

Expressions et instructions



<https://xkcd.com/303/>

L-valeurs et R-valeurs

Affectation

```
op_gauche = op_droit // l'expression  
// vaut op_droit
```

Doit être une L-valeur

Soit une L-valeur
ou une R-valeur

- L-valeur = variable, élément de tableau
- R-valeur = constante, expression arithmétique

2 + x = 12 // erreur! (2 + x) n'est pas une L-valeur

Effets secondaires

Side effects

- Quand une opération modifie l'environnement
- Exemple: modification de l'opérande de gauche par l'opérateur d'affectation

7 + (b = 3)

7 + (b = 3)

7 + 3 // et b vaut 3

10 // et b vaut 3

L'opérateur d'affectation

Assignment operator

a = b = c = 7

a = b = c = 7

a = b = 7 // et c vaut 7

a = b = 7 // et c vaut 7

a = 7 // et b vaut 7, c vaut 7

7 // et a vaut 7, b vaut 7, c vaut 7

- Par contre, on ne peut pas écrire

(a = 5) = b // Erreur!

car (a = 5) s'évalue à 5 qui n'est pas une L-value

Affectation combinée

- On peut combiner des opérations et des affectations

`+=, *=, -=, /=, %=`

`// admettons que int a vaut 2`

`a += 7`

`a = a + 7`

`a = a + 7`

`a = 2 + 7 // car a vaut 2`

`9 // et a vaut 9 = 2 + 7`

Opérateur préfixe d'incrémentation

++u

- Opérateur unaire appliqué à une **L-valeur**
- Effet secondaire: il augmente de 1 le contenu de la **L-valeur**
- L'expression vaut la valeur après l'incrément

```
// admettons que int u vaut 7  
int v;
```

```
v = ++u // u vaut 7+1 = 8  
v = 8
```

Opérateur postfixe d'incrémentation

`u++`

- Opérateur unaire appliqué à une **L-valeur**
- Effet secondaire: il augmente de 1 le contenu de la **L-valeur**
- L'expression vaut la valeur **avant** l'incrément

```
// admettons que int u vaut 7  
int v;
```

```
v = u++ // u vaut 7+1 = 8  
v = 7
```

Opérateurs d'incrémentation

```
int u = 7;  
int v;
```

Préfixe	Postfixe
<pre>v = ++u;</pre>	<pre>v = u++;</pre>
<pre>// u vaut 7+1 = 8 // v vaut 8</pre>	<pre>// u vaut 7+1 = 8 // v vaut 7</pre>

Opérateurs de décrémentation

```
int u = 7;  
int v;
```

Préfixe

```
v = --u;
```

```
// u vaut 7-1 = 6  
// v vaut 6
```

Postfixe

```
v = u--;
```

```
// u vaut 7-1 = 6  
// v vaut 7
```

Le type booléen

Vrai ou faux

- Historiquement en C il n'y a pas de **type** booléen (“vrai” ou “faux”)
- Une **valeur** de **type** numérique peut être interprétée comme “vrai” ou “faux”:

*Si une **valeur** est égale à **zéro**, alors on l'interprète comme “**faux**” ❌, toute autre **valeur non-nulle** compte comme “**vrai**” ✅.*

- On utilise souvent des entiers, avec **0** pour “faux” et **1** pour “vrai”
- Depuis C99 il existe le type `_Bool` et depuis C23 il existe le type `bool`

Les opérateurs logiques

NON

$\neg a$

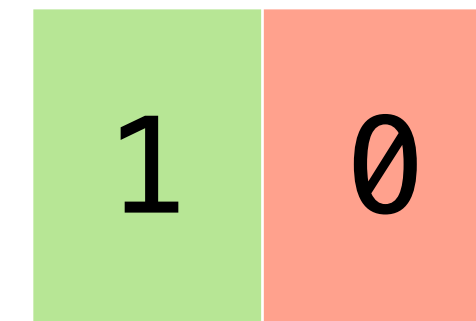
$\neg 1$ vaut 0

$\neg 0$ vaut 1

NON logique

a	0	1
-----	-----	-----

$\neg a$	1	0
----------	-----	-----



Les opérateurs logiques

OU

a || b

0 || 0 vaut 0

1 || 0 vaut 1

0 || 1 vaut 1

1 || 1 vaut 1

OU logique

a || b 0 1

0	0	1
1	1	1

Les opérateurs logiques

ET

`a && b`

`0 && 0` vaut `0`

`1 && 0` vaut `0`

`0 && 1` vaut `0`

`1 && 1` vaut `1`

ET logique

<code>a && b</code>	<code>0</code>	<code>1</code>
<code>0</code>	<code>0</code>	<code>0</code>
<code>1</code>	<code>0</code>	<code>1</code>

Conversion implicite vers booléen

- Dans une opération booléenne la valeur de chaque opérande est transformée
 - en 0 si elle est égale à 0
 - en 1 si elle est différente de 0
- ! est évalué en premier
- ⚠ && a la priorité sur ||

Exemples

`!10` vaut `0`

`10` est non-nul, donc "vrai", donc `non-10` est "faux", donc `0`

`100 && 0 || 1 && 0`

`0 || 1 && 0`

`0 || 0` qui vaut `0`

`"bonjour" && -1`

`1 && 1` vaut `1`

Opérateur d'égalité ==

- Teste l'égalité de deux valeurs

```
10 == 10 // vaut 1 (vrai)
```

```
int a = 6, b = 7;
```

```
a == b // vaut 0 (faux)
```

```
a == b - 1 // vaut 1 (vrai)
```


Opérateur d'inégalité !=

`(expr1) != (expr2)`

🧐 C'est équivalent à `!((expr1) == (expr2))`

`7 != 8 // vrai`

`1 != 1 // faux`

Opérateurs de comparaison

>, <, >=, <=

10 >= 10 // vaut 1 (vrai)

10 < 100 // vaut 1 (vrai)

-100 >= 5 // vaut 0 (faux)

<= : "Plus petit ou égal"

>= : "Plus grand ou égal"

Instructions simples

et composées

DCT, 2024.01

Les instructions

Statements

- Les **instructions** sont des fragments de code qui contrôlent l'exécution du programme
- Une **instruction simple** est composée d'une **expression** `<expression>;`
- L'expression elle-même peut être complexe = calculs, appels de fonctions, affectations, etc.
- Une **déclaration** est aussi une **instruction**, par exemple:

```
int a = 41;
```

Les instructions composées

Compound statements

- Une **instruction composée** ou un **bloc** (*compound statement*, *block*) est une suite d'**instructions** et de **déclarations** entourée d'accolades

```
{  
  <déclaration> | <instruction>  
  ...  
  <déclaration> | <instruction>  
}
```

- Elle peut elle-même contenir des **instructions composées**
- La **définition** de la fonction `main` est une **instruction composée**

```
{  
  int test = 8;  
  {  
    printf("C'est une instruction composée "  
          "dans une instruction composée\n");  
    test = 12;  
  }  
  printf("test vaut %d\n", test);  
  // Affiche: test vaut 12  
}
```

Portée d'une variable

Block scope

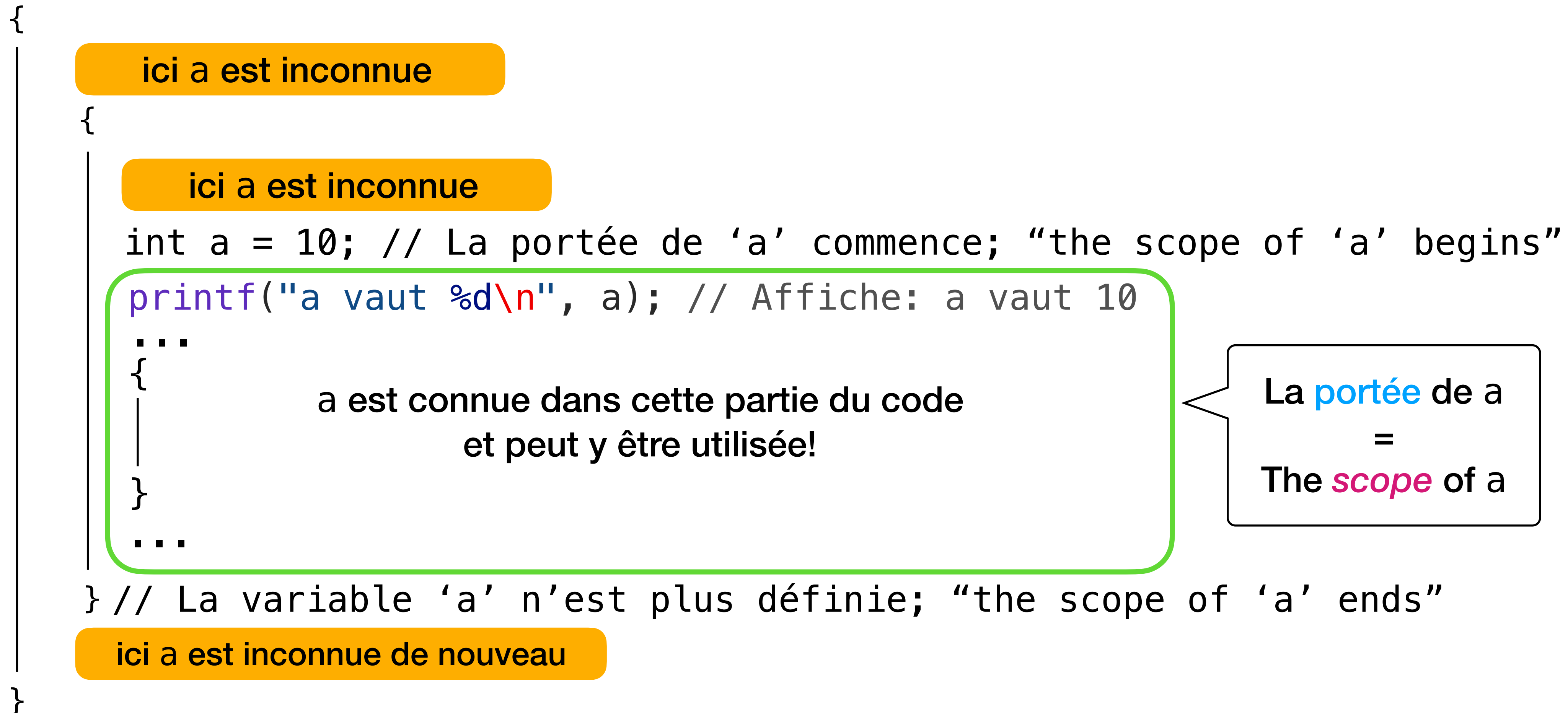
- Que se passe-t-il quand on définit une **variable** à l'intérieur d'un **bloc** ?

```
{
  int a = 8;
  {
    int b = 13;
    printf("a vaut %d\n", a); // Affiche: a vaut 8
    printf("b vaut %d\n", b); // Affiche: b vaut 13
  }
  printf("b vaut %d\n", b); // Erreur: b n'est pas définie!
}
```

- Chaque **identifiant/variable** a une **portée (scope)** = après sa définition dans le **bloc** où elle est définie, et y compris dans les **blocs** imbriqués

Portée d'une variable

Block scope



Instructions conditionnelles

DCT, 2024.01

Choix

- Jusqu'ici toutes les instructions de la fonction `main` sont exécutées
- Nous voulons pouvoir choisir quel code on exécute selon l'entrée
- Par exemple:
 - Validation de l'entrée — on veut un nombre positif, mais l'utilisateur donne un nombre négatif 😞
 - Confirmation — “Voulez-vous vraiment effacer `C:\Windows\System32` ?”
 - ???

L'instruction `if`

`if` *statement*

- Exécuter une instruction **uniquement si** une condition est vraie

```
if (<expression>) instruction-vrai
```

- *Souvenez-vous*: `0` signifie “faux”, toute autre valeur non-nulle signifie “vrai”
- On peut aussi spécifier l’alternative avec la clause “else”:

```
if (<expression>) instruction-vrai else instruction-faux
```

- Une **instruction composée** est fortement encouragée, mais une instruction simple fonctionne aussi

Welcome to the Matrix

```
char pill;
printf("Which pill do you want to take?"
      " (red = r or blue = b) \n");
scanf("%c", &pill);
if (pill == 'r')
{
    printf("Welcome to the desert of the real\n");
}
else
{
    printf("You are back in the Matrix. Sad.\n");
}
```



Welcome to the Matrix

Ligne active	Expression à évaluer	pill vaut
7-8	printf("Which...	?
9	scanf("%c", &pill);	'r'
10	if (pill == red)	'r'
	// vrai -> 1ere branche	
12	printf("Welcome...	'r'
	// On sort du if	
	Fin du programme	

```
6: char pill;
7: printf("Which pill do you want to take?"
8:         " (red = r or blue = b) \n");
9: scanf("%c", &pill);
10: if (pill == 'r')
11: {
12:     printf("Welcome to the desert of the real\n");
13: }
14: else
15: {
16:     printf("You are back in the Matrix. Sad.\n");
17: }
```

```
> ./matrix
Which pill do you want to take? (red = 0 or blue = anything else)
r
Welcome to the desert of the real.
```

Enchaîner plusieurs if

```
printf("Vous voulez connaitre le titre d'un livre du Seigneur des Anneaux ?\n");
int volume;
scanf("%d", &volume);
if (volume == 1)
{
    printf("La Communauté de l'Anneau.\n");
}
else if (volume == 2)
{
    printf("Les Deux Tours.\n");
}
else if (volume == 3)
{
    printf("Le Retour du Roi.\n");
}
else
{
    printf("Je ne connais pas ce volume.\n");
}
```

L'opérateur ternaire

a ? b : c

- Signifie: “si **a** est vrai, alors l'expression a la valeur **b**, sinon elle a la valeur **c**”
- Exemple 1: calculer la valeur absolue d'un entier

```
int x;  
scanf("%d", &x);
```

```
int abs_x = x < 0 ? -x : x;
```

- Exemple 2: calculer le max de deux nombres

```
int x, y;  
scanf("%d %d", &x, &y);
```

```
int max = x < y ? y : x;
```

Opérateur d'égalité ==

... pour les réels

```
double somme = 0.1 + 0.7;

if (somme == 0.8)
{
    printf("somme == 0.8\n");
}
else
{
    printf("somme != 0.8\n");
}
// Affiche:
// somme != 0.8

printf("%.20lf\n", 0.7);
// affiche:
// 0.699999999999999999995559
```

⚠ Attention aux nombres réels — ce sont des représentations *approximatives*, il ne faut pas compter sur l'égalité...

Opérateur d'égalité ==

... pour les réels

Fonctions mathématiques

```
#include <math.h>
```

```
...  
if (fabs(somme - 0.8) < 0.0000001)  
{  
    printf("somme == 0.8\n");  
}  
else  
{  
    printf("somme != 0.8\n");  
}  
// Affiche:  
// somme == 0.8
```

✓ On teste si $| \text{somme} - 0.8 | < \varepsilon$
avec la fonction valeur absolue fabs

Opérateur d'égalité ==

... pour les tableaux / strings

```
char salut[] = "salut";  
  
if (salut == "salut")  
{  
    printf("C'est la même chose\n");  
}  
else  
{  
    printf("Ce n'est pas la même chose\n");  
}  
// Affiche:  
// Ce n'est pas la même chose
```

⚠ Ne fonctionne pas pour des tableaux, donc ni pour des strings...

— il faut tester élément par élément (dans une boucle)

Mises en garde



- Pourquoi?
- Le point-virgule après la condition du `if`
- L'instruction vide `;` est valide
- Peut-être le compilateur vous le fera remarquer, mais ne comptez pas dessus!

```
int cinq = 5;
if (cinq > 10);
{
    printf("Cinq est plus grand que dix.\n");
}
// Affiche "Cinq est plus grand que dix."
```

```
[build] values.c:13:19: warning: if statement has empty body [-Wempty-body]
[build]     if (cinq > 10);
[build]                   ^
```

Mises en garde



- Pourquoi?
- Dans la condition de l'**if** il y a une affectation, pas un test d'égalité
- `(cinq = 10)` est une expression de valeur 10 qui en booléen donne 1=vrai
- Peut-être le compilateur vous le fera remarquer, mais ne comptez pas dessus!

```
int cinq = 5;
if (cinq = 10)
{
    printf("Cinq est égal à dix.\n");
}
// Affiche "Cinq est égal à dix."
```

```
[build] values.c:19:14: warning: using the result of an assignment as a
condition without parentheses [-Wparentheses]
[build]     if (cinq = 10)
[build]           ~~~~~^~~~~
```