Introduction à la Programmation des Algorithmes

2.1. Langage C – Opérateurs booléens, conditions

François Fleuret
https://fleuret.org/11x001/

Nous avons vu comment écrire un programme composé d'expressions, chacune terminée par un ; et que l'ordinateur évalue successivement.

```c
#include <stdio.h>

int main(void) {
  int x;
  x = 15;
  printf("Pour commencer x=%d\n", x);
  /* Là je fais un calcul idiot */
  x = x * 3 - 5;
  printf("Et après un calcul sans interet x=%d\n", x);
  return 0;
}
```

Pour le moment les opérations que l'ordinateur exécute sont toujours les mêmes.

Il est impératif de pouvoir faire dépendre les opérations à exécuter des valeurs à traiter.

Par exemple si nous voulons programmer

$$\forall n \in \mathbb{N}, f(n) = \begin{cases} \frac{1}{2}n & \text{si } n \in 2\mathbb{N} \\ 3n+1 & \text{sinon} \end{cases}$$

cela serait difficile avec les opérateurs que nous avons vus.

Les langages de programmation offrent des mécanismes pour contrôler dynamiquement la séquence d'instructions à exécuter.

On fait référence à cette séquence comme étant le **flux** du programme, et à sa modulation comme le **contrôle du flux.**

Opérateurs relationnels et logique booléenne

Le **calcul booléen** est une branche des mathématiques qui formule la logique de manière algébrique.

Les deux valeurs possibles sont 1 pour "vrai" et 0 pour "faux".

Les opérations standard sont:

- La négation: $\text{NOT}(0) = 1$, $\text{NOT}(1) = 0$.
- Le et: $\text{AND}(0,0) = 0$, $\text{AND}(0,1) = 0$, $\text{AND}(1,0) = 0$, $\text{AND}(1,1) = 1$.
- Le ou: $\text{OR}(0,0) = 0$, $\text{OR}(0,1) = 1$, $\text{OR}(1,0) = 1$, $\text{OR}(1,1) = 1$.

Note: Ce sont ces opérations fondamentales qui sont combinées pour construire les opérations arithmétiques.

Par ex. si on ajoute deux nombres binaires à un chiffre a et b, on obtient un nombre à deux chiffres $\text{AND}(a, b)$ *et* $\text{AND}(\text{OR}(a, b), \text{NOT}(\text{AND}(a, b)))$.

Le langage C n'a pas de type de donnée spécifique pour les valeurs booléennes et utilise pour cela le type `int`.

Un opérateur qui calcule un résultat booléen produira la valeur 0 ou la valeur 1, et un opérande booléen peut être de n'importe quel type numérique et sera interprété comme "faux" s'il est nul et "vrai" sinon.

Le C met à notre disposition deux groupes d'opérateurs pour exprimer des conditions booléennes:

- Les **opérateurs relationnels** combinent des opérandes de types numériques et produisent des résultats booléens, et
- les **opérateurs booléens** combinent des opérandes booléens et produisent des résultats booléens.

Les **opérateurs relationnels** sont

`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

et sont tous de priorité identique.

```c
int a, b;
a = 3 < 4;
printf("a=%d\n", a);
b = 3 > 4;
printf("b=%d\n", b);
```

affiche

```
a=1
b=0
```


Les **opérateurs booléens** sont, par ordre de priorités:

`!`	négation logique (opérateur unaire)
`&&`	et logique
`||`	ou logique

```c
int x, y;
x = 0;
y = 1;
printf("%d\n", x && y);
printf("%d\n", x || y);
printf("%d\n", x && !y);
printf("%d\n", !x && y);
```

affiche

```
0
1
0
1
```



```c
int a, b, c, d;
a = 1;
b = 0;
c = 0;
d = a && (!a || b || c);
printf("d=%d", d);
```

affiche

```
d=0
```

Les opérateurs ont la priorité

```
!
&&
||
=
```

ce qui nous donne par substitution (avec `a=1`, `b=0` et `c=0`):

```
d = a && (!a || b || c)
d = 1 && (!a || b || c)
d = 1 && (!1 || b || c)
d = 1 && (!1 || 0 || c)
d = 1 && (!1 || 0 || 0)
d = 1 && (0 || 0 || 0)
d = 1 && (0 || 0)
d = 1 && (0)
d = 1 && 0
d = 0
```

`0` (et `d` a été modifiée!)

Ces opérateurs peuvent être combinés avec les opérateurs arithmétiques, d'affectation, et d'affectations composées que nous avons vus précédemment.

Nous pouvons résumer les priorités des opérateurs que nous avons déjà vus en les rangeant par ordre décroissant. Deux opérateurs sur la même ligne ont même priorité, auquel cas ils sont évalués de gauche à droite.

```
()
++ -- - (unaire)

* / %
+ - (binaire)
< <= > >=
== !=
&&
||
= += -= *= /= %=
```

```c
int x, y, q;
x = 3;
y = 5;
q = 3 * x - 10 < 2 * x - y;
printf("q=%d\n", q);
```


Les opérateurs ont la priorité

*

- (binaire)

<

=

ce qui nous donne par substitution (avec `x=3` et `y=5`):

```
q = 3 * x - 10 < 2 * x - y
q = 3 * 3 - 10 < 2 * x - y
q = 3 * 3 - 10 < 2 * 3 - y
q = 3 * 3 - 10 < 2 * 3 - 5
q = 9 - 10 < 2 * 3 - 5
q = 9 - 10 < 6 - 5
q = -1 < 6 - 5
q = -1 < 1
q = 1
```

`1` (et `q` a été modifiée!)

Comme indiqué précédemment pour les opérateurs booléens: **tout opérande nul est interpreté comme "faux" et tout opérande non-nul comme "vrai".**

Ceci est donc valide en C:

```
float a = 3.1415926, b = -5;
int q = a && b;
```

La variable q vaudra 1.

Le laxisme du C permet aussi

```
int ouch = 3 < 4 < 2;

printf("%d\n", ouch);
```

qui affiche 1.

En effet l'évaluation de gauche à droite donne

```
ouch = 3 < 4 < 2
ouch = 1 < 2
ouch = 1
```

Clauses

Nous allons voir dans la suite des **instructions de contrôle du flux** qui permettent de spécifier dynamiquement si certaines parties du programme doivent être ignorées ou répétées.

Nous appellerons **clause** un bout de programme qui est:

- une expression terminée par un `;`, ou
- une instruction de contrôle du flux, ou
- un bloc entre `{}`, contenant plusieurs clauses.

Cette définition est donc **récursive**.

Des exemples de clauses:

```
a = 3;
```

```
{
  a = 3;
  b = a + 2;
}
```

```
{
  a = 3;
  {
    c = a;
    a++;
  }
  b = a + 2;
}
```

Traditionnellement, on indente un programme en décalant le contenu d'une clause de plusieurs caractères vers la gauche pour faciliter la lecture.

Instruction `if/else`

L'instruction principale de contrôle de flux qui permet de définir un comportement **conditionnel** en C est le `if`, qui a la forme suivante:

```
if(condition)
  clause_si_vrai
```

où `condition` définit la condition qui doit être vraie, et calcule un résultat qui prend la valeur "vrai" ou "faux".

Si ce résultat est "vrai" alors `clause_si_vrai` est évaluée.

La condition doit impérativement être entre parenthèses,

Par exemple

```
1  if(n < 0)
2    n = -n;
```

ou

```
1  if(x == 0) {
2    printf("x est nul!");
3    x = 1;
4    nb_de_trucs_nuls++;
5  }
```

Il est possible de rajouter dans un `if` une clause à évaluer dans le cas où la condition est fausse avec `else`:

```
if(condition)
  clause_si_vrai
else
  clause_si_faux
```


Par exemple si nous voulons programmer

$$\forall n \in \mathbb{N}, f(n) = \begin{cases} \frac{1}{2}n & \text{si } n \in 2\mathbb{N} \\ 3n+1 & \text{sinon} \end{cases}$$

nous pouvons faire

```
if(n % 2 == 0)
  f_de_n = n / 2;
else
  f_de_n = 3 * n + 1;
```

Il est aussi possible de rajouter autant de clauses additionnelles si la clause du `else` est elle même un `if`::

```
if(condition_1)
  clause_1
else if(condition_2)
  clause_2
  ...
else if(condition_k)
  clause_k
else
  clause_defaut
```

Une clause est exécutée si et seulement si toutes les conditions qui la précèdent sont fausses à l'exception de celle juste avant qui doit être vraie.

La clause par défaut après le `else` final est évaluée si toutes les conditions sont fausses.

Par exemple nous pouvons faire

```
1  if (n == 0)
2    printf("n est nul");
3  else if(n % 2 == 0)
4    printf("n est pair et non-nul");
5  else
6    printf("n est impair");
```

qui est équivalent à

```
1  if (n == 0)
2    printf("n est nul");
3  else {
4    if(n % 2 == 0)
5      printf("n est pair et non-nul");
6    else
7      printf("n est impair");
8  }
```