Introduction à la programmation en C

Séance 2 : variables, déclarations, affectation, commandes, composition séquentielle, blocs, conditionnelles

Luca SAIU

http://ageinghacker.net

IUT de Villetaneuse, Université Paris 13

Septembre 2018

Sommaire

Rappel

- Mon site web : `http://ageinghacker.net`
- vous trouvez la page web officielle du cours en suivant le lien "*Teaching*" ;
- je suis facile à contacter (*n'utilisez pas l'ENT*) ;
- si vous voulez le livre sur C ("*GNU C Intro and Reference*", Stallman and Rothwell, en anglais) demandez-moi.

Méta-langage : grammaires (rappel)

Cette notation est une méta-langage mathématique, employé pour décrire la syntaxe d'autres langages.

« ::= » et « | », comme utilisées ici, ne sont pas partie du langage C !)

On lit « ::= » comme « peut être » (“*can be*” en anglais), et « | » comme « ou bien » (“*or*” en anglais).

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

expression ::= littéral
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| (expression)
| - expression
| expression == expression
| expression != expression
| expression < expression
| expression <= expression
| expression > expression
| expression => expression
| variable

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

expression ::= littéral
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| (expression)
| - expression
| expression == expression
| expression != expression
| expression < expression
| expression <= expression
| expression > expression
| expression => expression
| variable

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

expression ::= littéral
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| (expression)
| - expression
| expression == expression
| expression != expression
| expression < expression
| expression <= expression
| expression > expression
| expression => expression
| variable

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

expression ::= littéral
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| (expression)
| - expression
| expression == expression
| expression != expression
| expression < expression
| expression <= expression
| expression > expression
| expression => expression
| variable

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

```
expression ::= littéral
             | expression + expression
             | expression - expression
             | expression * expression
             | expression / expression
             | ( expression )
             | - expression
             | expression == expression
             | expression != expression
             | expression < expression
             | expression <= expression
             | expression > expression
             | expression => expression
             | variable
```

Syntaxe des expressions (rappel)

J'ai montré la syntaxe des expressions au tableau la dernière fois, en suivant cet ordre (arbitraire) :

```
expression ::= littéral
             | expression + expression
             | expression - expression
             | expression * expression
             | expression / expression
             | ( expression )
             | - expression
             | expression == expression
             | expression != expression
             | expression < expression
             | expression <= expression
             | expression > expression
             | expression => expression
             | variable
```

Syntaxe des littéraux

littéral ::= littéral-entier
| littéral-flottant
| littéral-caractère
| littéral-chaîne
| littéral-booléen

littéral-booléen ::= `true`
| `false`

Pour utiliser les littéraux booléens il faut inclure la ligne
`#include <stdbool.h>`
(que j'expliquerai en détail plus tard) au début de votre fichier source (« `.c` »).

Syntaxe des littéraux

littéral ::= littéral-entier
| littéral-flottant
| littéral-caractère
| littéral-chaîne
| littéral-booléen

littéral-booléen ::= `true`
| `false`

Pour utiliser les littéraux booléens il faut inclure la ligne
`#include <stdbool.h>`
(que j'expliquerai en détail plus tard) au début de votre fichier source (« `.c` »).

Syntaxe des littéraux

littéral ::= littéral-entier
| littéral-flottant
| littéral-caractère
| littéral-chaîne
| littéral-booléen

littéral-booléen ::= `true`
| `false`

Pour utiliser les littéraux booléens il faut inclure la ligne
`#include <stdbool.h>`
(que j'expliquerai en détail plus tard) au début de votre fichier source (« .c »).

Syntaxe des littéraux

littéral ::= littéral-entier
| littéral-flottant
| littéral-caractère
| littéral-chaîne
| littéral-booléen

littéral-booléen ::= `true`
| `false`

Pour utiliser les littéraux booléens il faut inclure la ligne
`#include <stdbool.h>`
(que j'expliquerai en détail plus tard) au début de votre fichier source (« `.c` »).

Expressions logiques

Syntaxe :

expression ::= ! expression
| expression && expression
| expression || expression

Sémantique :
Ces expressions combinent des booléens pour obtenir des autres booléens. Comme d'habitude, 0 représente la valeur *fausse*, et n'importe quel autre entier une valeur *vraie*.

! est une *négation logique* (« *non* »),
&& est une conjonction logique (« *et* »),
|| est une disjonction logique (« *ou* »).

Expressions logiques

Syntaxe :

expression ::= ! expression
| expression && expression
| expression || expression

Sémantique :
Ces expressions combinent des booléens pour obtenir des autres booléens. Comme d'habitude, 0 représente la valeur *fausse*, et n'importe quel autre entier une valeur *vraie*.

! est une *négation logique* (« *non* »),
&& est une conjonction logique (« *et* »),
|| est une disjonction logique (« *ou* »).

Expressions logiques

Syntaxe :

expression ::= ! expression
| expression && expression
| expression || expression

Sémantique :
Ces expressions combinent des booléens pour obtenir des autres booléens. Comme d'habitude, 0 représente la valeur *fausse*, et n'importe quel autre entier une valeur *vraie*.

! est une *négation logique* (« *non* »),
&& est une conjonction logique (« *et* »),
|| est une disjonction logique (« *ou* »).

Expressions bit à bit

Syntaxe :

expression ::= ~ expression
| expression & expression
| expression | expression
| expression ^ expression

Sémantique :
Ces expressions travaillent sur des entiers vus comme des tableaux de bits en combinant les bits des arguments dans les mêmes positions ; leur résultat sera encore un entier du même type.

~ est une *négation bit à bit*, & est une *conjonction bit à bit*, | est une *disjonction bit à bit*, ^ est une *disjonction exclusive bit à bit*.

Par exemple : ~ 0 donne un entier avec tout bit à 1,
7 & 13 donne 5 car $(0111)_2$ & $(1101)_2 = (0101)_2$.

Expressions bit à bit

Syntaxe :

expression ::= ~ expression
| expression & expression
| expression | expression
| expression ^ expression

Sémantique :
Ces expressions travaillent sur des entiers vus comme des tableaux de bits en combinant les bits des arguments dans les mêmes positions ; leur résultat sera encore un entier du même type.

~ est une *négation bit à bit*, & est une *conjonction bit à bit*, | est une *disjonction bit à bit*, ^ est une *disjonction exclusive bit à bit*.

Par exemple : ~ 0 donne un entier avec tout bit à 1,
7 & 13 donne 5 car $(0111)_2$ & $(1101)_2 = (0101)_2$.

Expressions bit à bit

Syntaxe :

expression ::= ~ expression
| expression & expression
| expression | expression
| expression ^ expression

Sémantique :
Ces expressions travaillent sur des entiers vus comme des tableaux de bits en combinant les bits des arguments dans les mêmes positions ; leur résultat sera encore un entier du même type.

~ est une *négation bit à bit*, & est une *conjonction bit à bit*, | est une *disjonction bit à bit*, ^ est une *disjonction exclusive bit à bit*.

Par exemple : ~ 0 donne un entier avec tout bit à 1,
7 & 13 donne 5 car $(0111)_2$ & $(1101)_2 = (0101)_2$.

Expressions bit à bit

Syntaxe :

expression ::= ~ expression
| expression & expression
| expression | expression
| expression ^ expression

Sémantique :
Ces expressions travaillent sur des entiers vus comme des tableaux de bits en combinant les bits des arguments dans les mêmes positions ; leur résultat sera encore un entier du même type.

~ est une *négation bit à bit*, & est une *conjonction bit à bit*, | est une *disjonction bit à bit*, ^ est une *disjonction exclusive bit à bit*.

Par exemple : ~ 0 donne un entier avec tout bit à 1,
7 & 13 donne 5 car $(0111)_2$ & $(1101)_2 = (0101)_2$.

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée a de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée `a` de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée `a` de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée `a` de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée `a` de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables

Vous pouvez penser à une variable comme à un *nom pour une valeur en mémoire* contenant une valeur d'un certain type, associée à une certaine adresse.

(Par exemple : une variable nommée `a` de type `int` sera stockée à l'adresse 0x10000 en mémoire : l'adresse exacte sera décidée par le compilateur et le système d'exploitation).

- Une variable a *toujours* une valeur—peut-être une valeur invalide !
  - Si vous voulez une valeur spécifique dans une variable, il faut *initialiser la variable*. La valeur initiale d'une variable est toujours indéfinie, et imprévisible.
- La valeur d'une variable peut changer pendent l'exécution d'un programme, mais son type et son adresse en mémoire restent constants.

[simplification : en réalité un bon compilateur, par exemple GCC, choisira de ne pas stocker la valeur d'une variable en mémoire, quand c'est possible : accéder à la mémoire est (relativement) lent.]

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

déclaration-de-variable ::= type variable ;
| type variable = expression ;

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

```
déclaration-de-variable ::= type variable ;
                          | type variable = expression ;
```

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

```
déclaration-de-variable ::= type variable ;
                          | type variable = expression ;
```

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

```
déclaration-de-variable ::= type variable ;
                          | type variable = expression ;
```

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

déclaration-de-variable ::= type variable ;
| type variable = expression ;

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

déclaration-de-variable ::= type variable ;
| type variable = expression ;

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

déclaration-de-variable ::= type variable ;
| type variable = expression ;

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Variables : déclarations

En C toute variable doit être *déclarée* avant son utilisation.

Une déclaration spécifie :

- le type de la variable ;
- le nom de la variable (un identifiant C) ;
- *optionnellement*, la valeur initiale d'une variable.

Syntaxe [simplifiée] :

déclaration-de-variable ::= type variable ;
| type variable = expression ;

Exemples :

- `float foo;`
- `bool bar = true;`
- `float x = 10.0 / foo;`

Où déclare-t-on les variables ?

J'ai défini la syntaxe de « déclaration-de-variable », mais je n'ai jamais encore *utilisé* déclaration-de-variable.

Vous avez déjà vu qu'on peut mettre des déclarations entre `{` et `}` après `main`, mais vous ne connaissez pas encore les détails.

Une variable peut être :

- une variable *globale* (visible partout dans le même fichier source après sa déclaration), ou bien...
- ... une variable *locale* (juste visible dans son *bloc* après la déclaration), ou bien...
- ... un *paramètre formel* (on verra ces variables plus tard).

Où déclare-t-on les variables ?

J'ai défini la syntaxe de « déclaration-de-variable », mais je n'ai jamais encore *utilisé* déclaration-de-variable.

Vous avez déjà vu qu'on peut mettre des déclarations entre `{` et `}` après `main`, mais vous ne connaissez pas encore les détails.

Une variable peut être :

- une variable *globale* (visible partout dans le même fichier source après sa déclaration), ou bien...
- ... une variable *locale* (juste visible dans son *bloc* après la déclaration), ou bien...
- ... un *paramètre formel* (on verra ces variables plus tard).

Où déclare-t-on les variables ?

J'ai défini la syntaxe de « déclaration-de-variable », mais je n'ai jamais encore *utilisé* déclaration-de-variable.

Vous avez déjà vu qu'on peut mettre des déclarations entre `{` et `}` après `main`, mais vous ne connaissez pas encore les détails.

Une variable peut être :

- une variable *globale* (visible partout dans le même fichier source après sa déclaration), ou bien...
- ... une variable *locale* (juste visible dans son *bloc* après la déclaration), ou bien...
- ... un *paramètre formel* (on verra ces variables plus tard).

Où déclare-t-on les variables ?

J'ai défini la syntaxe de « déclaration-de-variable », mais je n'ai jamais encore *utilisé* déclaration-de-variable.

Vous avez déjà vu qu'on peut mettre des déclarations entre `{` et `}` après `main`, mais vous ne connaissez pas encore les détails.

Une variable peut être :

- une variable *globale* (visible partout dans le même fichier source après sa déclaration), ou bien...
- ... une variable *locale* (juste visible dans son *bloc* après la déclaration), ou bien...
- ... un *paramètre formel* (on verra ces variables plus tard).

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est... 8

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est...

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est... 8

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est... 8

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est. . . 8

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : lecture

La lecture d'une variable est facile :

Une expression-variable (il y a un cas « variable » dans la syntaxe des expressions) donne la valeur de la variable comme résultat.

Par exemple, si la variable `n` de type `int` a actuellement valeur 7 :

- le résultat de l'évaluation de l'expression `n` est 7 ;
- le résultat de l'évaluation de l'expression `n + 1` est. . . 8

Dans une expression, on peut combiner librement les variables avec n'importe quelles autres expressions.

Variables : écriture

Vous l'avez vu informellement la dernière fois. Comment change-t-on le contenu d'une variable ?

L'opérateur... (quel ?) :

On le verra dans une minute.

Variables : écriture

Vous l'avez vu informellement la dernière fois. Comment change-t-on le contenu d'une variable ?

L'opérateur... (quel ?) :

=

On le verra dans une minute.

Variables : écriture

Vous l'avez vu informellement la dernière fois. Comment change-t-on le contenu d'une variable ?

L'opérateur... (quel ?) :

=

On le verra dans une minute.

Commandes

Une *commande* (en anglais "*statement*") est une phrase du langage C évaluée pour sont effet. Les commandes, différemment par rapport aux expressions, ne donnent aucun résultat.

L'*effet* d'une commande peut être un changement de l'état du programme (valeur des variables / écritures en mémoire) ou de l'entrée/sortie.

Les commandes se combinent entre elles par composition séquentielle : on exécute une commande *après* l'autre.

Commandes

Une *commande* (en anglais "*statement*") est une phrase du langage C évaluée pour sont effet. Les commandes, différemment par rapport aux expressions, ne donnent aucun résultat.

L'*effet* d'une commande peut être un changement de l'état du programme (valeur des variables / écritures en mémoire) ou de l'entrée/sortie.

Les commandes se combinent entre elles par composition séquentielle : on exécute une commande *après* l'autre.

Commandes

Une *commande* (en anglais "*statement*") est une phrase du langage C évaluée pour sont effet. Les commandes, différemment par rapport aux expressions, ne donnent aucun résultat.

L'*effet* d'une commande peut être un changement de l'état du programme (valeur des variables / écritures en mémoire) ou de l'entrée/sortie.

Les commandes se combinent entre elles par composition séquentielle : on exécute une commande *après* l'autre.

Sortie

Vous avez déjà utilisé `printf ( ... );` comme une commande.

- Affichier un entier (expression donnant résultat de type `int`) :
  `printf ("%i", expression);`
- Affichier un caractère (expression donnant résultat de type `char`) :
  `printf ("%c", expression);`
- Affichier un floattant (expression donnant résultat de type `double`) :
  `printf ("%f", expression);`
- Affichier une chaîne (expression donnant résultat de type `char *`) :
  `printf ("%s", expression);`

Sortie

Vous avez déjà utilisé `printf ( ... );` comme une commande.

- Affichier un entier (expression donnant résultat de type `int`) :
  `printf ("%i", expression);`
- Affichier un caractère (expression donnant résultat de type `char`) :
  `printf ("%c", expression);`
- Affichier un floattant (expression donnant résultat de type `double`) :
  `printf ("%f", expression);`
- Affichier une chaîne (expression donnant résultat de type `char *`) :
  `printf ("%s", expression);`

Sortie

Vous avez déjà utilisé `printf ( ... );` comme une commande.

- Affichier un entier (expression donnant résultat de type `int`) :
  `printf ("%i", expression);`
- Affichier un caractère (expression donnant résultat de type `char`) :
  `printf ("%c", expression);`
- Affichier un floattant (expression donnant résultat de type `double`) :
  `printf ("%f", expression);`
- Affichier une chaîne (expression donnant résultat de type `char *`) :
  `printf ("%s", expression);`

Sortie

Vous avez déjà utilisé `printf ( ... );` comme une commande.

- Affichier un entier (expression donnant résultat de type `int`) :
  `printf ("%i", expression);`
- Affichier un caractère (expression donnant résultat de type `char`) :
  `printf ("%c", expression);`
- Affichier un floattant (expression donnant résultat de type `double`) :
  `printf ("%f", expression);`
- Affichier une chaîne (expression donnant résultat de type `char *`) :
  `printf ("%s", expression);`

Sortie

Vous avez déjà utilisé `printf ( ... );` comme une commande.

- Affichier un entier (expression donnant résultat de type `int`) :
  `printf ("%i", expression);`
- Affichier un caractère (expression donnant résultat de type `char`) :
  `printf ("%c", expression);`
- Affichier un floattant (expression donnant résultat de type `double`) :
  `printf ("%f", expression);`
- Affichier une chaîne (expression donnant résultat de type `char *`) :
  `printf ("%s", expression);`

Variables : affectation (« écriture »)

On peut changer la valeur d'une variable grâce à l'opérateur d'affectation ("assignment" en anglais) =.

Syntaxe [simplifiée] :

expression ::= variable = expression

Sémantique :

L'expression à droite de = est évaluée en obtenant une valeur v, puis la valeur de la variable à gauche de = est mise à v, en effaçant la valeur précédente. Le résultat de l'expression-affectation est v.

Exemple : si `foo` a valeur 6, l'expression

```
foo = foo + 2
```

met la valeur de `foo` à 8, et donne 8 comme résultat.

Variables : affectation (« écriture »)

On peut changer la valeur d'une variable grâce à l'opérateur d'affectation ("assignment" en anglais) =.

Syntaxe [simplifiée] :

expression ::= variable = expression

Sémantique :

L'expression à droite de = est évaluée en obtenant une valeur v, puis la valeur de la variable à gauche de = est mise à v, en effaçant la valeur précédente. Le résultat de l'expression-affectation est v.

Exemple : si `foo` a valeur 6, l'expression

```
foo = foo + 2
```

met la valeur de `foo` à 8, et donne 8 comme résultat.

Variables : affectation (« écriture »)

On peut changer la valeur d'une variable grâce à l'opérateur d'affectation ("assignment" en anglais) =.

Syntaxe [simplifiée] :

expression ::= variable = expression

Sémantique :

L'expression à droite de = est évaluée en obtenant une valeur v, puis la valeur de la variable à gauche de = est mise à v, en effaçant la valeur précédente. Le résultat de l'expression-affectation est v.

Exemple : si `foo` a valeur 6, l'expression

```
foo = foo + 2
```

met la valeur de `foo` à 8, et donne 8 comme résultat.

Variables : affectation (« écriture »)

On peut changer la valeur d'une variable grâce à l'opérateur d'affectation ("assignment" en anglais) =.

Syntaxe [simplifiée] :

expression ::= variable = expression

Sémantique :
L'expression à droite de = est évaluée en obtenant une valeur v, puis la valeur de la variable à gauche de = est mise à v, en effaçant la valeur précédente. Le résultat de l'expression-affectation est v.

Exemple : si `foo` a valeur 6, l'expression
`foo = foo + 2`
met la valeur de `foo` à 8, et donne 8 comme résultat.

Affectation : remarques

expression ::= variable = expression

En C (différemment que dans des autres langages) l'affectation est une expression...

... et l'expression à droite de = peut être une autre affectation !

Les *affectations chaînées* s'évaluent de droite à gauche :

```
a = b = c = 5
```

mets à 5 *les trois variables* a, b, et c.

On pourrait dire que = est un opérateur « associant à droite », car `a = b = c = 5` est égal à `a = (b = (c = 5))`. Certainement non à `((a = b) = c) = 5` (qui ne respecte même pas la syntaxe).

Affectation : remarques

expression ::= variable = expression

En C (différemment que dans des autres langages) l'affectation est une expression...

... et l'expression à droite de = peut être une autre affectation !
Les *affectations chaînées* s'évaluent de droite à gauche :
`a = b = c = 5`
mets à 5 *les trois variables* `a`, `b`, et `c`.

On pourrait dire que = est un opérateur « associant à droite », car `a = b = c = 5` est égal à `a = (b = (c = 5))`. Certainement non à `((a = b) = c) = 5` (qui ne respecte même pas la syntaxe).

Affectation : remarques

expression ::= variable = expression

En C (différemment que dans des autres langages) l'affectation est une expression...

... et l'expression à droite de = peut être une autre affectation !
Les *affectations chaînées* s'évaluent de droite à gauche :
`a = b = c = 5`
mets à 5 *les trois variables* `a`, `b`, et `c`.

On pourrait dire que = est un opérateur « associant à droite », car `a = b = c = 5` est égal à `a = (b = (c = 5))`. Certainement non à `((a = b) = c) = 5` (qui ne respecte même pas la syntaxe).

Comparaison et affectation : astuce pratique

Faites attention à la différence entre = et == :

`a == 3` est très différent de `a = 3` !

Je vous conseille d'écrire toujours « `3 == a` » à la place de « `a == 3` ». (Pourquoi, à votre avis ?)

Comparaison et affectation : astuce pratique

Faites attention à la différence entre = et == :
`a == 3` est très différent de `a = 3` !

Je vous conseille d'écrire toujours « `3 == a` » à la place de « `a == 3` ». (Pourquoi, à votre avis ?)

Blocs

Un *bloc* est une phrase du langage qui contient une séquence de commandes et/ou de déclarations de variables. Un bloc est lui-même une commande.

Syntaxe :

```
commande    ::=    expression ;
                 |    bloc

bloc    ::=    { contenu-du-bloc }

contenu-du-bloc    ::=
                       |    commande contenu-du-bloc
                       |    déclaration-de-variable contenu-du-bloc
```

Sémantique (bloc) :

On exécute les commandes dans un bloc en ordre, l'un après l'autre. Chaque variable locale est juste visible à l'intérieur de son bloc, y compris les sous-blocs, après sa déclaration.

Blocs

Un *bloc* est une phrase du langage qui contient une séquence de commandes et/ou de déclarations de variables. Un bloc est lui-même une commande.

Syntaxe :

commande ::= expression ;
| bloc

bloc ::= { contenu-du-bloc }

contenu-du-bloc ::=
| commande contenu-du-bloc
| déclaration-de-variable contenu-du-bloc

Sémantique (bloc) :

On exécute les commandes dans un bloc en ordre, l'un après l'autre. Chaque variable locale est juste visible à l'intérieur de son bloc, y compris les sous-blocs, après sa déclaration.

Blocs

Un *bloc* est une phrase du langage qui contient une séquence de commandes et/ou de déclarations de variables. Un bloc est lui-même une commande.

Syntaxe :

commande ::= expression ;
| bloc

bloc ::= { contenu-du-bloc }

contenu-du-bloc ::=
| commande contenu-du-bloc
| déclaration-de-variable contenu-du-bloc

Sémantique (bloc) :
On exécute les commandes dans un bloc en ordre, l'un après l'autre. Chaque variable locale est juste visible à l'intérieur de son bloc, y compris les sous-blocs, après sa déclaration.

Conditionnelles

Syntaxe :

commande ::= `if (` expression `)` commande `else` commande

On appelle "*condition*" l'expression, "*branche then*" la première (sous-)commande "*branche else*" la deuxième (sous-)commande.

Sémantique :

On évalue la condition : si le résultat (booléen) est vrai, alors on exécute juste la branche then ; si le résultat de la condition est faux alors on exécute juste la branche else.

[Correction : dans les transparents originaux utilisés en classe j'avais mis un `;` à la fin de la règle de grammaire pour la conditionnelle. Non, c'était incorrecte : le `;` à la fin de la commande conditionnelle, quand nécessaire, vient de la dernière commande, si elle n'est pas un bloc ; si elle est un bloc il n'y a pas de `;` obligatoire.]

Conditionnelles

Syntaxe :

commande ::= `if (` expression `)` commande `else` commande

On appelle "*condition*" l'expression, "*branche then*" la première (sous-)commande "*branche else*" la deuxième (sous-)commande.

Sémantique :
On évalue la condition : si le résultat (booléen) est vrai, alors on exécute juste la branche then ; si le résultat de la condition est faux alors on exécute juste la branche else.

[Correction : dans les transparents originaux utilisés en classe j'avais mis un `;` à la fin de la règle de grammaire pour la conditionnelle. Non, c'était incorrecte : le `;` à la fin de la commande conditionnelle, quand nécessaire, vient de la dernière commande, si elle n'est pas un bloc ; si elle est un bloc il n'y a pas de `;` obligatoire.]

Conditionnelles sans `else`

Syntaxe : commande ::= `if ( expression )` commande

Sémantique :

`if ( expression )` commande
se comporte exactement comme
`if ( expression )` commande `else {}`

[Correction : dans les transparents originaux utilisés en classe j'avais mis un `;` à la fin de la règle de grammaire pour la conditionnelle. Non, c'était incorrecte : le `;` à la fin de la commande conditionnelle, quand nécessaire, vient de la dernière commande, si elle n'est pas un bloc ; si elle est un bloc il n'y a pas de `;` obligatoire.]

Conditionnelles sans `else`

Syntaxe : commande ::= `if (` expression `)` commande

Sémantique :
`if (` expression `)` commande
se comporte exactement comme
`if (` expression `)` commande `else {}`

[Correction : dans les transparents originaux utilisés en classe j'avais mis un `;` à la fin de la règle de grammaire pour la conditionnelle. Non, c'était incorrecte : le `;` à la fin de la commande conditionnelle, quand nécessaire, vient de la dernière commande, si elle n'est pas un bloc ; si elle est un bloc il n'y a pas de `;` obligatoire.]

Don't panic !

Vos programmes ne marcheront pas à la première tentative.

C'est normal, même pour les experts. Ne vous inquiétez pas !

Welcome to the world of *debugging*.

Bibliography I


Saiu, L. (2018). La page web de mes cours.
`http://ageinghacker.net/teaching`
La page web officielle du cours contient des pointeurs à des ressources web, et une copie des mes transparents.


Stallman, R. and Rothwell, T. (2019). *GNU C Intro and Reference*. Free Software Foundation.
(Manuscrit non encore officiellement publié. Demandez-moi un exemplaire si vous êtes intéressés.).