

Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas l'objet* sur lequel P pointe.

On peut uniquement affecter des adresses à un pointeur.

Le pointeur NUL

Seule exception: La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



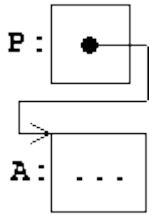

```
int *P;
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

Résumons:



Après les instructions:

```
int A;
int *P;
P = &A;
```

A désigne le contenu de A

&A désigne l'adresse de A

P désigne l'adresse de A

***P** désigne le contenu de A

En outre:

&P désigne l'adresse du pointeur P

***A** est illégal (puisque A n'est pas un pointeur)

6. Pointeurs et tableaux

En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs. En général, les versions formulées

avec des pointeurs sont plus compactes et plus efficientes, surtout à l'intérieur de fonctions. Mais, du moins pour des débutants, le 'formalisme pointeur' est un peu inhabituel.

6.1. Adressage des composantes d'un tableau

Comme nous l'avons déjà constaté au chapitre 7, le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes:

`&tableau[0]` et **`tableau`**

sont une seule et même adresse.

En simplifiant, nous pouvons retenir que *le nom d'un tableau est un **pointeur constant** sur le premier élément du tableau*.

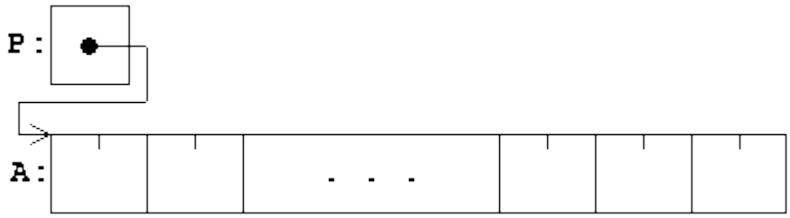
Exemple

En déclarant un tableau A de type **int** et un pointeur P sur **int**,

```
int A[10];
int *P;
```

l'instruction:

`P = A;` est équivalente à **`P = &A[0];`**



Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante. Plus généralement,

`P+i` pointe sur la i-ième composante derrière P et

`P-i` pointe sur la i-ième composante devant P.

Ainsi, après l'instruction,

```
P = A;
```

le pointeur P pointe sur A[0], et

`*(P+1)` désigne le contenu de A[1]

`*(P+2)` désigne le contenu de A[2]

`...` **`...`**

`*(P+i)` désigne le contenu de A[i]

7. Arithmétique des pointeurs

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le

confort de ces opérations en C est basé sur le principe suivant:

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

- **Affectation par un pointeur sur le même type**

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

```
P1 = P2;
```

fait pointer P1 sur le même objet que P2

- **Addition et soustraction d'un nombre entier**

Si P pointe sur l'élément A[i] d'un tableau, alors

P+n pointe sur A[i+n]

P-n pointe sur A[i-n]

- **Incrémentation et décrémentation d'un pointeur**

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

P++; P pointe sur A[i+1]

P+=n; P pointe sur A[i+n]

P--; P pointe sur A[i-1]

P-=n; P pointe sur A[i-n]

Domaine des opérations

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies *à l'intérieur d'un tableau*. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Seule exception: Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur *avant* l'évaluation de la condition d'arrêt.

Exemples

```
int A[10];

int *P;
P = A+9;    /* dernier élément -> légal */
P = A+10;   /* dernier élément + 1 -> légal */
P = A+11;   /* dernier élément + 2 -> illégal */
P = A-1;    /* premier élément - 1 -> illégal */
```

- **Soustraction de deux pointeurs**

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:

P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2
- zéro, si P1 = P2
- positif, si P2 precède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par **<**, **>**, **<=**, **>=**, **==**, **!=**.

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

8.Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur int peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur char peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur char peut en plus contenir *l'adresse d'une chaîne de caractères constante* et il peut même être *initialisé* avec une telle adresse.

A la fin de ce chapitre, nous allons anticiper avec un exemple et montrer que les pointeurs sont les éléments indispensables mais effectifs des fonctions en C.

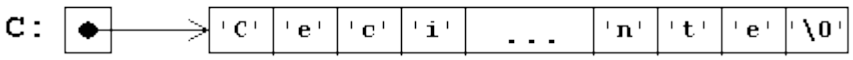
- Pointeurs sur char et chaînes de caractères constantes

Affectation

a) On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**:

Exemple

```
char *C;
C = "Ceci est une chaîne de caractères constante";
```



Nous pouvons lire cette chaîne constante (p.ex: pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

Initialisation

b) Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *B = "Bonjour !";
```

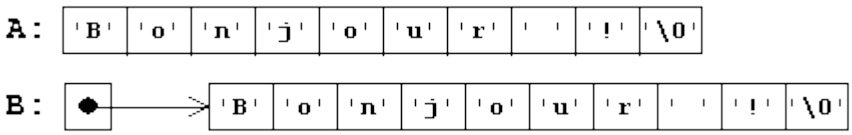
Attention !

Il existe une différence importante entre les deux déclarations:

```
char A[] = "Bonjour !";   /* un tableau  */
char *B  = "Bonjour !";   /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



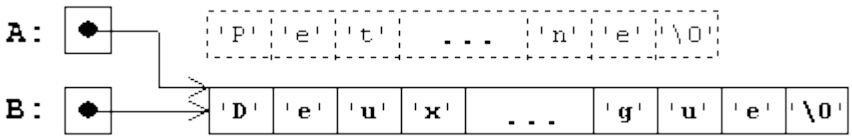
Modification

c) Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur:

Exemple

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:

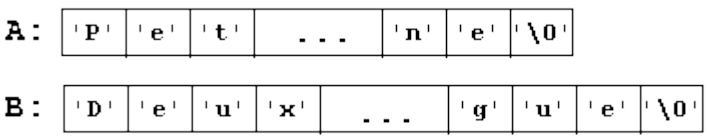


Attention !

Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;              /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !";    /* IMPOSSIBLE -> ERREUR !!! */
```



Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que ***l'adresse représentée par le nom d'un tableau reste toujours constante***.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle) ou déléguer cette charge à une fonction de <*stdio*> ou <*string*>.

9. Pointeurs et tableaux à deux dimensions

L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions. Voyons cela sur un exemple:

Exemple

Le tableau M à deux dimensions est défini comme suit:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe (oh, surprise...) sur le ***tableau*** M[0] qui a la valeur:

{0,1,2,3,4,5,6,7,8,9}.

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur:

{10,11,12,13,14,15,16,17,18,19}.

Explication

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le ***vecteur* {0,1,2,3,4,5,6,7,8,9}**, le deuxième élément est **{10,11,12,13,14,15,16,17,18,19}** et ainsi de suite.

L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que:

M+I désigne l'adresse du tableau **M[I]**

Problème

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d.: aux éléments M[0][0], M[0][1], ... , M[3][9] ?

Discussion

Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type* ***int)*** en un pointeur sur ***int***. On pourrait se contenter de procéder ainsi:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
int *P;
P = M;   /* conversion automatique */
```

Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0]. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent: Généralement, on gagne en lisibilité en explicitant la conversion mise en oeuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

Solution

Voici finalement la version que nous utiliserons:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
int *P;
P = (int *)M;  /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Exemple

Les instructions suivantes calculent la somme de tous les éléments du tableau M:

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
     SOM += *(P+I);
```

Attention !

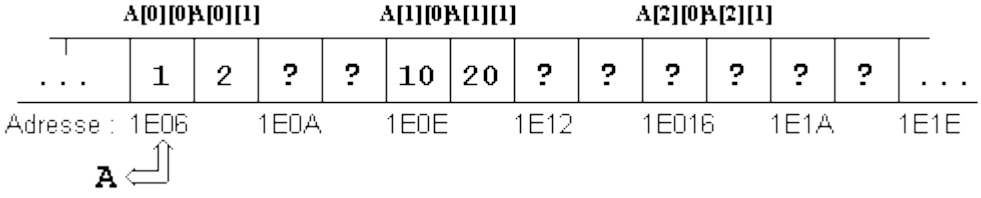
Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel ***il faut calculer avec le nombre de colonnes indiqué dans la déclaration*** du tableau.

Exemple

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes:

```
int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;
```

Dans la mémoire, ces composantes sont stockées comme suit :



L'adresse de l'élément A[I][J] se calcule alors par:

A + I*4 + J

10. Tableaux de pointeurs

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.

Déclaration d'un tableau de pointeurs

```
int *D[];
```

déclare ***un tableau de pointeurs*** sur des éléments du type **int**

D[i]	peut pointer	sur des variables simples ou sur les composantes d'un tableau.
D[i]		désigne l'adresse contenue dans l'élément i de D (Les adresses dans D[i] sont variables)
***D[i]**		désigne le contenu de l'adresse dans D[i]

Si D[i] pointe dans un tableau,

10. Utilisation de pointeur dans les procédures

En C, il est nécessaire de programmer de façon modulaire en utilisant des procédures,;
Voici un exemple d'utilisation de procédure:

```
void increment(int bbb) {
bbb=bbb+1;
printf("\n bbb vaut %d\n",bbb);
return;
}
main(){
int aaa;  /* la variable entière aaa est déclarée */
aaa=5;  /* aaa est initialisée à la valeur 5 */
increment(aaa);  /* appel de incrément avec la variable aaa comme paramètre */
printf("%d",aaa);  /* affichage de la valeur contenue dans aaa donc 5 */
}
```

Déroulons l'exécution du programme:

- aaa est initialisée à 5
- appel increment en lui "passant" aaa comme paramètre
- le contenu de aaa est recopié dans le variable bbb locale à la procédure increment
- bbb est incrémenté de 1
- on sort de la procédure increment
- on affiche le contenu de aaa qui vaut 5

Chaque fois que l'on appelle une procédure, les paramètres passés à la procédure au moment de l'appel sont recopiés dans des variables locales à la procédure (on parle de **passage de paramètres par valeur**)

Pour pouvoir modifier la variable aaa depuis la procédure, il faut passer l'adresse de aaa à la procédure en paramètre, adresse qui sera recopiée dans une variable pointeur locale à la procédure incrément; le programme devient alors:

```
void increment(int* bbb) { / * la variable bbb est un pointeur vers un entier */
*bbb=*bbb+1;  /* le contenu de la variable pointée par bbb est incrémenté de 1 */
printf("\n bbb pointe vers une variable entiere contenant %d\n",*bbb);
return;
}
main(){
int aaa;  /* la variable entière aaa est déclarée */
aaa=5;  /*la variable aaa est initialisée à la valeur 5 */
increment(&aaa);  /* appel de incrément avec l'adresse de la variable aaa comme paramètre*/
printf("%d",aaa);  /* la variable aaa contient maintenant 6 */
}
```

Déroulons l'exécution du programme:

- aaa est initialisée à 5
- appel increment en lui "passant" l'adresse de aaa comme paramètre (&aaa),
- l'adresse de aaa est recopiée dans le pointeur vers entier bbb local à la procédure increment,
- la variable dont l'adresse est contenu dans bbb est incrémenté de 1 (*bbb=*bbb+1) (donc le contenu de la variable aaa est incrémenté de 1)
- on sort de la procédure increment,
- on affiche le contenu de aaa qui vaut maintenant 6

NB: Ceci explique que lorsqu'on utilise la procédure scanf pour initialiser le contenu de variables, toutes les variables passées à scanf doivent être précédées par & (i.e. on passe l'adresse des variables à initialiser avec les valeurs saisies par l'utilisateur)

2. Utilisation dans les tableaux

En C, on peut définir des tableaux; ces tableaux sont stockés comme une suite de variable du même type; Ainsi la déclaration d'un tableau de 100 entiers:

```
int montableau[100];
```

Ces 100 entiers sont stockés de manière consécutives, on peut utiliser un pointeur pour parcourir ce tableau

Soit le code:

```
int i,montableau[100];
int * versint;
versint=&montableau[0];
for (i=0;i<100;i++) *(versint+i)=i;
```

Ce code initialise les 100 entiers du tableau, chacun avec une valeur différente (t[n] vaut n).

Il est à noter que que l'identificateur d'un tableau peut aussi être considéré comme un pointeur vers le 1er élément du tableau ; ainsi l'identificateur mon tableau est donc aussi un pointeur vers le 1er entier du tableau de 100 entiers, on aurait pu donc tout aussi bien écrire:

```
int montableau[100];
int i,* versint;
versint=montableau;
for (i=0;i<100;i++) *(versint+i)=i;
```