Langage C

Les expressions

L2 Mathématique et Informatique

Université de Marne-la-Vallée

Expressions en C

- Elles sont la base de l'écriture des instructions: si on ajoute un **;** après une expression, on forme une instruction;
- Leur évaluation suit des règles de priorité précises;
- Une expression est formée à partir de constantes et d'identificateurs à l'aide d'**opérateurs**;
- Elles peuvent se combiner entre elles (et devenir illisibles).

Une expression

- possède un **type**
  (évaluation au moyen des règles de typage)
- détermine une **valeur**
  (évaluation en respectant les priorités des opérateurs)
- peut avoir une **action**
  (effet de bord)

```
1   short h = 2; int x, y, i = 10;
2       x = y = 3 * h + i;
3   /* expression de type int, qui vaut 16, et qui
4      attribue aux variables x et y la valeur 16.*/
```

Construction d'une expression

- une constante est une expression
- un identificateur est une expression
- si *exp1* et *exp1* sont des expressions et *op* un opérateur alors *exp1 op exp2* est une expression.

Le compilateur admet un mélange d'objets des divers types arithmétiques dans une expression, en particulier dans une affectation, mais effectue des conversions implicites lors de l'évaluation

- en l'absence de `unsigned`, les objets sont convertis dans le type le plus fort selon l'ordre décroissant :
  `long double, double, float, int`.
- les règles pour les opérandes `unsigned` dépendent de l'implémentation
- la conversion en `unsigned` ne se fait qu'en cas de nécessité
- **promotion entière** : l'arithmétique se fait au moins en `int`, c'est-à-dire que `char` et `short` sont promus en `int`
- **attention**, suivant l'implémentation un `char` peut être promu en un entier négatif !

Coercition

Lors de l'évaluation, on peut forcer (*cast* en anglais) le type d'un élément. Celui-ci est alors vu comme étant du type précisé

```
1    printf("%g  %g \n",(double)(5/2),(double)5/2);
```

de réels affiche `2  2.5`.
Le format `%g` est un format économique d'affichage de réels: le nombre minimum de chiffres est utilisé.

Liste des opérateurs

Préc.↘	OPÉRATEUR	SYMBOLE	EXEMPLE	ASSOC.
	indexation	`[]`	`t[i][j]`	
	champ de struct.	`. ->`	`c.reel`	`->`
	appel de fonct.	`()`	`max(t)`	
	adresse	`&`	`&i`	
	indirection	`*`	`*p`	
	taille	`sizeof`	`sizeof i`	
	coercition	`(`*type*`)`	`(float) p`	
logique	NON logique	`!`	`!(i==2)`	`<-`
bit à bit	NON bit à bit	`~`	`~066`	
arithmé-tiques	pl/moins unaires	`+ -`	`-2`	
	incrémentation	`++`	`++i, i++`	
	décrémentation	`--`	`--i, i--`	
	multiplication	`*`	`a*b`	`->`
	division/modulo	`/ %`	`a/b`	
	addition/soustr.	`+ -`	`a+b a-b`	`->`
bit à bit	décalages	`<< >>`	`x>>n`	`->`
compa-rateurs	inégalités	`< <= > >=`	`a<3`	`->`
	égalité/différ.	`== !=`	`i==0`	`->`
bit à bit	ET bit à bit	`&`	`a & mask`	`->`
	OU excl. b. à b.	`^`	`a^3777`	`->`
	OU bit à bit	`|`	`a|b`	`->`
logi-ques	ET logique	`&&`	`a&&b`	`->`
	OU logique	`||`	`a||b`	`->`
	expr. condition.	`?:`	`x>0?x:-x`	`<-`
	affect. simple	`=`	`i=j=2`	`<-`
	affect. étendue	`+= *= ...`	`i+=2`	
	exp. composée	`,`	`i=2, g=6`	`->`

Précédence et associativité

- les opérateurs unaires sont prioritaires sur les opéateurs binaires.
- les trois opérateurs arithmétiques multiplicatifs (`* / %`) ont tous la même précédence, plus forte que celle des deux opérateurs additifs (`+ -`)

`-a*b+c` s'interprète comme `((-a)*b)+c`

- à l'intérieur d'une même classe de précédence les opérateurs s'associent en général de gauche à droite, sauf les opérateurs unaires et les opérateurs d'affectation

`a/b*c` s'interprète comme `(a/b)*c`,
et non comme `a/(b*c)`
`- sizeof i` s'interprète comme `-(sizeof i)`
`x = y = z` s'interprète comme `x = (y = z)`

- les parenthèses forcent la précédence ou l'associativité
- l'ordre d'évaluation des opérandes dépend de l'implémentation, sauf pour ces quatre opérateurs : | `\|\|` | `&&` | `?:` | `,` | dont l'interprétation se fait de gauche à droite.

Opérateurs arithmétiques

- il y a différentes arithmétiques
  - dans les entiers (signés ou non)
  - dans les flottants
  - sur les pointeurs
- opérateurs

		entiers	flottants	pointeurs
plus unaire	+	√	√	
moins unaire	-	√	√	
incrémentation	++	√	√	√
décrémentation	--	√	√	√
multiplication	*	√	√	
division	/	entière	√	
modulo	%	√		
addition	+	√	√	√
soustraction	-	√	√	√

Les opérateurs d'incrémentation ++ et --

S'appliquent à une expression désignant un objet en mémoire ("`L-value`") pouvant être changé.

- le type : celui de l'élément
- la valeur: dépend de la position de l'opérateur
- l'effet de bord : incrémente ou décrémente (*quand?*)

expression	`++i`	`i++`	`--i`	`i--`
valeur	`i+1`	`i`	`i-1`	`i`
valeur de `i` **après** évaluation de **toute** l'expression où il apparait	`i+1`		`i-1`	

A manipuler avec méfiance

```
1  i=3;
2  printf("%d␣",i++ + i++);
3  /* ici seulement on est sur de la valeur de i */
4  printf("%d",i);
```

affiche `6 5`

Opérateur d'affectation =

- le type : celui de l'élément affecté
- la valeur: celle de l'élément affecté
- l'effet de bord : change la valeur de l'élément affecté

l'expression `a=3` a pour valeur 3. Ceci permet :

- de combiner affectation et test

```
1 while(((lettre=getchar())!=EOF)
```

la valeur de l'affectation (entre −1 et 255 si lettre est de type `int`) est comparée à EOF(−1).

- d'effectuer des affectations multiples : `b=a=3`, équivaut à b=(a=3) (associativité de droite à gauche)
  `a=3` vaut 3, valeur affectée à `b`.
  Attention ne pas mélanger les types

```
1 int n,p;
2 unsigned char a;
3 n=a=p=256;
4 printf("n=%d␣a=%d␣p=%d\n",n,a,p);
```

affichage `n=0 a=0 p=256`

Affectations étendues

Pour tous les opérateurs arithmétiques et les opérateurs bit à bit, *op*, on peut utiliser la forme:
`exp1` *op*`=` `exp2` à la place de `exp1=(exp1)`*op* `exp2`

- `exp1` n'est évaluée qu'une fois;
- gagne en lisibilé si `exp1` est une expression compliquée.

Expressions booléennes

Pas de types booléens en C.

- Toute valeur non nulle est interprétée comme `Vrai` si elle est utilisée comme booléen.
- La valeur 0 est interprétée comme `Faux` si elle est utilisée comme booléen.
  Même si l'interprétation est identique, préférer un test explicite.
  `if(truc!=0)` est plus lisible que `if(truc)`.
- une expression logique est de type `int` et a pour valeur 1 si le résultat est `Vrai`, et 0 si le résultat est `Faux`. Utile pour les valeurs de retour (`return i!=j;`

Opérateurs logiques

NON logique	!
ET logique	&&
OU logique	||

La condition $a \leq x \leq b$ se traduit par l'une des expressions:
`a <= x && x <= b` ou `!(a>x || b<x)`
Les opérandes des expressions logiques sont évaluées de gauche à droite de façon paresseuse.
ET logique

e1	e2	e1 && e2
$= 0$	non évaluée	$= 0$
$\neq 0$	évaluée	$= 1$ si e2 $\neq 0$ $= 0$ si e2 $= 0$

OU logique

e1	e2	e1 || e2
$\neq 0$	non évaluée	$= 1$
$= 0$	évaluée	$= 1$ si e2 $\neq 0$ $= 0$ si e2 $= 0$

- Utile pour controler l'indice d'un tableau

```
1  while(i<taille && t[i]!=x)
```

- Dangereux avec des expressions à effets de bord

```
1  int i=0,j=0;
2  if(i++!=0 && j++!=0)
3     printf("%d␣%d␣\n",i,j);
4  if(i++!=0 && j++!=0)
5     printf("%d␣%d␣\n",i,j);
6  if(i++!=0 && j++!=0)
7     printf("%d␣%d␣\n",i,j);
8  /*c'est scandaleux d'ecrire un code pareil!*/
```

Expression conditionnelle

exp1 ? *exp2* : *exp3*

- si *exp1* est VRAIE, la **valeur** et le **type** sont ceux de *exp2*, et *exp3* n'est pas évaluée
- si *exp1* est FAUSSE, la **valeur** et le **type** sont ceux de *exp3*, et *exp2* n'est pas évaluée

Pratique si utilisé avec modération, mais difficile à lire.

```
1 abs = x>=0 ? x : -x;
2 max = x>y ? x : y;
3 printf("Il␣y␣a␣%d␣element%s", n, n>1 ? "s" : "");
```

Horrible en cas d'abus

```
1 max = x>y ? x>z ? x : z : y>z ? y : z;
```

Le parenthèsage est bienvenu:

```
1 abs = (x>=0 ? x : -x);
```

Composition

exp1, *exp2*, ..., *expn*

```
1    i=1, j=2, k=i*j+1;
```

- les expressions sont évaluées de la gauche vers la droite
- la valeur et le type de l'expression composée est celle de la dernière expression évaluée, celle de droite
- usage à réserver essentiellement pour les boucles `for`

```
1    for(i=1, j=2; condition(i,j)!=0; i++,j--){
```

Règles et pièges

- préférer la clarté à la concision. Ce n'est pas parce que c'est écrit en peu d'instructions que ce sera compilé en un programme plus efficace, mais ça sera plus dur à maintenir.
- attention aux effets de bord, en particulier pour ++ et --
- l'ordre d'évaluation des opérandes n'est fixé que pour 4 opérateurs! Toute expression qui dépend de l'ordre d'évaluation des opérandes doit être jugée incorrecte

```
1  t[i] = i++
2  (x = 3) * x
3  f(&x) + g(x)
```

Une variable soumise à un effet de bord ne doit pas apparaitre ailleurs dans la même expression.