Opérateurs et expressions

1. Introduction

Pour l'essentiel un programme consiste à demander au processeur de faire des calculs à partir de données contenues dans des variables (plus ou moins simple ou organisées) et à en ranger le résultat dans d'autres.

Ces calculs sont organisés par des instructions de contrôle qui permettent de choisir les données, ordonner les calculs et ranger et présenter les résultats.

Le C++ considère que en dehors des déclarations et des instructions de contrôle tout est expression, il faut donc bien comprendre comment s'écrivent et s'interprètent les opérations pour pouvoir comprendre et écrire des programmes.

Une façon de le comprendre est de voir que ceci est une instruction du C++ :

```
expression ;
```

Une expression quelconque terminée par un point-virgule est une instruction.

Cela permet de comprendre que si
```
i++
```
est une expression alors
```
i++ ;
```
est une instruction d'incrémentation.

Conséquence :
```
12 ;
```
est une instruction valide du C++ ! Heureusement ou malheureusement elle ne fait rien !

Mais c'est la source de beaucoup d'erreur dans les programmes !

2. Expression

Qu'est-ce qu'une expression ?

Dans le dictionnaire Larousse on trouve comme définition d'expression au sens informatique :
Suite d'identificateurs de variables ou de fonctions, de constantes et de symboles opératoires représentant, dans un programme, un calcul à effectuer. (Une *expression arithmétique* ne contient que des variables et des opérations arithmétiques ; une *expression logique* ou *booléenne,* que des variables et des opérateurs logiques.)

En savoir plus sur http://www.larousse.fr/dictionnaires/francais/expression/32326#yTYb3QbKUWvq8Z70.99
On peut retenir de cette définition que puisque un calcul aboutit forcément à un résultat, une expression représente au final une valeur.

Bien sûr une expression doit être correctement écrite pour être comprise par un ordinateur, mais là n'est pas le propos.

On peut aussi définir une expression par une définition récursive :

- une expression peut être **simple**, c'est à dire composée d'un seul « élément » : une valeur **constante** ou une **variable**,
- une expression peut être **complexe**, c'est à dire composée d'**expressions** reliées par des **opérateurs**. Les expressions intervenant dans une expression plus complexe sont appelées des **sous-expressions**.

Avec ces définitions voici quelques exemples d'expression
expression simples :

```
0   1   3.14    true    NULL'C' "Bonjour"   i   prenom
```

expression complexes :

```
5+9 sin(0)  i + 1   2 * x       i < N && T[i] != 0
```

Valeur d'une expression

Puisqu'une expression représente une valeur, tout le problème est de savoir calculer cette valeur ou plutôt de savoir comment l'ordinateur va calculer cette valeur.
Dans le cas d'une expression simple, il n'y a pas de calcul à faire et la valeur est facile à trouver, il s'agit de la valeur de la constante ou de la variable. Attention dans le cas d'une variable celle-ci aura toujours une valeur mais qui pourra être indéterminée si la variable n'a pas été initialisée.
Exemples :
l'expression 1 vaut … la valeur 1 !
si `i` est une variable, l'expression `i` vaut … la valeur de `i` !
Dans le cas d'une expression complexe contenant un seul opérateur, la valeur est (presque toujours) facile à déterminer :
l'expression `3 + 7` vaut `10`.
l'expression `T[i]` vaut la valeur de la case numéro `i` du tableau `T`. Attention c'est la (i + 1) ème case !

Dans le cas d'une expression complexe contenant plusieurs opérateurs, l'ordre dans lequel ceux-ci seront appliqués au calcul prend une grande importance. La connaissance de cet ordre est donc primordial.
Cet ordre repose sur 3 points :

- **présence de parenthèses** : une sous-expression entre parenthèses est calculée avant le calcul de l'expression la contenant, ainsi 3 * (4 + 5) vaut 3 * 9 = 27 et non pas 12 + 5 = 17,

- **priorité des opérateurs** intervenant dans l'expression : chaque opérateur appartient à une classe de priorité et dans une expression contenant 2 opérateurs de priorités différentes c'est la sous-expression de l'opérateur qui a la plus forte priorité qui est évaluée d'abord, ainsi la multiplication ayant une priorité supérieure à l'addition l'expression : 2 + 3 * 4 vaut 14 et pas 20,

- **sens d'associativité** : chaque classe de priorité possède un sens d'associativité qui indique le sens d'évaluation dans le cas d'expression contenant 2 opérateurs de même priorité. Dans la plupart des cas le sens d'associativité est gauche-droite (→), on verra plus loin un exemple de sens d'associativité droite-gauche (←).

3. Les opérateurs

Les expressions étant formées par l'application d'opérateurs à des sous-expressions nous allons étudier tous les opérateurs au travers des caractéristiques précédentes : priorité, sens d'associativité et les expressions formées au travers des caractéristiques suivantes que peuvent présentées les expressions formées :

Caractéristiques d'une expression

Une expression possède toujours ces 2 caractéristiques :

- **Valeur** : une expression **vaut toujours** quelque chose :

toutes les expressions possèdent une valeur, si c'est une expression simple, c'est la valeur de la constante ou de la variable, si c'est une expression complexe, c'est la valeur résultant de l'application de l'opérateur à son (ou ses) opérande(s) ;

- 2021 pour l'expression : 2021
- 10 pour l'expression : 2 * 3 + 4
- true pour l'expression 2 * 3 + 4 == 10

- **Type** : une expression est **toujours typée**, elle possède un type au sens du c++ :

le type d'une expression est le type de sa valeur :

- entier pour l'expression simple 1965,
- flottant pour l'expression complexe 18 * 0.07
- flottant pour l'expression cos(0)
- booléen pour N < 100

Une expression peut de plus posséder ces 2 caractéristiques :

- **Adresse** : une expression **peut représenter une adresse** (PAS toujours) :

une expression peut représenter une adresse de la mémoire de la machine, afin d'en modifier éventuellement le contenu :
- si `compteur` est une variable, l'expression simple `compteur` représente l'adresse en mémoire de `compteur`,
- si `Tablo` est un tableau, l'expression complexe `Tablo[62]` représente l'adresse en mémoire de la case numéro 62 de `Tablo` (la 63e case donc!).

On appelle ce genre d'expression, une expression qui représente une adresse, une ***lvalue***.
Plus précisément ce nom vient de la syntaxe habituelle d'une affectation :
X = Y
où, pour qu'elle soit valide, l'expression X doit désigner un emplacement mémoire pour pouvoir y ranger la valeur de l'expression Y.
On parle de valeur gauche ou ***left value***. Pour la même raison, l'expression Y est appelée ***Rvalue***.

- **Effet de bord** : une expression **peut faire** quelque chose (PAS toujours) :

la caractéristique la plus intéressante d'une expression C. L'évaluation d'une expression dans le déroulement d'un programme peut entraîner une modification d'un de ses opérandes, comme par exemple dans le cas de l'expression booléenne :
`Tablo[i++] < 3` cette expression vaut *`true`* ou *`false`* suivant la valeur de `Tablo[i]`,
mais de plus, suite à son évaluation, **la valeur de `i` sera augmentée de 1**.

Table de priorité des opérateurs et sens d'associativité

Priorité	Opérateur(s)	Sens de l'associativité
15	( ) [ ] . ->	→
14	! ~ ++ -- $+_1$ $-_1$ $*_1$ $\&_1$ sizeof (*type*)	←
13	$*_2$ / %	→
12	$+_2$ $-_2$	→
11	<< >>	→
10	< <= > >=	→
9	== !=	→
8	$\&_2$	→
7	^	→
6	|	→
5	&&	→
4	||	→
3	? :	←
2	= *= /= %= += -= <<= >>= &= ^= |=	←
1	,	→

Le chiffre en indice indique l'arité de l'opérateur en cas d'utilisation multiple :
par exemple $-_1$ correspond au moins unaire (-3) et $-_2$ au moins binaire (i-1).

Les opérateurs

Dans l'ordre de la table précédente sauf quelques exceptions décrites à la fin !
Dans tous les cas les arguments sont des sous-expressions notées E ou E_i, qui peuvent avoir certaines contraintes.

[ ] : Indexation

Accès à un objet dont l'adresse est calculée à partir d'une adresse de base et d'un déplacement entier.
Utilisation courante accès à une case d'un tableau.

Contraintes : 2 opérandes E_1 et E_2 : une adresse et un entier
Syntaxe : $E_1[E_2]$ (voir remarque plus bas)
Valeur : la valeur qui se trouve à l'adresse A + i * (la taille mémoire d'une case)
Type : le type d'un case
Adresse : en fonction du tableau
Effet : non

Remarque amusante mais pas forcément à utiliser :
En fait la norme du C++ n'indique pas quel argument doit être placé devant et quel argument doit être placé entre les crochets ! Ainsi si T est un tableau et i un entier i[T] correspond à la même chose que T[i] et de même avec une constante : 3[T] est tout à fait valide.

. : Sélection

Accès à un champ d'une classe, d'une structure ou d'une union.
Nécessite 2 arguments de types différents : un objet (d'une classe, d'une structure ou d'une union) et un membre de cet objet.

Contraintes : 2 opérandes E_1 un objet et E_2 un membre de l'objet
Syntaxe : $E_1.E_2$
Valeur : la valeur du membre de l'objet
Type : le type du membre de l'objet
Adresse : oui (sauf si tableau)
Effet : non

-> : Sélection par un pointeur - Accès à un champ d'une classe, d'une structure ou d'une union à partir d'un pointeur

Accès à un champ d'une classe, d'une structure ou d'une union par un pointeur.
E -> m est Équivalent à (*E).m

Contraintes : 2 opérandes E_1 un pointeur sur un objet et E_2 un membre de l'objet
Syntaxe : E_1 -> E_2
Valeur : la valeur du membre de l'objet pointé
Type : le type du membre de l'objet pointé
Adresse : oui (sauf si tableau)
Effet : non

! : Non booléen (ou logique)

Sa table de vérité est bien connue.
À noter la compatibilité entier/booléen :
0 est équivalent à *false* et n'importe quelle valeur différente de 0 est équivalente à *true*.

Contraintes : 1 opérande E d'un type compatible booléen
Syntaxe : ! E
Valeur : *true* ou *false*
Type : bool
Adresse : non
Effet : non

~ : Complément à 1

Attention cette opération repose sur le codage de l'opérande, sa portabilité n'est pas assurée.

Contraintes	: 1 opérande entier E1
Syntaxe	: ~ E1
Valeur	: la valeur décodée après avoir inversé tous les bits de l'opérande
Type	: celui de l'opérande
Adresse	: non
Effet	: non

++ : Incrémentation

Il existe 2 versions de cet opérateur unaire, l'un préfixé c'est à dire écrit avant son opérande et l'autre postfixé, c'est à dire écrit après son opérande préfixé : ++E, postfixé : E++
Les deux versions diffèrent par la valeur de l'expression formée, les autres caractéristiques sont identiques :

Préfixé		**Postfixé**	
Contraintes	: 1 opérande E supportant l'incrémentation	Contraintes	: 1 opérande E supportant l'incrémentation
Syntaxe	: ++E	Syntaxe	: E++
Valeur	**: E**	**Valeur**	**: E + 1**
Type	: type de E	Type	: type de E
Adresse	: non	Adresse	: non
Effet	**: E est incrémenté : E = E + 1**	**Effet**	**: E est incrémenté : E = E + 1**

Exemple :

Avec ces déclarations	`int i ;` `int x = 2 ;`	
	`i = ++x ;`	`i = x++ ;`
Valeur de i	3	2

Les 2 opérateurs sont équivalents du point de vue de leur opérande, c'est la valeur de l'expression, mise en évidence par son affectation à i, qui est différente.

-- : Décrémentation

Cet opérateur fonctionne exactement comme le précédent en remplaçant + par -, +1 par -1 et incrémentation par décrémentation (et 3 par 1 dans l'exemple !).

+ : Plus unaire

Opérateur "neutre" qui permet d'écrire par exemple +x au lieu de x.

Contraintes : 1 opérande numérique E
Syntaxe : + E
Valeur : celle de l'expression
Type : celui de l'expression
Adresse : non
Effet : non

- : Moins unaire

Cet opérateur change le signe de l'expression.

Contraintes : 1 opérande numérique E
Syntaxe : - E
Valeur : l’opposé de l'expression
Type : celui de l'expression
Adresse : non
Effet : non

* : Indirection ou déréférencement

Cet opérateur permet d'accéder à un objet pointé.

Contraintes : 1 opérande qui doit être une adresse (***lvalue***)
Syntaxe : * E
Valeur : la valeur de l'objet pointé par E
Type : le type des objets pointés par E
Adresse : oui
Effet : non

& : Adresse de

Cet opérateur permet d'obtenir l'adresse d'un objet.

Contraintes : 1 opérande qui doit posséder une adresse (***lvalue***)
Syntaxe : & E
Valeur : l'adresse de E
Type : il s'agit d'un type adresse (ou pointeur) : adresse de (type de E)
Adresse : non
Effet : non

*** / % + -** : Opérations arithmétiques

Dans l'ordre : multiplication, division, modulo (reste de la division entière), addition et soustraction.
On note que les 2 derniers sont de priorité inférieure.

Contraintes : 2 opérandes E_1 et E_2
Syntaxe : E_1 @ E_2 en notant @ un opérateur parmi {*, /, %, +, -}
Valeur : le résultat de l'opération appliqué à E_1 et E_2 dans cet ordre
Type : le type le plus "grand" (cf. conversion)
Adresse : non
Effet : non

<< : Transfert vers un flux de sortie

Cet opérateur envoie une valeur vers un flux de sortie, fichier ou écran par exemple.

Contraintes : 2 opérandes E_1 flux de sortie et E_2 type "affichable"
Syntaxe : E_1 << E_2
Valeur : E_1
Type : E_1
Adresse : oui
Effet : E_1 a reçu E_2

L'interprétation d'une expression de type :
`cout << a << " et " << b << endl ;`
se fait en suivant le sens d'associativité de l'opérateur ainsi :
`( ( (cout << a) << " et ") << b) << endl ;`
et s'appuie sur les caractéristiques au-dessus :
la sous -expression
`cout << a`
est évaluée d'abord, ce qui provoque l'affichage de a et vaut l'opérande de gauche cout
l'expression
`cout << " et "`
est évaluée ensuite de la même façon
puis
`cout << b`
et enfin
`cout << endl ;`

>> : Extraction d'un flux d'entrée

Cet opérateur extrait une valeur d'un flux de sortie, fichier ou clavier par exemple.

Contraintes : 2 opérandes E_1 flux d'entrée et E_2 ***lvalue*** de type "saisissable"
Syntaxe : E_1 >> E_2
Valeur : E_1
Type : E_1
Adresse : oui
Effet : E_1 est prêt à fournir la valeur suivante

On comprend grâce au sens de l'associativité le fonctionnement de :

```
cin >> a >> b
```

qui peut est évaluée :

```
( ( cin >> a ) >> b )
```

la première saisie est envoyée dans a et la deuxième dans b.

<< et >> : Décalage bit à bit vers la gauche et vers la droite

Ces opérateurs ont conservé leur sens initial en provenance du C, il s'agit d'opérations qui travaillent sur les bits du premier opérande en les décalant vers la gauche ou vers la droite du nombre de fois indiqué par le deuxième opérande.
Les bits sortants sont perdus, les bits entrants sont 0 ou le bit de signe.
Arithmétiquement le décalage d'un chiffre correspond à une multiplication (gauche) ou à une division (droite) par la base.
Ces opérations ne sont pas portables.

Contraintes	: 2 opérandes E_1 et E_2 de type entier
Syntaxe	: E_1 << E_2 ou E_1 >> E_2
Valeur	: E_1 décalé E_2 fois à gauche ou à droite
Type	: E_1
Adresse	: non
Effet	: non

== != < <= > >= : Comparaison

Opérateurs de comparaison courants égal, différent, inférieur, inférieur ou égal, supérieur, supérieur ou égal

Contraintes : 2 opérandes E_1 et E_2 supportant la coparaison
Syntaxe : E_1 @ E_2 en notant @ un opérateur parmi {==, !=, <, <=, >, >=}
Valeur : *true* ou *false*
Type : bool
Adresse : non
Effet : non

& ^ | : Et, Xor, Ou bit à bit

Opérateurs bit à bit réalisant un ET, un XOR, un OU sur les bits des opérandes.
Par exemple avec des entiers sur un octet a = 103 (01100111) et b = 85 (01010101)

```
          01100111              01100111              01100111
          01010101              01010101              01010101
a&b :     01000101     a^b:     00110010     a|b :    01110111
              69                    50                    119
```

Ces opérations ne sont pas portables

Contraintes : 2 opérandes E_1 et E_2 entiers
Syntaxe : E_1 @ E_2 en notant @ un opérateur parmi {&, ^, |}
Valeur : la valeur obtenue calcul des différents bits
Type : entier
Adresse : non
Effet : non

&& || : Et et Ou booléen

Opérateurs logiques (utilisés dans des tests)
Remarque : le compilateur a quelques connaissances en logique élémentaire et connaît les tables de vérité compacte de ces 2 opérateurs :

E_1	E_1 && E_2	E_1 || E_2
false	*false*	E_2
true	E_2	*true*

Par exemple dans l'évaluation de
E_1 && E_2
l'ordre d'évaluation est fixé, E_1 est évaluée d'abord,
si elle est *false*, alors la valeur de l'expression complète est *false* et E_2 n'est pas évaluée
sinon E_2 est évaluée et donne sa valeur à l'expression.

2 avantages à cette façon dévaluer :

- efficacité : si une évaluation n'est pas utile elle n'est pas faite,
- sécurité de certains tests :
  soit l'exemple suivant d'une boucle de recherche de valeur nulle dans un tableau T de taille N :

```
i = 0 ;
while (i < N && T[i] != 0)
  i++ ;
```

si le tableau ne contient pas de valeur nulle, la boucle va s'arrêter lorsque i < N sera fausse, c'est à dire quand i sera égale à N,
dans ce cas il y aurait un risque de débordement de tableau dans l'expression T[i], le plus grand indice d'un tableau de taille N étant N – 1,
mais comme cette sous-expression n'est pas évaluée il n'y a pas de débordement.

Bien sûr si le test est écrit en inversant les sous-expression, le risque de débordement persiste.

Contraintes : 2 opérandes E_1 et E_2 interprétables comme booléens
Syntaxe : E_1 && E_2, E_1 || E_2
Valeur : ET, OU logique entre les opérandes
Type : bool
Adresse : non
Effet : non

= : Affectation

Cet opérateur permet de donner une valeur à une variable. Il illustre bien le fait que (presque) tout est expression en C et son comportement s'appuie sur l'effet de bord qui le caractérise.

Contraintes : 2 opérandes E_1 et E_2 de types compatibles, celui de gauche doit être une ***lvalue*** et celui de droite est une ***rvalue***
Syntaxe : $E_1 = E_2$
Valeur : E_2 converti par le type de E_1
Type : E_1
Adresse : non
Effet : E_2 est affectée de E_1

Le fait de considérer l'affectation comme une expression permet d'écrire :

```
a = b = c = d = 1
```

qui doit se comprendre d'après le sens d'associativité de l'opérateur :

```
a = ( b = ( c = ( d = 1) ) )
```

avec donc évaluation d'abord de

```
d = 1
```

qui a pour effet d'affecter 1 à d et qui a pour valeur l'opérande de droite (1),
ce qui permet d'évaluer

```
c = 1
```

de la même façon en affectant 1 à c,
et de la droite vers la gauche d'affecter 1 à b puis à a.

*= /= %= += -= <<= >>= &= ^= |= : Autres affectations

Il s'agit d'affectations avec application d'un opérateur, plus précisément :
E_1 et E_2 étant 2 expressions et @ un opérateur, E_1 @= E_2 est équivalent à E_1 = E_1 @ E_2.

Contraintes : 2 opérandes E_1 et E_2 de types compatibles, celui de gauche doit être une ***lvalue*** et celui de droite est une ***rvalue***
Syntaxe : E_1 @= E_2
Valeur : E_1 @ E_2
Type : E_1
Adresse : non
Effet : E_1 est affectée de E_1 @ E_2

i += 2 est équivalent à i = i + 2

j *= 10 est équivalent à j = j * 10

etc.

Remarque : optimisation

Ces opérateurs ont été créés pour accélérer le programme, en effet dans i = i * 2, il faut aller consulter 2 fois i dans la mémoire alors que dans i *= 2 le compilateur peut s'apercevoir facilement qu'une fois suffit.

En pratique les compilateurs modernes optimisent correctement ces expressions, il n'est plus utile de s'en servir.

, : Évaluation en séquence

Cet opérateur garantit l'ordre d'évaluation de 2 expressions

Contraintes : 2 opérandes E_1 et E_2
Syntaxe : E_1 , E_2
Valeur : E_2
Type : E_2
Adresse : non
Effet : non
Lors de l'évaluation de E_1 , E_2 la première expression (E_1) est évaluée avant la seconde (E_2).

Utilisation principale :
L'instruction for prévoit 3 expressions dans sa syntaxe : initialisation, test de continuité, incrémentation.
Si on souhaite avoir plusieurs initialisations ou plusieurs incrémentations l'opérateur **,** peut servir à les séparer :

```
for (i = 0, j = 100 ; i < N ; i++, j /= 2)
    …
```

Attention, cela n'a pas de sens d'utiliser la virgule dans la 2e expression du if (test) :

`i < N , j != 0` vaut **`j != 0`**

( ) : Appel de fonction

Bonne illustration du fait que tout est vu comme une expression, un appel de fonction est donc l'application d'une fonction à des arguments par l'opérateur ( ).

E_1, E_2, E_3, …, E_n étant des expressions, $E_1(E_2, E_3, …, E_n)$ est l'appel de la fonction E_1 avec les arguments E_i.

E_1 doit être une fonction, les E**i** sont les arguments effectifs de la fonction.

Contraintes	: E_1 une fonction, E_2, E_3, …, E_n des expressions
Syntaxe	: $E_1(E_2, E_3, …, E_n)$
Valeur	: valeur de retour de la fonction
Type	: type de retour de la fonction
Adresse	: selon le type de retour de la fonction
Effet	: selon le comportement de la fonction

sizeof : Taille mémoire d'un objet

Cet opérateur calcule la taille de l'espace mémoire occupé par un type, en byte (octet en général), avec sizeof (char) égal à 1.
2 versions de cet opérateur :
- avec un type entre parenthèses comme opérande

Contraintes : un opérande qui doit être un type
Syntaxe : sizeof (*type*)
Valeur : taille en mémoire d'un objet du type *type*
Type : size_t
Adresse : non
Effet : non

- avec une expression sans parenthèses comme opérande

dans ce cas l'expression n'est pas évaluée, c'est son type qui intéresse l'opérateur

Contraintes : un opérande expression E
Syntaxe : sizeof E
Valeur : taille en mémoire d'un objet du type de la valeur de 2 (non calculée)
Type : size_t
Adresse : non
Effet : non

Remarques
- si l'expression est un tableau statique (déclaré avec le nombre d'éléments exprimé entre crochets), sizeof calcule l'encombrement total du tableau, ce qui permet de calculer le nombre d'éléments d'un tableau statique par exemple avec :

```
int T[120] ;
sizeof T / sizeof T[0]
```
vaut 120 !

- le type de `sizeof` est un type entier, mais dépend du système (machine/processeur/système d'exploitation et version). Il s'appelle `size_t` et est défini dans la librairie `<cstddef>`. Il correspond au type entier capable de représenter la taille de n'importe que objet.

(*type*) : Cast – changement de type

Encore l'opérateur parenthèses avec une syntaxe différente, il sert à convertir un type dans un autre.

Contraintes : un type *type* et une expression E
Syntaxe : (*type*) E
Valeur : la valeur de E exprimée dans le type *type*
Type : *type*
Adresse : non
Effet : non

Attention toutes les conversions ne sont pas possibles et certaines donnent des résultats inutiles.

? : : Expression conditionnelle

Un "genre" de `if … then … else` façon expression.

Contraintes : 3 expressions E_1, E_2 et E_3, la première interprétable comme un booléen
Syntaxe : E_1 ? E_2 : E_3
Valeur : E_2 ou E_3
Type : E_2 ou E_3
Adresse : possible
Effet : non (pas du fait de l'opérateur **? :**)

L'évaluation de cette expression se fait ainsi dans cet ordre :

- l'expression E_1 est évaluée puis
- si elle est vraie — l'expression vaut E_2 et E_3 n'est pas évaluée
- sinon (E_1 est fausse) — l'expression vaut E_3 et E_2 n'est pas évaluée

Remarque sur l'ordre d'évaluation des expressions

Il y a en fait très peu de situation où l'ordre d'évaluation des sous-expressions est garanti :

- les opérateurs **&&** et **||**
- l'opérateur **,**
- l'opérateur **? :**

Dans tous les autres cas le langage ne spécifie rien (ce qui signifie qu'il peut décider de l'ordre d'évaluation des sous-expressions pour des raisons d'efficacité par exemple). Ainsi un cas classique indécidable :
si A et B sont 2 tableaux et i un entier :

```
int A[N], B[N], i
```

On peut écrire pour copier le tableau B dans le tableau A :

```
i = 0;
while (i < N)
    A[i] = B[i++];
```

l'expression `A[i] = B[i++]` est problématique :

- pour pouvoir faire l'affectation, il faut que l'adresse de A[i] et la valeur de B[i++] soit déterminées,
- pour déterminer B[i++], il faut calculer i++

Avec i = 3 :

Si le compilateur commence par A[i] on a :
- adresse de A[i] est adresse de A[i]
- valeur de B[i++] est valeur de B[3]
- i = 4

A[3] = B[3]

Si le compilateur commence par B[i++] on a
- valeur de B[i++] est valeur de B[3]
- i = 4
- adresse de A[i] = adresse de A[4]

A[4] = B[3]