Introduction à la programmation en C

Séance 4 : abstraction procédurale

Luca SAIU

http://ageinghacker.net

IUT de Villetaneuse, Université Paris 13

Octobre 2018

Sommaire I

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.

Il faut terminer les *déclarations* aussi par ;

Comme montré dans la syntaxe, les déclarations sont terminées par `;`, comme les commandes qui ne sont pas de bloques.

Si vous oubliez le `;` un message d'erreur de GCC typique est `expected , or ; before` *`la ligne suivante`*

Je conseille de *compiler tout le temps*

Passer d'un programme syntaxiquement correct à un autre.

Je suggère de :

- compiler juste après chaque modification, même très petite ;
- corriger toute erreur (et warning), immédiatement ;
- . . . *puis*, essayer votre changement.

Bientôt, vous n'allez plus être distrait par des erreurs de syntaxe, et vous allez passer le temps en pensant au *comportement* de votre programme.

Je conseille de *compiler tout le temps*

Passer d'un programme syntaxiquement correct à un autre.

Je suggère de :

- compiler juste après chaque modification, même très petite ;
- corriger toute erreur (et warning), immédiatement ;
- . . . *puis*, essayer votre changement.

Bientôt, vous n'allez plus être distrait par des erreurs de syntaxe, et vous allez passer le temps en pensant au *comportement* de votre programme.

Je conseille de *compiler tout le temps*

Passer d'un programme syntaxiquement correct à un autre.

Je suggère de :

- compiler juste après chaque modification, même très petite ;
- corriger toute erreur (et warning), immédiatement ;
- . . . *puis*, essayer votre changement.

Bientôt, vous n'allez plus être distrait par des erreurs de syntaxe, et vous allez passer le temps en pensant au *comportement* de votre programme.

Je conseille de *compiler tout le temps*

Passer d'un programme syntaxiquement correct à un autre.

Je suggère de :

- compiler juste après chaque modification, même très petite ;
- corriger toute erreur (et warning), immédiatement ;
- . . . *puis*, essayer votre changement.

Bientôt, vous n'allez plus être distrait par des erreurs de syntaxe, et vous allez passer le temps en pensant au *comportement* de votre programme.

Je conseille de *compiler tout le temps*

Passer d'un programme syntaxiquement correct à un autre.

Je suggère de :

- compiler juste après chaque modification, même très petite ;
- corriger toute erreur (et warning), immédiatement ;
- . . . *puis*, essayer votre changement.

Bientôt, vous n'allez plus être distrait par des erreurs de syntaxe, et vous allez passer le temps en pensant au *comportement* de votre programme.

Utilisez printf pour comprendre votre programme : 1

```
int a = ...;
int b = ...;

while (a != b)
  if (a > b)
    a = a - b;
  else
    b = b - a;
...
```

Si vous avez écrit ce code et vous n'êtes pas sûrs de sa correction, ajoutez (temporairement) des printf.

Utilisez printf pour comprendre votre programme : 2

```
int a = ...;
int b = ...;

while (a != b)
  {
    printf ("a est %i\n", a);
    printf ("b est %i\n", b);
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
```

Le début d'une boucle est une bonne place où afficher les valeurs des variables.

Boucles imbriquées (1) :

On utilise normalement une boucle `for` pour exécuter une commande N fois :

```
int i;
for (i = 0; i < N; i = i + 1)
  COMMANDE
```

Boucles imbriquées (2) :

Dans ce cas, combien de fois on exécute la commande qui est le corps de la boucle interne ?

```
int i;
int j;
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    COMMANDE
```

Réponse : N^2 (les nombres d'itérations dans des boucles imbriquées se multiplient).

Boucles imbriquées (2) :

Dans ce cas, combien de fois on exécute la commande qui est le corps de la boucle interne ?

```
int i;
int j;
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
    COMMANDE
```

Réponse : N^2. Les nombres d'itérations dans des boucles imbriqués *se multiplient*.

Exercices à faire à la maison avant le partiel — *vraiment*

Écrivez des programmes C utilisant une seule boucle `for`, affichant...

- les nombres naturels entre 0 et 1000, compris ;
- les nombres naturels entre 1000 et 0, compris (en comptant *vers le bas*) ;
- les nombres naturels *paires* entre 0 et 1000 ;
- les nombres naturels *impaires* entre 1 et 1001.

Écrivez aussi des programmes C affichant :

- la somme des nombres naturels entre 1 et 1000 ;
- la sortie (infinie) d'un jeu de *Fizz buzz* :
  - c'est bien expliqué à la page `https://en.wikipedia.org/wiki/Fizz_buzz`

Toujours *en utilisant des fonctions* où c'est raisonnable (presque partout).

Exercices à faire à la maison avant le partiel — *vraiment*

Écrivez des programmes C utilisant une seule boucle `for`, affichant...

- les nombres naturels entre 0 et 1000, compris ;
- les nombres naturels entre 1000 et 0, compris (en comptant *vers le bas*) ;
- les nombres naturels *paires* entre 0 et 1000 ;
- les nombres naturels *impaires* entre 1 et 1001.

Écrivez aussi des programmes C affichant :

- la somme des nombres naturels entre 1 et 1000 ;
- la sortie (infinie) d'un jeu de *Fizz buzz* :
  - c'est bien expliqué à la page `https://en.wikipedia.org/wiki/Fizz_buzz`

Toujours *en utilisant des fonctions* où c'est raisonnable (presque partout).

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :

La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :

La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :

La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :

La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :
La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-incrément et post-incrément : ++

Syntaxe [simplifiée] :

expression ::= ++ variable
| variable ++

Sémantique :
La variable est affectée en lui donnant sa valeur actuelle plus 1.

- Le résultat de « ++ variable » est la *nouvelle* valeur de la variable.
- Le résultat de « variable ++ » est l'*ancienne* valeur de la variable.

La différence entre pre- et post-incrément n'est pas importante quand vous ignorez le résultat de l'expression : expression utilisée comme une commande, ou dans la partie *incrémentation* d'une boucle `for`.

Je suggère aux débutants de *n'utiliser ++ qu'en ignorant le résultat* de l'expression, juste pour son *effet*.

Pré-décrément et post-décrément : --

Syntaxe [simplifiée] :

expression	::=	-- variable
	|	variable --

Sémantique :

Exactement pareil à « ++ variable » et « variable ++ », sauf que -- affecte la variable en lui donnant sa valeur actuelle *moins 1* au lieu de plus 1.

Les considérations pratiques sont les mêmes.

Pré-décrément et post-décrément : --

Syntaxe [simplifiée] :

expression	::=	-- variable
	|	variable --

Sémantique :

Exactement pareil à « ++ variable » et « variable ++ », sauf que -- affecte la variable en lui donnant sa valeur actuelle *moins 1* au lieu de plus 1.

Les considérations pratiques sont les mêmes.

Pré-décrément et post-décrément : --

Syntaxe [simplifiée] :

expression	::=	-- variable
	|	variable --

Sémantique :
Exactement pareil à « ++ variable » et « variable ++ », sauf que -- affecte la variable en lui donnant sa valeur actuelle *moins 1* au lieu de plus 1.

Les considérations pratiques sont les mêmes.

Un autre opérateur avec effet : +=

Syntaxe [simplifiée] :

expression ::= variable += expression

Sémantique :

Équivalent à « variable = variable + expression », dans le résultat aussi. Le résultat de l'expression avec += est la nouvelle valeur de la variable.

Les débutants devraient utiliser cette syntaxe juste quand le résultat n'est pas utilisé (c'est à dire, quand l'expression est une commande).

Un autre opérateur avec effet : +=

Syntaxe [simplifiée] :

expression ::= variable += expression

Sémantique :

Équivalent à « variable = variable + expression », dans le résultat aussi. Le résultat de l'expression avec += est la nouvelle valeur de la variable.

Les débutants devraient utiliser cette syntaxe juste quand le résultat n'est pas utilisé (c'est à dire, quand l'expression est une commande).

Un autre opérateur avec effet : +=

Syntaxe [simplifiée] :

expression ::= variable += expression

Sémantique :

Équivalent à « variable = variable + expression », dans le résultat aussi. Le résultat de l'expression avec += est la nouvelle valeur de la variable.

Les débutants devraient utiliser cette syntaxe juste quand le résultat n'est pas utilisé (c'est à dire, quand l'expression est une commande).

Encore d'autres opérateurs avec effets . . .

De façon similaire à `+=` on a aussi :

- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `^=`
- . . .

Mêmes considérations.

Opérateurs avec effets : résumé

Au lieu de « variable = variable + 1 » vous pouvez écrire « variable ++ » ou « ++ variable ».

Au lieu de « variable = variable + expression » vous pouvez écrire « variable += expression ».

Au lieu de « variable = variable - expression » vous pouvez écrire « variable -= expression ».

...

Je suggère de ne pas utiliser le résultat de ces expressions, mais juste leur effet.

Opérateurs avec effets : résumé

Au lieu de « variable = variable + 1 » vous pouvez écrire « variable ++ » ou « ++ variable ».

Au lieu de « variable = variable + expression » vous pouvez écrire « variable += expression ».

Au lieu de « variable = variable - expression » vous pouvez écrire « variable -= expression ».

...

Je suggère de ne pas utiliser le résultat de ces expressions, mais juste leur effet.

Opérateurs avec effets : résumé

Au lieu de « variable = variable + 1 » vous pouvez écrire « variable ++ » ou « ++ variable ».

Au lieu de « variable = variable + expression » vous pouvez écrire « variable += expression ».

Au lieu de « variable = variable - expression » vous pouvez écrire « variable -= expression ».

...

Je suggère de ne pas utiliser le résultat de ces expressions, mais juste leur effet.

Opérateurs avec effets : résumé

Au lieu de « variable = variable + 1 » vous pouvez écrire « variable ++ » ou « ++ variable ».

Au lieu de « variable = variable + expression » vous pouvez écrire « variable += expression ».

Au lieu de « variable = variable - expression » vous pouvez écrire « variable -= expression ».

...

Je suggère de ne pas utiliser le résultat de ces expressions, mais juste leur effet.

Opérateurs avec effets : résumé

Au lieu de « variable = variable + 1 » vous pouvez écrire « variable ++ » ou « ++ variable ».

Au lieu de « variable = variable + expression » vous pouvez écrire « variable += expression ».

Au lieu de « variable = variable - expression » vous pouvez écrire « variable -= expression ».

...

Je suggère de ne pas utiliser le résultat de ces expressions, mais juste leur effet.

Sortie anticipée d'une boucle : `break` et `continue`

Syntaxe :

commande ::= `break ;`
| `continue ;`

Juste valides à l'intéreieur d'une boucle (`while`, ou `do`...`while`, ou `for` [`break` est aussi valide dans les conditionnelles `switch`, que je n'ai pas montré]).

Sémantique :

L'exécution du corps de la boucle s'arrête. Avec `break` *la boucle (la plus interne) termine*, et on passe à la commande suivante (après la boucle). Avec `continue` on passe *à la prochaine itération* de la même boucle, sans exécuter le reste du corps.

À la place de `break` et `continue` on pourrait utiliser `goto`.
Certains puristes n'aiment pas la sortie anticipée, mais elle est utile dans les vrais programmes.
`continue` n'est pas pour les débutants.

Sortie anticipée d'une boucle : `break` et `continue`

Syntaxe :

commande ::= `break ;`
| `continue ;`

Juste valides à l'intéreieur d'une boucle (`while`, ou `do...while`, ou `for` [`break` est aussi valide dans les conditionnelles `switch`, que je n'ai pas montré]).

Sémantique :

L'exécution du corps de la boucle s'arrête. Avec `break` *la boucle (la plus interne) termine*, et on passe à la commande suivante (après la boucle). Avec `continue` on passe *à la prochaine itération* de la même boucle, sans exécuter le reste du corps.

À la place de `break` et `continue` on pourrait utiliser `goto`.
Certains puristes n'aiment pas la sortie anticipée, mais elle est utile dans les vrais programmes.
`continue` n'est pas pour les débutants.

Sortie anticipée d'une boucle : `break` et `continue`

Syntaxe :

commande ::= `break ;`
| `continue ;`

Juste valides à l'intéreieur d'une boucle (`while`, ou `do...while`, ou `for` [`break` est aussi valide dans les conditionnelles `switch`, que je n'ai pas montré]).

Sémantique :
L'exécution du corps de la boucle s'arrête. Avec `break` *la boucle (la plus interne) termine*, et on passe à la commande suivante (après la boucle). Avec `continue` on passe *à la prochaine itération* de la même boucle, sans exécuter le reste du corps.

À la place de `break` et `continue` on pourrait utiliser `goto`.
Certains puristes n'aiment pas la sortie anticipée, mais elle est utile dans les vrais programmes.
`continue` n'est pas pour les débutants.

Sortie anticipée d'une boucle : `break` et `continue`

Syntaxe :

commande ::= `break ;`
| `continue ;`

Juste valides à l'intéreieur d'une boucle (`while`, ou `do...while`, ou `for` [`break` est aussi valide dans les conditionnelles `switch`, que je n'ai pas montré]).

Sémantique :
L'exécution du corps de la boucle s'arrête. Avec `break` *la boucle (la plus interne) termine*, et on passe à la commande suivante (après la boucle). Avec `continue` on passe *à la prochaine itération* de la même boucle, sans exécuter le reste du corps.

À la place de `break` et `continue` on pourrait utiliser `goto`.
Certains puristes n'aiment pas la sortie anticipée, mais elle est utile dans les vrais programmes.
`continue` n'est pas pour les débutants.

Fonctions en C

J'utilise *le nom « fonction »* car c'est dans la norme et tout le monde l'utilisent, mais il faudrait parler de *procédures* plutôt que de « fonctions » au sens mathématique.

En C une *fonction* C est un morceau de code contenant un bloc, potentiellement avec des paramètres et un résultat.

On utilise une fonction dans un *appel de fonction* (qui est une expression) : on fait l'appel en donnant le nom de la fonction et les paramètres.
La fonction fera son travail, puis renverra son résultat, et le code qui a appelé la fonction continuera.

Les fonctions sont *extrêmement importantes* pour écrire du bon code. Utiles pour *factoriser*.

Fonctions en C

J'utilise *le nom « fonction »* car c'est dans la norme et tout le monde l'utilisent, mais il faudrait parler de *procédures* plutôt que de « fonctions » au sens mathématique.

En C une *fonction* C est un morceau de code contenant un bloc, potentiellement avec des paramètres et un résultat.

On utilise une fonction dans un *appel de fonction* (qui est une expression) : on fait l'appel en donnant le nom de la fonction et les paramètres.
La fonction fera son travail, puis renverra son résultat, et le code qui a appelé la fonction continuera.

Les fonctions sont *extrêmement importantes* pour écrire du bon code. Utiles pour *factoriser*.

Fonctions en C

J'utilise *le nom « fonction »* car c'est dans la norme et tout le monde l'utilisent, mais il faudrait parler de *procédures* plutôt que de « fonctions » au sens mathématique.

En C une *fonction* C est un morceau de code contenant un bloc, potentiellement avec des paramètres et un résultat.

On utilise une fonction dans un *appel de fonction* (qui est une expression) : on fait l'appel en donnant le nom de la fonction et les paramètres.
La fonction fera son travail, puis renverra son résultat, et le code qui a appelé la fonction continuera.

Les fonctions sont *extrêmement importantes* pour écrire du bon code. Utiles pour *factoriser*.

Fonctions en C

J'utilise *le nom « fonction »* car c'est dans la norme et tout le monde l'utilisent, mais il faudrait parler de *procédures* plutôt que de « fonctions » au sens mathématique.

En C une *fonction* C est un morceau de code contenant un bloc, potentiellement avec des paramètres et un résultat.

On utilise une fonction dans un *appel de fonction* (qui est une expression) : on fait l'appel en donnant le nom de la fonction et les paramètres.
La fonction fera son travail, puis renverra son résultat, et le code qui a appelé la fonction continuera.

Les fonctions sont *extrêmement importantes* pour écrire du bon code. Utiles pour *factoriser*.

Le *prototype* (ou déclaration) d'une fonction

Syntaxe [simplifiée] :

```
prototype   ::=   type nom-fonction ( formels ) ;
formels     ::=   void
                | formels-non-vides
formels-non-vides   ::=   type nom
                        | type nom , formels-non-vides
```

Le nom de la fonction, « nom-fonction », est juste un identifiant.

Les prototypes sont des *déclarations globales* (hors de tout bloc).

Sémantique :

La fonction nommée est reconnue dans les appels à partir de la déclaration, même si elle n'a pas encore été définie. À partir de chaque déclaration on pourra appeler la fonction nommée.

Le type à gauche est le type du *résultat*, ou `void` s'il n'y a pas de résultat. Les autres types sont pour les *paramètres formels*, dans l'ordre.

Prototype *non obligatoire*, si la *définition précède le premier appel*.

Le *prototype* (ou déclaration) d'une fonction

Syntaxe [simplifiée] :

```
prototype   ::=   type nom-fonction ( formels ) ;
formels     ::=   void
                | formels-non-vides
formels-non-vides   ::=   type nom
                        | type nom, formels-non-vides
```

Le nom de la fonction, « nom-fonction », est juste un identifiant.

Les prototypes sont des *déclarations globales* (hors de tout bloc).

Sémantique :

La fonction nommée est reconnue dans les appels à partir de la déclaration, même si elle n'a pas encore été définie. À partir de chaque déclaration on pourra appeler la fonction nommée.

Le type à gauche est le type du *résultat*, ou `void` s'il n'y a pas de résultat. Les autres types sont pour les *paramètres formels*, dans l'ordre.

Prototype *non obligatoire*, si la *définition précède le premier appel*.

Le *prototype* (ou déclaration) d'une fonction

Syntaxe [simplifiée] :

```
prototype   ::=   type nom-fonction ( formels ) ;
formels     ::=   void
                | formels-non-vides
formels-non-vides   ::=   type nom
                        | type nom, formels-non-vides
```

Le nom de la fonction, « nom-fonction », est juste un identifiant.

Les prototypes sont des *déclarations globales* (hors de tout bloc).

Sémantique :

La fonction nommée est reconnue dans les appels à partir de la déclaration, même si elle n'a pas encore été définie. À partir de chaque déclaration on pourra appeler la fonction nommée.

Le type à gauche est le type du *résultat*, ou `void` s'il n'y a pas de résultat. Les autres types sont pour les *paramètres formels*, dans l'ordre.

Prototype *non obligatoire*, si la *définition précède le premier appel*.

Définition d'une fonction

Syntaxe [simplifiée] :

définition-fonction ::= type nom-fonction (paramètres-formels) bloc

Le bloc est le *corps* de la fonction.

Les définitions de fonctions sont (en C standard) *globales* (hors de tout bloc).

Sémantique :

La fonction définie est associée au nom. La définition peut remplacer un prototype et doit être cohérente avec un prototype de la même fonction

La différence syntaxique entre un prototype et une définition de fonction est juste donnée par ; (prototype) ou un bloc pour le corps (définition).

Définition d'une fonction

Syntaxe [simplifiée] :

définition-fonction ::= type nom-fonction (paramètres-formels) bloc

Le bloc est le *corps* de la fonction.

Les définitions de fonctions sont (en C standard) *globales* (hors de tout bloc).

Sémantique :

La fonction définie est associée au nom. La définition peut remplacer un prototype et doit être cohérente avec un prototype de la même fonction

La différence syntaxique entre un prototype et une définition de fonction est juste donnée par ; (prototype) ou un bloc pour le corps (définition).

Définition d'une fonction

Syntaxe [simplifiée] :

définition-fonction ::= type nom-fonction (paramètres-formels) bloc

Le bloc est le *corps* de la fonction.

Les définitions de fonctions sont (en C standard) *globales* (hors de tout bloc).

Sémantique :
La fonction définie est associée au nom. La définition peut remplacer un prototype et doit être cohérente avec un prototype de la même fonction

La différence syntaxique entre un prototype et une définition de fonction est juste donnée par ; (prototype) ou un bloc pour le corps (définition).

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.

La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.

La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :

Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.

En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.

La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.

La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :

Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.

En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.

La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.

La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :

Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.

En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.

La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.

La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :

Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.

En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.
La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.
La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :
Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.
En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Renvoi du résultat

Syntaxe [simplifiée] :

commande ::= `return ;`
| `return` expression `;`

L'expression est le *résultat* de la fonction.
La première forme (sans résultat) est juste valide où le type du « résultat » est `void`.
La deuxième forme est juste valide où le type du résultat *n'est pas* `void`.

Sémantique :
Dans le cas de la deuxième forme (avec résultat), on évalue l'expression, qui devient le *résultat de l'appel* de la fonction.
En tout cas, *l'exécution de la fonction termine*, et on repasse le contrôle à qui a appelé la fonction actuelle.

`return` (sans résultat) n'est pas indispensable dans une fonction « renvoyant » `void`, si le retour est à exécuter à la fin du corps.

Appel d'une fonction (toujours *call-by-value* en C)

Syntaxe [simplifiée] :

expression ::= nom-fonction (actuels)

actuels ::=
| actuels-non-vides

actuels-non-vides ::= expression
| expression , actuels-non-vides

Sémantique :

Les *paramètres actuels* sont évalués, *dans un ordre indéfini*.

L'exécution du code appelant la fonction est suspendue, et on passe à l'exécution du corps de la fonction, où *les paramètres formels ont été assignés, en tant que variables, aux résultats de l'évaluation des paramètres actuels*.

Dans le corps *les seuls variables visibles sont les paramètres formels et les variables globales*.

Quand le corps termine le contrôle revient au code qui à fait l'appel. Le résultat de l'appel est le résultat renvoyé par `return`.

Appel d'une fonction (toujours *call-by-value* en C)

Syntaxe [simplifiée] :

expression ::= nom-fonction (actuels)

actuels ::=
| actuels-non-vides

actuels-non-vides ::= expression
| expression , actuels-non-vides

Sémantique :

Les *paramètres actuels* sont évalués, *dans un ordre indéfini*.

L'exécution du code appelant la fonction est suspendue, et on passe à l'exécution du corps de la fonction, où *les paramètres formels ont été assignés, en tant que variables, aux résultats de l'évaluation des paramètres actuels*.

Dans le corps *les seuls variables visibles sont les paramètres formels et les variables globales.*

Quand le corps termine le contrôle revient au code qui à fait l'appel. Le résultat de l'appel est le résultat renvoyé par `return`.

Appel d'une fonction (toujours *call-by-value* en C)

Syntaxe [simplifiée] :

expression ::= nom-fonction (actuels)

actuels ::=
| actuels-non-vides

actuels-non-vides ::= expression
| expression , actuels-non-vides

Sémantique :

Les *paramètres actuels* sont évalués, *dans un ordre indéfini*.

L'exécution du code appelant la fonction est suspendue, et on passe à l'exécution du corps de la fonction, où *les paramètres formels ont été assignés, en tant que variables, aux résultats de l'évaluation des paramètres actuels*.

Dans le corps *les seuls variables visibles sont les paramètres formels et les variables globales.*

Quand le corps termine le contrôle revient au code qui à fait l'appel. Le résultat de l'appel est le résultat renvoyé par `return`.

Appel d'une fonction (toujours *call-by-value* en C)

Syntaxe [simplifiée] :

expression ::= nom-fonction (actuels)

actuels ::=
| actuels-non-vides

actuels-non-vides ::= expression
| expression , actuels-non-vides

Sémantique :

Les *paramètres actuels* sont évalués, *dans un ordre indéfini*.

L'exécution du code appelant la fonction est suspendue, et on passe à l'exécution du corps de la fonction, où *les paramètres formels ont été assignés, en tant que variables, aux résultats de l'évaluation des paramètres actuels*.

Dans le corps *les seuls variables visibles sont les paramètres formels et les variables globales*.

Quand le corps termine le contrôle revient au code qui à fait l'appel. Le résultat de l'appel est le résultat renvoyé par `return`.

Appel d'une fonction (toujours *call-by-value* en C)

Syntaxe [simplifiée] :

expression ::= nom-fonction (actuels)

actuels ::=
| actuels-non-vides

actuels-non-vides ::= expression
| expression , actuels-non-vides

Sémantique :

Les *paramètres actuels* sont évalués, *dans un ordre indéfini*.

L'exécution du code appelant la fonction est suspendue, et on passe à l'exécution du corps de la fonction, où *les paramètres formels ont été assignés, en tant que variables, aux résultats de l'évaluation des paramètres actuels*.

Dans le corps *les seuls variables visibles sont les paramètres formels et les variables globales*.

Quand le corps termine le contrôle revient au code qui à fait l'appel. Le résultat de l'appel est le résultat renvoyé par `return`.

`main` : maintenant vous la comprenez

- `main` est une fonction ;
- Celui qui « appelle » `main` est *le système d'exploitation* ;
- Le résultat (de type `int`) de `main` est un code numérique : zéro en cas de succès, non-zéro en cas d'échec.

Top-down et Bottom-up

Comment écrire, comment tester.

« Étant donnée »

« Étant donnée » : paramètre

Mes félicitations

[si vous avez maîtrisé tout élément essentiel. . .]

Vous êtes maintenant des programmeurs.

Don't panic ! (again)

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.).