Introduction à la Programmation des Algorithmes

1.4. Langage C – Opérateurs et expressions

François Fleuret

https://fleuret.org/11x001/

UNIVERSITÉ
DE GENÈVE

Une expression est une séquence d'opérateurs et d'opérandes qui définit un calcul à effectuer. Une opérande est un des arguments d'un opérateur.

Par exemple

```
3 + 2 * x
```

est une expression valide en C qui combine les opérandes `3`, `2`, et `x`, avec les opérateurs `+` et `*`.

Une expression est une séquence d'opérateurs et d'opérandes qui définit un calcul à effectuer. Une opérande est un des arguments d'un opérateur.

Par exemple

```
3 + 2 * x
```

est une expression valide en C qui combine les opérandes `3`, `2`, et `x`, avec les opérateurs `+` et `*`.

Étant données les règles standard de priorités, nous comprenons cette expression comme

```
3 + ( 2 * x )
```

Contrairement aux expressions dans un langage naturel, celles dans un langage de programmation ont un sens (une “sémantique”) exacte, qui correspond à la suite d'instructions de langage machine que l'ordinateur exécutera.

La conception de langages, de compilateurs et d'interpréteurs est un vaste domaine de recherche, et nous n'allons qu'effleurer les concepts utiles pour comprendre le C.

Tokens

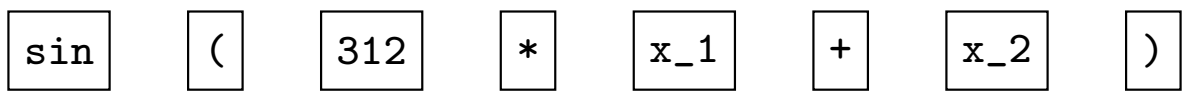
Un programme est un fichier composés de caractères qui sont regroupés en *tokens*, qui sont des composants élémentaires, eux-mêmes groupés en expressions.

$$\text{Caractères} \rightarrow \text{tokens} \rightarrow \text{expressions} \Rightarrow \text{langage machine}$$

Par exemple ce bout de C

```
y = sin( 312 * x_1 + x_2 )
```

sera décomposé en la suite de tokens suivante:



Et cette suite de token est interprétée par le compilateur comme un calcul à effectuer:

1. Calcule `312 * x_1`,
2. ajoute `x_2`,
3. applique `sin`.

Note: En interne le CPU combine un nombre fixe de variables et une "pile" où stocker les résultats intermédiaires s'il y en a beaucoup.

Il y a en C six familles de tokens:

- les signes de ponctuation (parenthèses, crochets, virgules),
- les identifiants (noms de variables ou de fonctions),
- les mots-clés,
- les constantes (valeurs numériques),
- les chaînes de caractères,
- les opérateurs.

Il y a en C six familles de tokens:

- les signes de ponctuation (parenthèses, crochets, virgules),
- les identifiants (noms de variables ou de fonctions),
- les mots-clés,
- les constantes (valeurs numériques),
- les chaînes de caractères,
- les opérateurs.

Nous allons voir des sous-ensembles de ces catégories, et nous les compléterons au fur et à mesure du cours.

En plus des tokens que le compilateur reconnaît, on peut rajouter dans un fichier source des **commentaires** que le compilateur ignore.

Il suffit de les placer entre les symboles `/*` et `*/`. Un commentaire peut être sur plusieurs lignes.

```
1  #include <stdio.h>
2
3  int main(void) {
4    /* Je déclare une variable pour y stocker le
5       nombre de tours */
6
7    int nombre_de_tours;
8
9    /* Mais c'est un programme idiot qui ne fait rien ! */
10
11   return 0;
12 }
```

Un **identifiant** est un label composé de chiffres, lettres minuscules, lettres majuscules et du "underscore", c'est à dire le caractère `'_'`. Un identifiant ne peut pas commencer par un chiffre.

Exemples:

- `n`
- `x0`
- `nombre_de_voitures`
- `compteurDeTours`
- `_tmp`

Les **mots-clés** sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identifiants. L'ANSI C en compte 32:

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Nous en verrons plusieurs dans ce cours.

Les **constantes** sont des valeurs numériques qui apparaissent littéralement dans le programme.

Il y a de nombreux points de syntaxe que nous n'allons pas détailler et nous nous limiterons aux entiers, par ex. `34`, ou `87934`, et aux valeurs à virgules, possiblement en notation scientifique, par ex. `3.1415926`, `-45.0`, ou `7.8e9`.

Les **constantes** sont des valeurs numériques qui apparaissent littéralement dans le programme.

Il y a de nombreux points de syntaxe que nous n'allons pas détailler et nous nous limiterons aux entiers, par ex. `34`, ou `87934`, et aux valeurs à virgules, possiblement en notation scientifique, par ex. `3.1415926`, `-45.0`, ou `7.8e9`.

Le compilateur considère qu'une constante numérique sans `.` est de type entier, et qu'une avec est de type à virgule.

Une constante entière qui commence par `0` est interprétée comme étant notée en base 8.

Les **chaînes de caractères** sont des suites de caractères qui peuvent être manipulées littéralement, c'est à dire que le compilateur ne donne pas de sens particulier à ce qu'elles contiennent.

Une chaîne de caractères est délimitée dans le programme par le caractère " par ex. `"ceci est une chaîne de caractères"`

On peut mettre certains caractères spéciaux en les préfaçant avec \, par ex.
`"cette chaîne revient\nà la ligne"`, `"et celle ci contient un \""`

Opérateurs

Les **opérateurs** sont les composants de base pour décrire des opérations à exécuter. Nous pouvons les grouper en six sous-familles:

- Opérateurs arithmétiques.
- Opérateur d'affectation.
- Opérateurs d'incrément, de décrément, et d'affectation composée.
- Opérateurs relationnels.
- Opérateurs logiques booléens.
- Opérateurs logiques bit à bit.

Les **opérateurs arithmétiques** incluent le - unaire ainsi que les opérateurs binaires classiques:

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	modulo

La division / de deux entiers est une division euclidienne, elle retourne le quotient entier. En revanche si une des opérandes est une valeur à virgule, le résultat sera à virgule.

L'opérateur de modulo % calcule le reste de la division euclidienne.

Les opérateurs +- ont une priorité inférieure à */%. Dans le cas d'opérateurs de même priorité, le calcul est effectué de gauche à droite.

```c
printf("%d\n", 3 + 4 * 5);
```

```c
printf("%d\n", 3 + 4 * 5);
```

affiche

23

```c
printf("%d\n", 3 + 4 * 5);
```

affiche

23

```c
printf("%d\n", 12 % 5);
```

```c
printf("%d\n", 3 + 4 * 5);
```

affiche

23

```c
printf("%d\n", 12 % 5);
```

affiche

2

```c
printf("%d\n", 3 * (1 + 2) );
```

```c
printf("%d\n", 3 * (1 + 2) );
```

affiche

9

```c
printf("%d\n", 3 * (1 + 2) );
```

affiche

```
9
```

```c
printf("%d\n", 15 / 20);
```

```c
printf("%d\n", 3 * (1 + 2) );
```

affiche

9

```c
printf("%d\n", 15 / 20);
```

affiche

0

```c
printf("%d\n", 3 * (1 + 2) );
```

affiche

```
9
```

```c
printf("%d\n", 15 / 20);
```

affiche

```
0
```

```c
printf("%f\n", 15 / 20.0);
```

```c
printf("%d\n", 3 * (1 + 2) );
```

affiche

```
9
```

```c
printf("%d\n", 15 / 20);
```

affiche

```
0
```

```c
printf("%f\n", 15 / 20.0);
```

affiche

```
0.750000
```

```c
printf("%d\n", 3 * 5 / 5);
```

```c
printf("%d\n", 3 * 5 / 5);
```

affiche

3

```c
printf("%d\n", 3 * 5 / 5);
```

affiche

3

```c
printf("%d\n", 3 / 5 * 5);
```

```c
printf("%d\n", 3 * 5 / 5);
```

affiche

3

```c
printf("%d\n", 3 / 5 * 5);
```

affiche

0

L'opérateur **d'affectation**, que nous avons déjà vu, copie la valeur de son opérande de droite dans son opérande de gauche qui doit donc être l'identifiant d'une variable. Il a une priorité plus faible que les opérateurs arithmétiques.

```
1  int main(void) {
2    int a;
3    3 = a;
4    return 0;
5  }
```

```
clang test.c
test.c:16:5: error: expression is not assignable
  3 = a;
```

En langage C, aussi bizarre que cela puisse paraître, une affectation a une valeur, qui est la valeur affectée. Par ex.

```
1  int a = 1, b = 2;
2  printf("%d %d\n", a, b);
3  a = (b = 3);
4  printf("%d %d\n", a, b);
```

affiche

```
1 2
3 3
```

En langage C, aussi bizarre que cela puisse paraître, une affectation a une valeur, qui est la valeur affectée. Par ex.

```
1  int a = 1, b = 2;
2  printf("%d %d\n", a, b);
3  a = (b = 3);
4  printf("%d %d\n", a, b);
```

affiche

```
1 2
3 3
```

Il est assez rare d'utiliser ce genre de syntaxe et cela doit être fait en connaissance de cause.

Les opérateurs d'**incrément**, et de **décrément**, sont des opérateurs unaires qui **modifient leur opérande.**

	Nom	Opération	Valeur
`n++`	post-incrément	ajoute 1 à `n`	`n` avant le changement
`++n`	pré-incrément	ajoute 1 à `n`	`n` après le changement
`n--`	post-décrément	soustrait 1 à `n`	`n` avant le changement
`--n`	pré-décrément	soustrait 1 à `n`	`n` après le changement

Par ex.

```
j = i++;
```

copie dans `j` la valeur courante de `i` puis incrémente `i` de 1. Équivalent à

```
j = i;
i = i + 1;
```

Par ex.

```
j = i++;
```

copie dans `j` la valeur courante de `i` puis incrémente `i` de 1. Équivalent à

```
j = i;
i = i + 1;
```

Alors que

```
j = ++i;
```

incrémente `i` de 1 puis copie dans `j` la nouvelle valeur de `i`. Équivalent à

```
i = i + 1;
j = i;
```

```c
int i, j;
i = 0;
j = i++;
printf("%d\n", j);
j = i++;
printf("%d\n", j);
j = ++i;
printf("%d\n", j);
j = ++i;
printf("%d\n", j);
```

affiche

```
0
1
3
4
```

Les opérateurs d'**affectation composée**, sont des opérateurs binaires qui modifient leur opérande de gauche.

	Opération	Valeur
`n += k`	`n = n + k`	`n` après le changement
`n -= k`	`n = n - k`	`n` après le changement
`n *= k`	`n = n * k`	`n` après le changement
`n /= k`	`n = n / k`	`n` après le changement
`n %= k`	`n = n % k`	`n` après le changement

Par ex.

```
i *= 15;
```

multiplie `i` par 15.

Par ex.

```
i *= 15;
```

multiplie `i` par 15.

Et

```
1  int a = 9;
2  int b = 4 + (a /= 3);
3  printf("a=%d b=%d\n", a, b);
```

affiche

```
a=3 b=7
```

La ligne 2 divise `a` par 3, puis copie dans `b` la valeur de `4 + a`.

Il est extrêmement périlleux d'utiliser la valeur d'une affectation:

```
1   int n;
2
3   n = 2;
4   printf("%d\n", n = n + 1);
5
6   n = 2;
7   printf("%d\n", n += 1);
8
9   n = 2;
10  printf("%d\n", n++);
```

affiche

```
3
3
2
```

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)
= += -= *= /= %=
```

Évaluation par substitution

Une manière efficace de déterminer comment une expression est “comprise” par le compilateur, et par conséquent par l'ordinateur, est l'**évaluation par substitution**, qui consiste à progressivement remplacer les opérateurs de plus forte priorité par la valeur qu'ils calculent. La substitution des variables par leurs valeurs a la plus forte priorité.

Par exemple avec `x=2` et `y=3`, on aurait

```
4 + x * (11 - y / 2)

4 + x * (11 - 3 / 2)

4 + x * (11 - 1)

4 + x * 10

4 + 2 * 10

4 + 20

24
```

Avec a=-1 et b=6, on aurait

```
3 + a * 5 / b

3 + (-1) * 5 / b

3 + (-1) * 5 / 6

3 + (-5) / 6

3 + 0

3
```

Conversions de types

Le langage C est très laxiste avec les types, et fait en particulier des **conversions implicites**: lorsque deux opérandes de types numériques différents sont combinées avec un opérateur, l'opérande avec la plus petite précision est convertie au type de l'autre avant de faire l'opération.

Par ex.

```
1  float x, y;
2  int z;
3
4  x = 3.1415926;
5  z = 2;
6  y = x + z; /* la valeur de z est convertie en float */
7  printf("%f\n", y);
```

compile sans erreur et affiche

```
5.141593
```

Le compilateur ne signale pas de problème si de l'information est perdue lors d'une affectation.

```
1   float x, y;
2   int z;
3
4   x = 4;
5   y = 1/x;
6   printf("%f\n", x * y);
7
8   x = 4;
9   z = 1/x; /* ouch */
10  printf("%f\n", x * z);
```

compile sans erreur et affiche

```
1.000000
0.000000
```

Il est possible de forcer une conversion en indiquant un type entre parenthèses avant une expression. Cet opérateur a priorité sur les opérateurs arithmétiques.

```
1  int q = 9;
2  float r;
3
4  r = 1 / q;
5  printf("%f\n", r);
6
7  r = 1 / (float) q;
8  printf("%f\n", r);
```

affiche

```
0.000000
0.111111
```

Fin