

UNIVERSITÉ DE LIÈGE

INFO0946

INTRODUCTION À LA PROGRAMMATION

GAMECODE : Un exercice dont vous êtes le Héros · l'Héroïne

Pointeurs

Benoit DONNET

Simon LIÉNARDY

Géraldine BRIEVEN

Tasnim SAFADI Lev MALCEV

20 décembre 2021



Préambule

Exercices

Dans ce GAMECODE, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur les pointeurs et la manipulation de la mémoire.

Il est dangereux d'y aller seul ¹ !

Partagez vos commentaires, questions, solutions alternatives sur le forum [eCampus](#). N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

7.1 Rappel Général sur les Expressions

Si vous avez déjà lu ce rappel ou si vous êtes suffisamment à l'aise avec les expressions et leurs évaluations, vous pouvez directement atteindre l'énoncé de l'exercice.

7.1.1 Expression

Une *expression* (cfr. Chapitre 1, Slide 17) est la description du calcul d'une valeur, le résultat du calcul ayant un certain type.

Plus précisément, une expression est :

1. une constante (ou valeur littérale – e.g., 'a'). L'évaluation retourne le littéral lui-même ;
2. une variable, dénotée par son identificateur (e.g., `x`). Dans ce cas, l'évaluation retourne la valeur courante de la variable ;
3. obtenue par l'application d'opérateurs à d'autres expressions.

7.1.2 Opérateurs

La définition générale d'une expression (voir point 3 de la définition d'une *expression*) indique qu'on peut appliquer un ou plusieurs opérateur(s) afin de composer une expression plus complexe.

Un *opérateur* permet d'évaluer une expression bien définie sur des valeurs (*opérandes*) en produisant un résultat (*valeur* de l'expression).

La forme générale d'une expression avec un opérateur est la suivante :

$$x \alpha x$$

où x est l'opérande et α l'opérateur.

Un opérateur peut être :

- *unaire*. Il requiert une seule opérande. Exemple : -5 .
- *binaire*. Il requiert deux opérandes. Exemple : $4 + x$.
- *ternaire*. Il requiert trois opérandes. Ce type d'opérateur ne sera pas abordé dans le cadre du cours.

Le tableau 1 donne une liste des opérateurs "simples" en C.

Alerte : Le contexte est ultra important

Attention, la signification exacte d'un opérateur peut dépendre du contexte dans lequel il est utilisé. Par exemple : $5/2$ correspond à la division entière et donne pour résultat 2. Par contre, $5/2.0$ correspond à la division réelle et donne pour résultat 2.5. Pour la division, c'est donc le type des opérandes qui détermine la signification exacte de l'opération.

Il en va de même pour l'opérateur $*$. La liste des opérateurs montre que l'opérateur $*$ a deux significations : la multiplication et le dérèférencement. A nouveau, c'est le contexte dans lequel l'opérateur va être utilisé qui déterminera sa sémantique. Ainsi, par exemple, $3 * x$ correspond clairement à une multiplication car on se trouve dans le cas d'un opérateur binaire. L'expression $*x$ correspondra au dérèférencement de x (opérateur unaire) à la condition, bien entendu, que le type de x soit pointeur.

Opérateur	Signification	Type
Opérateurs <i>Arithmétiques</i>		
+	addition	binaire
-	soustraction	
*	multiplication	
/	division	
%	modulo	unaire
-	changement de signe	
Opérateurs de <i>Comparaison</i>		
<	plus petit que	binaire
<=	plus petit ou égal	
>	plus grand que	
>=	plus grand ou égal	
==	égal	
!=	différent	
Opérateurs <i>Booléens</i>		
&&	et “lazy”	binaire
	ou “lazy”	unaire
!	négation	
Opérateurs sur les <i>Pointeurs</i>		
*	Déréférencement	unaire
&	Référencement	

TABLE 1 – Liste (non exhaustive) des opérateurs usuels en C.

7.2 Énoncé

Soit l'état de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

La première colonne indique le nom des variables, la deuxième des adresses mémoires et, enfin, la troisième donne la valeur stockée à cette adresse. De plus, voici comment ont été déclarées les variables (on suppose que les entiers sont représentés sur 4 octets) :

```

1 int data, x, *ptr = &x;
2 char *E = "Session", *F = E + 7;
```

Dans ce GAMECODE, il vous est demandé de donner la valeur des **expressions** des expressions ci-dessous. Considérez qu'entre chaque expression, la mémoire est réinitialisée telle que présentée dans le schéma. Si un accès mémoire est demandé à une adresse hors de l'intervalle [204, 340], mentionnez l'erreur de segmentation par la valeur SF (pour Segmentation Fault). Les adresses sont aussi représentées sur 4 octets.

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `&data - ptr`
9. `*(x = data)`
10. `F[-4] - *(E + 2)`

7.2.1 Méthode de Résolution

Étant donné que l'état de la mémoire est restauré à son état initial pour chaque expression, l'ordre de résolution de l'exercice n'a absolument aucune importance. Voici les liens vers les sections dédiées à chaque expression :

1. [*data](#)
2. [**data](#)
3. [&*data](#)
4. [*&x](#)
5. [*ptr++](#)
6. [++&data](#)
7. [ptr + data](#)
8. [&data - ptr](#)
9. [*\(x = data\)](#)
10. [F\[-4\] - *\(E + 2\)](#)

La synthèse des évaluations des expressions se trouve [ici](#).

7.3 *data

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.3.4](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.3.3](#)

7.3.1 Rappels sur les Pointeurs

7.3.1.1 Définition

En informatique, une *variable* associe un nom à une valeur appartenant à un ensemble donné. Le nom de la variable est son *identificateur* et permet d'y faire référence dans le cadre d'une expression. L'ensemble de valeurs correspond au *type* de la variable et indique l'ensemble de valeurs possibles de la variable (par conséquent, il précise la nature du contenu de la variable).

En pratique, une variable correspond à un emplacement mémoire qui contiendra une valeur.

Le schéma suivant donne un exemple de relation entre une variable et la mémoire pour les déclarations suivantes :

```
1 int entier = 255;
2 float flottant = 5620.45;
3 double reel = 3.141592653;
```

		:	:
	2016	??	
	2012	3.141592653	
reel :	2008		
flottant :	2004	5620.45	
entier :	2000	255	
	:	:	

La première colonne donne le nom des variables, la deuxième donne des adresses mémoires (ici exprimée en **base 10**²) et, enfin, la troisième donne la valeur stockée à cette adresse (?? signifie que la valeur est indéterminée). De plus, voici comment ont été déclarées les variables (on suppose que les entiers – **int** – sont représentés sur 4 octets³, les flottants – **float** – sur 4 octets et, enfin, les réels – **double** – sur 8 octets).

On voit, sur le schéma, que les adresses sont alignées sur des multiples de 4. C'est tout à fait normal. Pour rappel, un emplacement mémoire correspondra toujours à un *mot* mémoire, soit la plus petite unité mémoire manipulable directement. Ici, nous codons les entiers sur un mot mémoire, soit 4 octets. Chaque adresse va donc correspondre à un multiple de 4, ce qui coïncide avec les octets gérés par un mot.

A noter que la variable **reel** occupe deux mots mémoires puisque les réels (i.e., **double**) sont encodés sur 8 octets.

Il est possible de mémoriser, dans un emplacement mémoire (i.e., une variable), l'adresse d'un autre emplacement mémoire (i.e., une autre variable).

Un *pointeur* est un type particulier de variable dont la valeur est une adresse mémoire.

	:	:
	2016	??
entier :	2012	56
	2008	??
	2004	??
ptr :	2000	2012
	:	:

Par exemple, dans l'illustration de la mémoire ci-contre, la variable **ptr** est un pointeur car sa valeur, 2004, fait référence à l'adresse mémoire associée à la variable **entier**. La flèche bleue symbolise le fait que **ptr** "pointe" vers **entier**. On peut donc voir un pointeur comme un mécanisme d'indirection permettant d'accéder, indirectement, à une zone mémoire. Dans l'exemple ci-dessous, on retrouve deux variables, **entier** et **ptr**, qui regarde la même zone mémoire : **entier** de manière directe, **ptr** de manière indirecte.

2. En pratique, une adresse mémoire est représentée en base hexadécimale – cfr. Introduction, Slides 35 → 36.

3. les **short** sur 2, les **long** sur 8

7.3.2 Déclaration

On peut déclarer, bien entendu, des variables de type “pointeur”. Ainsi, Si T est un type quelconque, T* désigne le type d’un pointeur vers une variable de type T.

On peut donc appliquer les pointeurs à n’importe quel type de données, les types primitifs (e.g., int), comme les enregistrements.

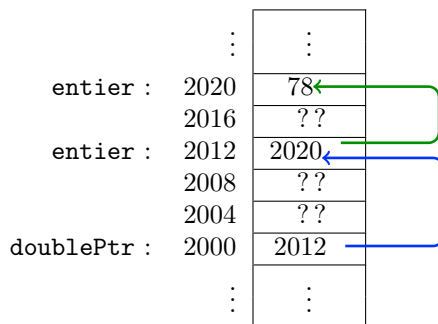
Par exemple :

```
1 int *ptr;
```

Dans ce cas, ptr est un pointeur sur entier. Le fait qu’il ny ai pas d’espace entre le * et le nom de la variable, dans la déclaration, n’a aucune importance (vous pouvez mettre un espace si vous le désirez). On peut, bien entendu, avoir un “pointeur de pointeur”. Par exemple :

```
1 int **doublePtr;
```

Dans ce cas, doublePtr contient un pointeur vers une zone mémoire qui elle-même contient un pointeur vers une zone mémoire contenant une valeur entière. Il y a donc deux niveaux d’indirection. On peut illustrer cela de la manière suivante :



En théorie, il n’y a aucune limite au mécanisme d’indirection. On peut avoir autant de niveaux d’indirection que désiré. En pratique, évidemment, les choses ne sont pas aussi simples : la mémoire d’un ordinateur n’est pas infinie et des limites seront imposées en fonction des ressources disponibles sur la machine. Dans tous les cas, le standard C impose que toute implémentation (d’un compilateur C) supporte au minimum 12 niveaux d’indirection.

7.3.2.1 Opérations sur les Pointeurs

Deux opérateurs permettent de manipuler les pointeurs :

référencement . C’est l’opérateur &. Appliqué à une variable (et une uniquement une variable – c’est donc un opérateur unaire), il permet d’obtenir l’adresse mémoire associée. C’est illustré à la ligne 5 du code ci-dessous. Le formatage %x permet d’afficher une adresse au format hexadécimal (c’est aussi le format pour les pointeurs). Le module printf() affichera la valeur 7D0 à l’écran (soit 2000 en base 10). La ligne 6, quant à elle, illustre un autre usage de l’opération de référencement : initialiser une variable de type pointeur, ip01, avec l’adresse d’une variable (i). ip01 contient la valeur 2000, soit l’adresse de i. Les lignes 8 à 18 sont d’autres exemples d’initialisation de pointeurs, à différents niveaux d’indirection.

déréférencement . C’est l’opérateur *. Appliqué à une variable (et uniquement une variable – c’est donc un opérateur unaire), il permet de “suivre la flèche” et d’aller dans le contenu de la case mémoire pointée par la variable. L’opérateur de déréférencement permet donc d’avoir soit une valeur à gauche (on va modifier la case pointée par la variable – cfr. ligne 19 dans l’extrait de code ci-dessous), soit une valeur à droite (on va lire le contenu de la case pointée par la variable pour l’utiliser dans une expression – cfr. ligne 7 dans l’extrait de code ci-dessous). L’opérateur de déréférencement peut être

appliqué plusieurs fois sur une même variable (cfr. ligne 19), pour autant que la variable a été déclarée avec autant de niveaux d'indirection.

```

1 #include <stdio.h>
2
3 int main(){
4     int i = 0;
5     printf("%x\n", &i);
6     int *ip01 = &i;
7     printf("%d\n", *ip01);
8     int **ip02 = &ip01;
9     int ***ip03 = &ip02;
10    int ****ip04 = &ip03;
11    int *****ip05 = &ip04;
12    int *****ip06 = &ip05;
13    int *****ip07 = &ip06;
14    int *****ip08 = &ip07;
15    int *****ip09 = &ip08;
16    int *****ip10 = &ip09;
17    int *****ip11 = &ip10;
18    int *****ip12 = &ip11;
19    *****ip12 = 1;
20
21    printf("%d\n", *****ip12);
22
23    return 0;
24 }//fin programme

```

	:	:	
ip12 :	2048	2044	←
ip11 :	2044	2040	←
ip10 :	2040	2036	←
ip09 :	2036	2032	←
ip08 :	2032	2028	←
ip07 :	2028	2024	←
ip06 :	2024	2020	←
ip05 :	2020	2016	←
ip04 :	2016	2012	←
ip03 :	2012	2008	←
ip02 :	2008	2004	←
ip01 :	2004	2000	←
i :	2000	1	←
	:	:	

7.3.2.2 Relation Pointeurs et Tableaux

Les tableaux (cfr. Chapitre 5) sont un cas particulier des pointeurs. En fait, un tableau est un pointeur. Soit le bout de code suivant :

```

1 #include <stdio.h>
2
3 int main(){
4     int tab[5] = {100, 101, 102, 103, 104};
5
6     return 0;
7 }//fin programme

```

Puisqu'on sait très bien qu'un tableau est représenté de manière contigue en mémoire (i.e., les différentes cases du tableau se suivent dans la mémoire), on obtient la représentation graphique suivante :

	:	:
	2016	104
	2012	103
	2008	102
	2004	101
tab :	2000	100
	:	:

On voit que la variable **tab** est associée avec l'adresse 2000. La première valeur du tableau (**tab[0]**) se trouve à l'adresse 2000. Les autres valeurs suivent en mémoire, par pas de 4 dans l'adressage (4 bytes pour stocker un **int**). De la même manière, la valeur associée à l'indice 1 (i.e., **tab[1]**) est 101 et se trouve à l'adresse 2004. Soit l'adresse de base de **tab** plus l'espace nécessaire pour passer la première valeur entière.

Si on poursuit le raisonnement, la valeur associée à l'indice 2 (i.e., `tab[2]`) est 102 et se trouve à l'adresse 2008. Soit l'adresse de base de `tab` plus l'espace nécessaire pour passer les deux premières valeurs entières.

On voit donc clairement que l'indice n'est là que pour donner un positionnement relatif par rapport à l'identificateur du tableau. Dès lors, de manière générale, l'instruction `tab[i]` permet d'accéder au contenu de la case située à `i` positions supérieures à `tab` dans la mémoire. Ceci revient à dire que `tab[i]` est équivalent à `*(tab+i)`. Il vient donc naturellement que `&tab[i]` est équivalent à `tab+i`.

En conclusion, un tableau est un pointeur vers la première valeur stockée dans le tableau. Les indices ne sont là que pour se déplacer dans la mémoire par rapport à la zone mémoire correspondant à la variable qui représente le tableau. C'est donc aussi la raison pour laquelle les indices d'un tableau commencent toujours à 0 (la position de base du tableau + 0 case mémoire).

Les deux extraits de code suivant :

```
1 int t[10];
2 int i;
3
4 for(i=0; i<10; i++)
5     t[i] = 1;
```

est équivalent à

```
1 int t[10];
2 int i;
3
4 for(i=0; i<10; i++)
5     *(t+i) = 1;
```

Il est évident que la notation `t+i` pour manipuler un tableau n'est pas naturelle et n'est pas à privilégier. Il est préférable de continuer à utiliser les tableaux comme on l'a toujours fait, soit avec les `[]`.

Enfin, il est bon de noter que lorsqu'un tableau est passé en paramètre d'un module (fonction ou procédure), il est converti automatiquement en son adresse (plus précisément, l'adresse de son premier élément). Ainsi, les deux prototypes suivants sont équivalents :

```
1 void init_tab(int tab[], int N);
2 void init_tab(int *tab, int N);
```

Dorénavant, nous privilégierons toujours la seconde écriture (utilisation du pointeur dans le prototype pour désigner un tableau).

7.3.2.3 Arithmétique des Pointeurs

Il est possible d'effectuer des opérations arithmétiques sur les pointeurs.

Les seules opérations valides sont les opérations externes (addition et soustraction des entiers) et la soustraction de pointeurs. Elles sont définies comme suit (la soustraction d'un entier est considérée comme l'addition d'un entier négatif) :

$p + i$ = adresse de $p + i$ *taille(élément pointé par p)

$p2 - p1$ = (adresse de $p2$ - adresse de $p1$) / taille(éléments pointés par $p1$ et $p2$)

Suite de l'Exercice

À vous! Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`

5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `&data - ptr`
9. `*(x = data)`
10. `F[-4] - *(E + 2)`

7.3.3 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `*data` et passez à la Sec. 7.3.4.

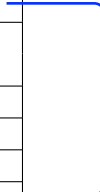
7.3.4 Mise en Commun de l'Évaluation de *data

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

Dans l'expression `*data`, l'opérateur `*` est un **opérateur de déréférencement**. Dans l'expression qui nous occupe, il permet de suivre la flèche et d'aller lire le contenu pointé par l'adresse mémoire contenue dans `data`. Si on repart de l'état initial de la mémoire et qu'on y applique une flèche au niveau de la variable `data`, il vient :

	⋮	⋮
	340	27
data:	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮



Ce mécanisme d'indirection (la flèche **bleue**) nous amène à la case d'adresse 208 dont la valeur stockée est 220. L'expression `*data` est donc évaluée à la valeur contenue dans la case d'adresse 208, soit la valeur 220.

Suite de l'Exercice

Évaluez les expressions :

1. `**data`
2. `&*data`
3. `*&x`
4. `*ptr++`
5. `++&data`

6. `ptr + data`
7. `&data - ptr`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.4 ****data**

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.4.2](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.4.1](#)

7.4.1 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `**data` et passez à la Sec. 7.4.2.

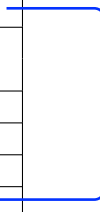
7.4.2 Mise en Commun de l'Évaluation de **data

Soit l'état initial de la mémoire suivant :

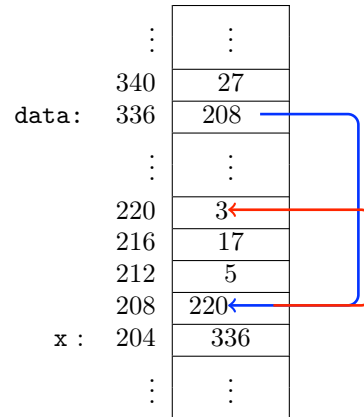
	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

Dans l'expression ****data**, l'opérateur ***** est un **opérateur de déréférencement**. Dans l'expression qui nous occupe, il permet de suivre la flèche et d'aller lire le contenu pointé par l'adresse mémoire contenue dans ***data**. On obtient donc la situation suivante :

	⋮	⋮
	340	27
data:	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮



Ce mécanisme d'indirection (la flèche **bleue**) nous amène à la case d'adresse 208 dont la valeur stockée est 220. Comme l'expression que nous devons évaluer contient deux niveaux d'indirection (i.e., deux **opérateurs de déréférencement**), il faut appliquer à ce résultat un deuxième déréférencement et aller voir le contenu de la case à l'adresse 220. Il vient :



Ce deuxième mécanisme d'indirection (la flèche **rouge**) nous amène à la case d'adresse 220 dont la valeur stockée est 3. L'expression ***data** est donc évaluée à la valeur contenue dans la case d'adresse 220, soit la valeur 3.

Suite de l'Exercice

Évaluez les expressions :

1. ***data**
2. **&*data**
3. ***&x**
4. ***ptr++**
5. **++&data**
6. **ptr + data**
7. **&data - ptr**
8. ***(x = data)**
9. **F[-4] - *(E + 2)**

7.5 `&*data`

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.5.3](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne vous souvenez plus des priorités des opérateurs, voyez la Section [7.5.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.5.2](#)

7.5.1 Rappels sur la Priorité des Opérateurs

La *priorité des opérateurs* précise l'ordre dans lequel les calculs doivent être effectués dans une expression complexe. Le Tableau 2 donne la priorité des opérateurs en C.

Priorité	Opérateurs	Sens d'Evaluation
++++	(), [], . ->	→
	!, --, ++, -, *, &	←
	*, /, %	→
	+, -	→
	«, »	→
	<, <=, >, >=	→
	==, !=	→
	&	→
	^	→
		→
	&&,	→
----	=, +=, -=, ...	←

TABLE 2 – Priorité des Opérateurs.

La deuxième ligne du **tableau** représente des opérateurs unaires. Dès lors, `*` est l'opérateur de déréférencement et `&` celui de référencement.

De manière générale, l'évaluation d'une expression se fait de la gauche vers la droite⁴. Cependant, dans certains cas (opérateurs unaires et opérateurs d'affectations), l'évaluation se fait de la droite (i.e., on évalue d'abord la *valeur à droite*) vers la gauche (on place le résultat dans la *valeur à gauche*).

Suite de l'Exercice

À vous! Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `&data - ptr`
9. `*(x = data)`
10. `F[-4] - *(E + 2)`

4. Puisque nous lisons de gauche à droite...

7.5.2 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `&*data` et passez à la Sec. 7.5.3.

7.5.3 Mise en Commun de l'Évaluation de `&*data`

Soit l'état initial de la mémoire suivant :

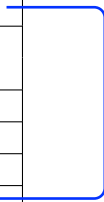
	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

L'expression `&*data` contient deux opérateurs : l'opérateur de référencement (`&`) et l'opérateur de déréférencement (`*`).

Ce sont deux opérateurs unaires ayant la même **priorité globale**. Cependant, l'évaluation de ces opérateurs se fait de droite à gauche. Dit autrement, l'expression `&*data` peut se réécrire : `&(*data)`.

On doit donc d'abord évaluer `*data`, ce qui donne graphiquement sur la mémoire :

	⋮	⋮
	340	27
data:	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮



Grâce au mécanisme d'indirection (la flèche **bleue**), on atteint la case dont l'adresse est 208 et la valeur 220. Sur ce résultat, il faut appliquer l'opérateur de référencement et, donc, obtenir l'adresse de la case mémoire sur laquelle nous sommes. Soit 208.

L'expression `&*data` est donc évaluée à la valeur 208.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `*&x`

4. `*ptr++`
5. `++&data`
6. `ptr + data`
7. `&data - ptr`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.6 *&x

Si vous voyez de quoi on parle, rendez-vous à la Section	7.6.2
Si la notion de pointeur vous pose problème, voyez la Section	7.3.1
Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section	7.1
Si vous ne vous souvenez plus des priorités des opérateurs, voyez la Section	7.5.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice	7.6.1

7.6.1 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `*&x` et passez à la Sec. 7.6.2.

7.6.2 Mise en Commun de l'Évaluation de `*&x`

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

L'expression `*&x` contient deux opérateurs : l'opérateur de référencement (`&`) et l'opérateur de déréférencement (`*`).

Ce sont deux opérateurs unaires ayant la même **priorité globale**. Cependant, l'évaluation de ces opérateurs se fait de droite à gauche. Dit autrement, l'expression `*&x` peut se réécrire : `*(&x)`.

On doit donc d'abord évaluer `&x`, ce qui revient à donner l'adresse de `x`. Soit 204.

Il faut maintenant appliquer l'opération de déréférencement sur la valeur 204 et aller voir le contenu de la case d'adresse 204. Soit 336.

Dès lors, au final, l'évaluation de `*&x` est équivalente à celle de l'expression (plus simple) `x` (dit autrement, le contenu de la case mémoire identifiée par la variable `x`).

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*ptr++`
5. `++&data`
6. `ptr + data`
7. `&data - ptr`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.7 *ptr++

Si vous voyez de quoi on parle, rendez-vous à la Section	7.7.3
Si la notion de pointeur vous pose problème, voyez la Section	7.3.1
Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section	7.1
Si vous ne vous souvenez plus des priorités des opérateurs, voyez la Section	7.5.1
Si l'opérateur d'incrémentatation reste un mystère, voyez la Section	7.7.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice	7.7.2

7.7.1 Rappels sur l'Opérateur d'Incrémentation/Décrémentation

Le **rappel sur l'affectation** indique que le sucre syntaxique permet, entre autre, de simplifier l'incréméntation/décrémentation d'une variable. Il existe une version encore plus simplifiée à l'aide des opérateurs d'incrémenté et de décrémentation.

L'opérateur d'incrémenté, ++, est un opérateur unaire (i.e., il ne s'applique qu'à une seule opérande) dont l'opérande est une valeur à gauche (i.e., l'opérande est là pour stocker le résultat de l'opération – ce ne peut donc être qu'une variable). L'équivalent pour la décrémentation existe : --.

L'opérateur d'incrémenté (resp. de décrémentation) peut être placé :

à droite de l'opérande :

```
1 x++
2 y--
```

la variable est incrémentée (x) ou décrémentation (y) d'une unité. Mais, attention, la valeur de toute l'expression est celle de l'opérande **avant** l'incrémenté/décrémentation.

à gauche de l'opérande :

```
1 ++x
2 --y
```

la variable est incrémentée (x) ou décrémentation (y) d'une unité. Mais, attention, la valeur de toute l'expression est celle de l'opérande **après** l'incrémenté/décrémentation.

Un moyen simple (mais efficace) pour se souvenir du fonctionnement de l'opérateur, c'est de s'appuyer sur le sens de lecture en français (gauche → droite). Si l'opérateur est à droite de l'opérande, alors on lit d'abord la valeur de variable, cette valeur devenant la valeur de toute l'expression, et, ensuite, on incrémente/décrémente la variable. Si l'opérateur est à gauche, on effectue d'abord l'incrémenté/décrémentation et, ensuite, on lit la valeur de la variable qui a déjà été modifiée par l'opérateur (la valeur de l'expression est alors cette nouvelle valeur de la variable).

Alerte : Lequel faut-il utiliser ?

Est-il souhaitable de placer l'opérateur à gauche ou à droite de l'opérande ?

En théorie, placer l'opérateur à gauche est plus efficace (une opération en moins pour le CPU, comparé au placement à droite).

En pratique, avec les machines dont on dispose en 2020, la différence entre les deux est devenue totalement négligeable.

Dans les codes que vous aurez à construire durant le quadrimestre, vous avez la possibilité d'utiliser le positionnement que vous préférez pour l'opérateur d'incrémenté/décrémentation.

Suite de l'Exercice

À vous ! Évaluez les expressions :

1. `*ptr++`
2. `++&data`

7.7.2 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `*ptr++` et passez à la Sec. 7.7.3.

7.7.3 Mise en Commun de l'Évaluation de `*ptr++`

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

Pour rappel, la variable `ptr` est déclarée comme suit :

```
1 int data, x, *ptr = &x;
```

Cela signifie que `ptr` contient l'adresse de `x` (peu importe où se trouve `ptr` en mémoire).

L'opérateur d'incrément est à droite de l'opérande. Cela signifie donc que l'expression entière est évaluée à l'opérande (i.e., `*ptr`). L'incrément se fait ensuite mais n'intervient pas dans l'évaluation de l'expression. On peut donc ne pas en tenir compte ici.

Il vient donc qu'il suffit d'évaluer `*ptr`. Sachant que `ptr` contient l'adresse de `x`, soit 204, il suffit d'aller voir ce qu'il y a à l'adresse mémoire 204 : la valeur 336.

L'expression `*ptr++` est donc évaluée à 336.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `++&data`
6. `ptr + data`
7. `&data - ptr`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.8 ++&data

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.8.2](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne vous souvenez plus des priorités des opérateurs, voyez la Section [7.5.1](#)
- Si l'opérateur d'incrémentatation reste un mystère, voyez la Section [7.7.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.8.1](#)

7.8.1 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `++&data` et passez à la Sec. 7.8.2.

7.8.2 Mise en Commun de l'Évaluation de ++&data

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

L'opérateur d'incrément est à gauche de l'opérande. Cela signifie donc que l'expression entière est évaluée à la valeur correspondant à l'incrément de l'opérande (**&data**). Attention, l'incrément ne consistera pas, ici, à faire une banal "+1". Il faut garder en mémoire que nous manipulons ici des pointeurs (i.e. `texttt&data`). Dès lors, et puisque la variable **data** est déclarée comme un **int**, l'incrément d'une unité revient à dire "incrément d'un espace mémoire permettant le stockage d'une valeur entière", soit 4 bytes (cfr. **énoncé**).

L'évaluation se fait donc de la manière suivante (les parenthèses sont rajoutées uniquement pour donner une idée de l'ordre) : **++(&data)**. L'opération d'incrément fonctionne ici car l'**opérateur de déréférencement** permet d'avoir une valeur à droite.

Commençons par **&data**. Ici, l'**opérateur de référencement** permet d'obtenir l'adresse de la variable **data**. Soit 336.

C'est ensuite cette valeur 336 qu'il faut incrémenter d'un espace mémoire permettant le stockage d'une valeur entière (i.e., 4 bytes). On obtient finalement la valeur 340.

L'expression **++&data** est donc évaluée à 340.

Suite de l'Exercice

Évaluez les expressions :

1. ***data**
2. ****data**
3. **&*data**
4. ***&x**
5. ***ptr++**
6. **ptr + data**
7. **&data - ptr**
8. ***(x = data)**
9. **F[-4] - *(E + 2)**

7.9 ptr + data

Si vous voyez de quoi on parle, rendez-vous à la Section [7.9.2](#)
Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.9.1](#)

7.9.1 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Attention, dans le cadre de l'expression `ptr + data`, il faut bien tenir compte du type de chaque expression/variable. Envisagez les choses sous l'angle de l'arithmétique des pointeurs. . .

Suite de l'Exercice

À vous ! Évaluez l'expression `ptr + data` et passez à la Sec. 7.9.2.

7.9.2 Mise en Commun de l'Évaluation de `ptr + data`

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

Pour rappel, la variable `ptr` est déclarée comme suit :

```
1 int data, x, *ptr = &x;
```

Cela signifie que `ptr` contient l'adresse de `x` (peu importe où se trouve `ptr` en mémoire). `ptr` est donc de type "pointeur sur entiers" et `data` de type "entier". On ne peut donc pas additionner directement le contenu de `ptr` (qui est l'adresse de `x - 204`) avec celui de `data` (qui est un entier quelconque - 208).

On est bien, ici, dans le cas d'une arithmétique des pointeurs, en particulier l'addition. Par rapport à la **formule générale**, `ptr` correspond à `p` et `data` à `i`. Il suffit alors d'appliquer la formule donnée ci-dessus pour trouver la solution.

Il vient donc : `ptr + data*4`, où 4 est la taille (en bytes) nécessaire au stockage d'une valeur entière (cfr. **énoncé**). On obtient donc le calcul : $204 + (208 \times 4)$. Soit 1036.

L'expression `ptr + data` est donc évaluée à 1036.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `&data - ptr`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.10 &data - ptr

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.10.2](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.10.1](#)

7.10.1 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Envisagez les choses sous l'angle de l'arithmétique des pointeurs...

Suite de l'Exercice

À vous ! Évaluez l'expression `&data - ptr` et passez à la Sec. 7.10.2.

7.10.2 Mise en Commun de l'Évaluation de `&data - ptr`

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
<code>data :</code>	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
<code>x :</code>	204	336
	⋮	⋮

Pour rappel, la variable `ptr` est déclarée comme suit :

```
1 int data, x, *ptr = &x;
```

Cela signifie que `ptr` contient l'adresse de `x` (peu importe où se trouve `ptr` en mémoire). `ptr` est donc de type "pointeur sur entiers" et `data` de type "entier". On va devoir ici soustraire l'adresse de `data` (i.e., 336) avec la valeur contenue dans `ptr` (i.e., une adresse – 204). Pour rappel, les valeurs entières sont stockées sur 4 bytes (cfr. [énoncé](#)).

On est bien, ici, dans le cas d'une arithmétique des pointeurs, en particulier la soustraction de pointeurs. Par rapport à la [formule générale](#), `&data` correspond à `p2` et `ptr` à `p1`. On applique donc la formule et il vient : $\frac{336 - 204}{4}$. Soit 33.

L'expression `&data - ptr` est donc évaluée à 33.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `*(x = data)`
9. `F[-4] - *(E + 2)`

7.11 *(x = data)

- Si vous voyez de quoi on parle, rendez-vous à la Section 7.11.3
- Si la notion de pointeur vous pose problème, voyez la Section 7.3.1
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section 7.1
- Si vous ne vous souvenez plus de la priorité des opérateurs, voyez la Section 7.5.1
- Si l'opérateur d'affectation (=) vous pose question, voyez la Section 7.11.1
- Si vous ne voyez pas comment faire, reportez-vous à l'indice 7.11.2

7.11.1 Rappels sur l'Opérateur d'Affectation

L'opérateur d'*affectation* (=) est probablement l'opérateur le plus important car il permet la sauvegarde des informations en mémoire (i.e., dans une variable). Il se présente de la façon suivante :

```
1 var = expr
```

L'opérateur d'affectation permet de stocker le résultat de l'expression `expr` (appelée aussi *valeur à droite*) dans la variable `var` (appelée aussi *valeur à gauche*). Il est évident que le type du résultat de l'évaluation de la valeur à droite doit être cohérent avec le type de la valeur à gauche. On ne stocke pas des poires dans des abricots!

La **priorité des opérateurs** indique bien que l'évaluation de l'expression se fait de droite (la valeur à droite) à gauche (la valeur à gauche). Dès lors, après l'affectation, l'expression entière devient égale à la valeur affectée. Le fonctionnement général de l'affectation est illustré à la Fig. 1.

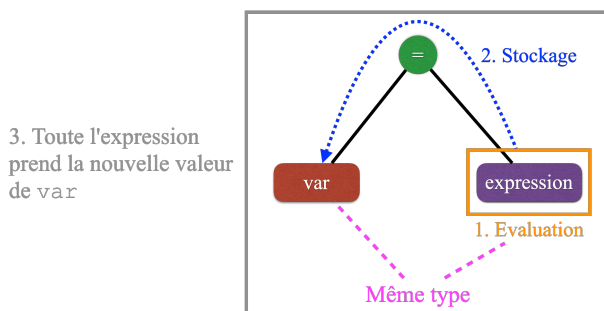


FIGURE 1 – Illustration du fonctionnement général de l'affectation.

Alerte : Risque de confusion

Il est très fréquent que les étudiants confonde l'opérateur d'*affectation* (=) et l'opérateur d'*égalité* (==). Le premier permet de stocker le résultat d'une expression dans une variable (et l'expression complète est évaluée à la valeur stockée), tandis que le deuxième permet de tester l'égalité entre deux expressions (le résultat de l'expression est alors booléen, i.e., soit *vrai*, soit *faux*). La confusion entre les deux opérateurs peut amener à des situations problématiques, en particulier dans le cadre des structures de contrôle (cfr. Chap. 2).

Une expression classique qui utilise l'opérateur d'affectation est l'incrément (ou la décrémentation) d'une variable.⁵ Par exemple :

```
1 x = x + 2
```

Dans ce cas, l'expression va stocker dans la variable `x` l'ancienne valeur de `x` augmentée de 2.

Le langage C fournit un raccourci (on parle de *sucré syntaxique*⁶) pour ce genre d'expression qui permet de combiner en un seul opérateur l'affectation et l'opération arithmétique (i.e., incrément ou décrémentation).

5. *Incrémenter* signifie *augmenter* une valeur. À l'inverse, *décrémenter* signifie diminuer une valeur.

6. Le sucre syntaxique est une extension de la syntaxe d'un langage de programmation afin de le rendre plus agréable à lire et à écrire, sans changer son expressivité.

Le sucre syntaxique pour l'incrémentation/décrémentation se présente de la façon suivante :

1 `var α = expr`

où $\alpha \in \{+, -, *, /, \%\}$.

Ce raccourci est équivalent à

1 `var = var α expr`

L'évaluation de l'expression se fait comme pour une affectation normale.

Attention, l'utilisation du sucre syntaxique ajoute, implicitement, des parenthèses autour de `expr`. Il faut donc comprendre l'expression comme suit :

1 `var = var α (expr)`

Par exemple :

1 `x += 2`

est équivalent à

1 `x = x + 2`

Suite de l'Exercice

À vous ! Évaluez l'expression `*(x = data)` et passez à la Sec. [7.11.3](#).

7.11.2 Indice

Repartez de l'état initial de la mémoire. Sur cet état, représentez graphiquement les manipulations effectuées par l'expression. Il vous suffit, ensuite, de “suivre” les flèches.

Suite de l'Exercice

À vous ! Évaluez l'expression `*(x = data)` et passez à la Sec. 7.11.3.

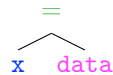
7.11.3 Mise en Commun de l'Évaluation de `*(x = data)`

Soit l'état initial de la mémoire suivant :

	⋮	⋮
	340	27
data :	336	208
	⋮	⋮
	220	3
	216	17
	212	5
	208	220
x :	204	336
	⋮	⋮

L'expression `*(x = data)` va faire une **opération de déréférencement** sur le résultat de l'expression `x = data`.

Il faut donc commencer par évaluer `x = data`. Il s'agit d'une expression basée sur l'opérateur **d'affectation**. L'évaluation se fait de la façon suivante :



Il faudra d'abord évaluer la valeur à droite (cfr. le **rappel sur l'affectation**). Celle-ci est constituée uniquement de la variable **data**. Pour évaluer cette expression, il suffit de connaître la valeur qu'elle contient, soit 208.

Ce résultat est ensuite placé dans la valeur à gauche, i.e., la variable **x**. L'entièreté de l'expression `x = data` est alors évalué à la nouvelle valeur de **x**, soit 208.

Cette valeur 208 est alors déréférencée (opérateur `*`). Ici, il suffit de regarder ce qu'il y a à la case mémoire d'adresse 208, soit la valeur 220.

L'expression `*(x = data)` est donc évaluée à 220.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `&data - ptr`
9. `F[-4] - *(E + 2)`

7.12 $F[-4] - *(E + 2)$

- Si vous voyez de quoi on parle, rendez-vous à la Section [7.12.2](#)
- Si la notion de pointeur vous pose problème, voyez la Section [7.3.1](#)
- Si vous n'êtes pas à l'aise avec la notion d'expression, voyez la Section [7.1](#)
- Si vous ne voyez pas comment faire, reportez-vous à l'indice [7.12.1](#)

7.12.1 Indice

Assurez-vous de bien comprendre le lien entre **tableaux et pointeurs**. C'est la clé pour évaluer cette expression.

Suite de l'Exercice

À vous ! Évaluez l'expression $F[-4] - *(E + 2)$ et passez à la Sec. **7.12.2**.

7.12.2 Mise en Commun de l'Évaluation de $F[-4] - *(E + 2)$

Pour rappel, les variables F et E sont déclarées comme suit :

```
1 char *E = "Session", *F = E + 7;
```

L'endroit où se trouve ces variables en mémoire n'a aucune importance. L'état initial de la mémoire, tel que décrit dans l'énoncé n'a aucune importance ici. Par contre, on peut représenter graphiquement l'état de la mémoire pour la variable E. Comme, ici, les adresses en mémoire n'ont pas d'importance, nous choisissons de ne pas les indiquer. Le schéma indique juste le nom des variables et les valeurs associées (on n'oublie pas de rajouter le terminateur à la chaîne de caractères). Il vient :

	:	:
E + 7 :	:	'\0'
E + 6 :	:	'n'
E + 5 :	:	'o'
E + 4 :	:	'i'
E + 3 :	:	's'
E + 2 :	:	's'
E + 1 :	:	'e'
E :	:	'S'
	:	:

Puisque F est initialisé à E+7, il apparaît clairement que *F vaut '\0'.

Intéressons nous d'abord à $F[-4]$. Nous savons qu'il existe un lien entre pointeurs et tableaux. L'expression $F[-4]$ peut se réécrire $*(F-4)$. Comme F est initialisé à E+7, on peut remplacer, dans l'expression, F par E+7. L'expression devient : $*(E+7-4)$. Soit $*(E+3)$, ou encore $E[3]$. L'évaluation donne donc le caractère 's'.

Passons maintenant à $*(E+2)$. Grâce au lien entre pointeurs et tableaux, on peut réécrire cette expression comme suit : $E[2]$. L'évaluation donne donc le caractère 's'.⁷

Au final, l'expression $F[-4] - *(E + 2)$ donne 's' - 's'. Peu importe la valeur entière qui représente le caractère 's', le résultat donnera 0.

L'expression $F[-4] - *(E + 2)$ est donc évaluée à 0.

Suite de l'Exercice

Évaluez les expressions :

1. `*data`
2. `**data`
3. `&*data`
4. `*&x`
5. `*ptr++`
6. `++&data`
7. `ptr + data`
8. `&data - ptr`
9. `*(x = data)`

7. L'évaluation ici aurait pu se faire directement sur base de la représentation graphique.

7.13 Synthèse de toutes les Expressions

<code>*data</code>	220
<code>**data</code>	3
<code>&*data</code>	208
<code>*&x</code>	336
<code>*ptr++</code>	336
<code>++&data</code>	340
<code>ptr + data</code>	1036
<code>&data - ptr</code>	33
<code>*(x = data)</code>	220
<code>F[-4] - *(E + 2)</code>	0