Les fonctions en C

Jean-Gabriel Luque

September 20, 2025

1 Rôle et utilisation des fonctions

1.1 À quoi ça sert?

Tout d'abord remarquons que vous avez déjà manipulé des fonctions en C. Par exemple, les chapitres précédents vous ont permis d'utiliser les fonctions `scanf` et `printf`.
D'un point de vue "machine", tout ce que vous pouvez faire avec des fonctions peut être fait sans fonction. À ce moment du cours, nous n'avons pas introduit les instructions de test ni les instructions de boucle. Donc, pour l'instant, exécuter un programme revient à lire et exécuter ses instructions ligne par ligne à partir de la ligne "`int␣main(void)`". Quelques soient les actions de l'utilisateur, les mêmes lignes seront exécutées et dans le même ordre. Nous verrons que l'introduction des fonctions rend la lecture du programme moins linéaire néanmoins l'ordre d'exécution des instructions restera indépendant des actions de l'utilisateur.
L'informatique ne consiste pas simplement à communiquer avec un ordinateur. Une part non négligeable du travail de l'informaticien doit être consacrée à la clarté de son programme. Le programmeur doit toujours penser aux utilisateurs. Le concept d'utilisateur ne se limite pas aux clients "finaux" de l'application mais à toute une chaîne d'intervenants comprenant aussi d'autres programmeurs. Un programme doit pouvoir être lu, corrigé et amélioré par d'autres. Il doit donc être écrit de la façon la plus claire et la plus consise possible.
Un programme illisible n'est pas un programme!
Le concept de fonction est absolument essentiel en informatique car il permet de réduire considérablement le code et de le rendre beaucoup plus lisible. En effet, une fonction est un "bout de code" paramétrable pouvant être appelé à tout moment dans un programme. L'appel d'une fonction peut permettre

- d'agir sur les composants internes (calcul en mémoire, modification de variables etc.),
- d'agir sur les périphériques (lecture clavier, écriture à l'écran, gestion des fichiers etc.),
- de renvoyer une valeur entrant dans le processus d'évaluation d'une expression.

Reprenons l'exemple de la fonction `scanf`. Celle-ci est définie dans le fichier `stdio.c` mais appelle d'autres fonctions définies dans d'autres fichiers. Au total plus de 300 lignes de programme sont nécessaires pour décrire correctement cette fonction. Imaginez-vous à chaque appel, recopier ces 300 lignes?
En conclusion, le concept de fonction n'est pas un outil purement décoratif : il s'agit d'un concept fondamental en informatique qu'il faut maîtriser pour pouvoir programmer.

1.2 Prototypes des fonctions

Le prototype d'une fonction permet de savoir comment celle-ci peut-être utilisée. Il permet de préciser le nom d'appel de la fonction, ses paramètres et le type de sa valeur de retour (valeur renvoyée par la fonction).
En `C`, il est d'usage de donner le prototype d'une fonction en utilisant la même syntaxe que pour sa déclaration:

```
type_retour nom_de_la_fonction (liste des paramètres sous la forme "type nom")
```

Exemple 1. Le prototype

```
int add(int a, int b);
```

signifie que la fonction de nom `add` possède deux paramètres de type `int` nommés `a` et `b` et renvoie une valeur de type `int`.

Remarque 1. Attention, le prototype d'une fonction ne contient que des informations d'ordre syntaxique. Il ne permet pas de savoir ce que fait une fonction. Par exemple,

```
int add(int a, int b) {
  return a - b;
}
```

réalise une soustraction et non une addition!
C'est le rôle du programmeur de bien choisir le nom d'une fonction et de documenter son utilisation, soit dans un manuel soit sous forme de commentaires directement avant la déclaration dans le programme. Par exemple, la déclaration de la fonction précédente aurait du être sous cette forme:

```
/*
 * sub: réalise la soustraction de deux valeurs de type int
 * AE: Aucune
 * AS: sub == a - b
 */

int sub(int a, int b);
```

Nous décrirons un peu plus loin la forme précise que devront avoir les commentaires.

Une fonction peut ne pas renvoyer de valeur. Ceci est précisé en utilisant le type `void`.

Exemple 2. Le prototype

```
void inc(int *pa);
```

signifie que la fonction de nom `inc` prend un paramètre de type `int␣*` mais ne renvoie pas de valeur.
Pour éviter les confusions, il est recommandé, dans un premier temps, de préfixer par `p` les noms des variables de type pointeur.

Une fonction peut ne pas avoir de paramètre. Dans ce cas, le mot clef `void` est écrit entre les parenthèses suivant le nom[1].

[1] Il est autorisé aussi de ne rien mettre entre les parenthèses.

Exemple 3. Le prototype

```
void print_helloworld(void);
```

signifie que la fonction de nom `print_helloworld` ne prend aucun paramètre ni ne renvoie de valeur.

Remarque 2. Ce semestre nous demandons à ce que la syntaxe du prototype coïncide avec celle de la déclaration. Néanmoins, dans la pratique, la syntaxe des déclarations peut se présenter sous diverses formes. Par exemple, la fonction `scanf` est déclarée dans le fichier `stdio.h`. Le code de sa déclaration utilise une macro-constante:

```
#define _EXFUN(N,P) N P
int     _EXFUN(scanf, (const char *, ...));
```

Le code obtenu est équivalent au prototype:

```
int scanf (const char *format, ...);
```

Les `...` font parties du prototype, ils signifient que `scanf` est une fonction variadique, c'est à dire qu'elle accepte un nombre quelconque de variable en paramètre (en plus de celles spécifiées). Nous n'approfondirons pas plus cette notion ce semestre.

Parfois (comme dans la fonction `scanf`), le mot clef `const` est apposé devant la déclaration d'un paramètre de type adresse. Cela signifie que la valeur située à cette adresse ne sera pas modifiées par la fonction. Ceci est plus qu'une indication, car l'utilisation de ce mot clef empêche la modification en signalant une erreur lors de la compilation. C'est une garantie pour l'utilisateur de la fonction que l'intégrité de ses données sera conservée. Ce sera en particulier très utile lorsque l'on manipulera des tableaux.

1.3 Appel d'une fonction

Pour pouvoir appeler une fonction, il faut que celle-ci soit déclarée avant son premier appel soit dans le fichier lui-même, soit dans un fichier séparé et appelé grâce à `#include` .
L'appel d'une fonction se fait en écrivant son nom suivi de ses paramètres dans l'ordre donné par le prototype.

Exemple 4. Considérons la fonction dont la déclaration est

```
/* inc: ajoute 1 à la variable de type int située à l'adresse pa
 * AE: Aucune
 * AS: La valeur située à l'adresse pa a été augmentée de 1
 *     par rapport à sa valeur avant l'appel
 */

void inc(int *pa};
```

Le code

```
int a = 0;
printf("%d\n", a);
inc(&a);
printf("%d\n", a);
```

affiche

```
0
1
```

L'effet de `inc(&a)` a été d'ajouter `1` au contenu de la variable `a`.

Lorsqu'une fonction renvoie une valeur cette dernière peut être utilisée dans une expression.

Exemple 5. Considérons le prototype suivant:

```
/*
 * add: réalise l'addition de deux valeurs de type int
 * AE: Aucune
 * AS: add == a + b
 */

int add(int a, int b);
```

Le code

```
int x = 1;
int y = 3;
printf("%d", add(x, y) / 2);
```

affiche `2`. La valeur de retour de `add(x, y)` est `4`. Elle a été substituée dans l'expression pendant l'évaluation. Ceci donne `4␣/␣2` qui est évalué en `2`.

Remarque 3. La valeur de retour d'une fonction peut être utilisée dans un appel de fonction. Par exemple, le code

```
int x = 1;
int y = 1;
int z = -2;
print("%d", add(add(x, y), z));
```

affiche la valeur `0` à l'écran. La valeur de retour de `add(x,␣y)` est `2`. Elle a été substituée dans l'expression pour donner `add(2,␣z)`. Cette expression a été évaluée à son tour pour donner `0`.

Le résultat de l'appel d'une fonction n'est garanti que si les conditions d'utilisation décrites dans le descriptif sont respectées. C'est ce que l'on appellera les assertions d'entrée.
De façon symétrique, les assertions de sorties décrira l'état des variables après l'appel à la fonction. La valeur de retour sera symbolisée par une variable fictive ayant le nom de la fonction.

Exemple 6. La déclaration suivante

```
/* div:
* Renvoie le résultat de la division d'un entier a
* de type int et d'un entier non nul b de type int
* AE:  La valeur de b est non nulle
* AS:  div == a / b
*/
int div(int a, int b);
```

Le descriptif indique explicitement qu'aucun résultat ne peut être garanti si `b␣==␣0` dans le cas contraire elle renvoie la valeur résultant de l'expression `a␣/␣b`. Cela implique qu'avant l'appel il faut s'assurer de la non nullité de la variable `b`.
On peut modifier cette fonction afin qu'elle soit valide pour toutes les valeurs du type `int`. Cela implique que l'on se dote d'une stratégie de gestion des erreurs. On peut, par exemple, utiliser la variable globale `errno`, proposée dans la bibliothèque `errno.h` et qui sert à signaler des erreurs.

```
/* div:
 * Prend deux entiers a et b de type int en paramètre
 * Si b != 0  alors elle renvoie a / b
 * Si b == 0 alors elle assigne -1 à la variable errno
 * AE:  Aucune
 * AS:  (errno == -1 && b == 0) ^^ (div == a / b && b != 0)
 */
int div(int a, int b);
```

Lorsque `b␣==␣0` la fonction modifie la valeur de la variable `errno`; charge à l'utilisateur de tester cette valeur afin de s'assurer que la valeur de retour de la fonction est pertinente. Dans les deux cas la fonction (`b == 0` ou `b != 0`) doit renvoyer une valeur; lorsque `b == 0` la valeur ne doit pas être utilisée par la suite. On remarque ici qu'une assertion est une proposition logique qui a une valeur booleenne. Cela sous-entend que le cas particulier `b␣==␣0` est detecté après l'appel à la fonction grâce à l'examen de la valeur de `errno`. L'utilisateur peut donc traiter ce cas en affichant, par exemple un message d'erreur.
Le symbole `&&` désigne l'opérateur `et` et le symbole`^^` désigne le `ou` exclusif (`xor`). Nous reviendrons plus loin sur la syntaxe précise.

Remarque 4. La forme proposée pour la description des fonctions en commentaire n'est pas standard. Elle est imposée par soucis d'homogénéités et pour bien mettre en valeur certaines notions. Les assertions d'entrées et de sorties sont rarement données explicitement mais elles se déduisent de la description littérale.

Remarque 5. Pour appeler une fonction variadique, la liste de longueur variable se situe à la fin des paramètres. Les appels aux fonctions `printf` et `scanf`, vues dans les chapitres précédents, donnent des exemples d'utilisation.

2 Écrire des fonctions

2.1 Structure d'un programme comprenant des fonctions

Lorsque l'on veut ajouter une fonction dans un programme, il y a deux étapes à réaliser: la déclaration et la définition.

2.1.1 Déclaration

La déclaration est une étape optionnelle dans le sens où le programme peut fonctionner du moment où la définition est écrite avant la première utilisation de la fonction. La définition jouera alors aussi le rôle de la déclaration. Néanmoins, pour des raisons de lisibilité, il est fortement conseillé de ne pas omettre l'étape de déclaration.
En principe, il est possible de déclarer des fonctions à n'importe quel endrois mais, encore pour des raisons de lisibilité, seules deux solutions sont recommandées:

- **Déclarer les fonctions dans un fichier externe**[2]. Les fonctions sont déclarées dans un fichier d'extension `.h` et appelées en début de programme grâce à une instruction `#include`. C'est la solution retenue en général par les développeurs car non seulement elle permet de structurer des programmes complexes mais elle donne la possibilité d'appeler les mêmes fonctions dans plusieurs programmes. Les fichiers regroupent généralement les fonctions par thème. On parle alors de bibliothèque de fonctions. Par exemple, les fonctions d'entrée/sortie sont déclarées dans le fichier `stdio.h` qui est chargé par la commande `#include<stdio.h>` placée en début du fichier.

- **Déclarer les fonctions dans le fichier principal.**[3] Les fonctions sont déclarées dans le programme principal, juste avant la définition de la fonction `main`. C'est la solution que nous priviligierons ce semestre pour écrire des programmes peu complexes.

La déclaration consiste à donner le prototype de la fonction précédé d'un commentaire de description. La description devra suivre le schémas suivant

```
/* nom de la fonction: description littérale de son comportement.
 * [ Entrées: liste des paramètres avec leurs types et leurs rôles]
 * [ Sortie: type et signification de la valeur renvoyée ]
 * AE: Assertion d'entrée
 * AS: Assertion de sortie
 */
```

1. En premier lieu doit apparaitre le nom et le descriptif précis de la fonction.

2. Les entrées désignent les paramètres de la fonction. Les crochets indiquent que cette partie est optionnelle car souvent les informations contenues dans cette rubrique peuvent être déduites de la description littérale et de la rubrique `AE`.

[2] Les fichiers d'extension `.h` s'appellent des header.
[3] Celui dans lequel se trouve la fonction `main`.

3. La sortie désigne la valeur renvoyée. Les crochets indiquent que cette partie est optionnelle car souvent les informations contenues dans cette rubrique peuvent être déduites de la description littérale et de la rubrique `AS`.

4. L'acronyme `AE` signifie assertion d'entrée. Cela désigne une expression logique faisant intervenir les paramètres avant l'exécution de la fonction et dont la valeur doit être vraie pour que le résultat soit garanti.

5. L'acronyme `AS` signifie assertion de sortie. Cela désigne une expression logique dont est vraie une fois la fonction exécutée.

Une assertion est une proposition ayant une valeur de vérité. Elle peut être écrite de façon littérale ou bien en utilisant des symboles logiques. Dans ce dernier cas, il est recommandé d'utiliser des expressions de logiques booleennes ainsi qu'un opérateur additionnel, noté `^^` afin de désigner le ou exclusif. Pour l'assertion de sortie, le nom de la fonction désignera la valeur renvoyée. Lorsqu'aucune condition n'est requise dans les assertions ont pourra écrire `vraie`, `true` ou `aucune`.

Exemple 7. Reprenons l'exemple 6 de la section précédente.

```
/*  div:
*  Renvoie le résultat de la division d'un entier a de type int
*  et d'un entier non nul b de type int
 *  Entrées: deux entiers a et b de type int.
 *  Sortie: une valeur de type int
 *  AE:  b != 0
 *  AS:  div == a / b
 */
int div(int a, int b);
```

Dans cet exemple, nous voyons que les informations données dans les rubriques `Entrées` et `Sortie` peuvent être reconstituées à partir de la description littérale et des rubriques `AE` et `AS`.
Pour que le résultat soit garanti, il faut que l'expression booléenne `b␣!=␣0` soit évaluées à `true` autrement dit il faut que la variable `b` contienne une valeur non nulle.
À la fin de la fonction, la valeur `a␣/␣b` sera renvoyée, c'est ce que signifie l'expression booléenne `div␣==␣a␣/␣b`.
Nous avions aussi examiné une version alternative de cette déclaration

```
/* div:
 * Prend deux entiers a et b de type int en paramètre
 * Si b != 0  alors elle renvoie a / b
 * Si b == 0 alors elle assigne -1 à la variable errno
 * Entrées: deux entiers a et b de type int
 * Sortie: une valeur de type int
 * AE:  Aucune
 * AS:  (errno == -1 && b == 0) ^^ (errno == 0 && div == a / b && b != 0)
 */
int div(int a, int b);
```

Pour bien comprendre l'expression de l'assertion de sortie, il faut interpréter le symbole `^^` par ou bien. À la fin de l'appel de la fonction, l'une des deux situations (disjointes) est valide

- soit la variable de la valeur `errno` vaut `-1` et le paramètre `b` vaut `0` (la valeur de retour n'est pas spécifiée)
- ou bien la variable de la valeur `errno` vaut `0`, la fonction renvoie la valeur `a␣/␣b` et le paramètre `b` n'est pas nul.

Remarque 6. Afin d'éviter des phrases trop longues dans les assertions, on pourra utiliser le raccourci `old` afin de désigner une valeur avant l'appel à la fonction. La rubrique assertion d'entrée de l'exemple 4 peut aussi s'écrire

```
* AS: *pa == old(*pa) + 1
```

Notez que ce raccourci n'est à utiliser que lorsqu'il y a une ambiguïté. En toute rigueur les assertions de sortie la première version de la fonction `div` de l'exemple 6 devraient être

```
* AS:  div == old(a) / old(b)
```

Cela alourdi fortement la notation alors que le contexte suggère que les paramètres `a` et `b` n'ont pas été modifiés lors de l'exécution de la fonction. Il est donc, dans ce cas, préférable de ne pas utiliser `old`.

2.1.2 Définition

La définition d'une fonction est la partie du fichier qui contient le code de celle-ci. En principe, la définition d'une fonction peut être placée à n'importe quel endroit du programme. Néanmoins, pour des raisons de lisibilité, seules deux solutions sont recommandées:

- **Définir les fonctions dans un fichier externe.** Comme pour les déclarations, il est possible (et en général souhaitable) de définir les fonctions dans des fichiers séparés. Ces fichiers doivent être compilés séparément puis il faut une étape de construction afin de lier le programme principal aux différents sous programmes. Pour un programme complexe, il est nécessaire de créer un fichier `Makefile` décrivant l'ordre de compilation des différents fichiers le constituant. C'est la solution qui est en général retenue par les programmeurs car elle permet de pouvoir faire évoluer les fonctions sans modifier les prototypes ni les programmes les appelant.
- **Définir les fonction dans le fichier principal.** Lorsque les programmes sont simples, il est envisageable de définir directement les fonctions dans le fichier principal. Ce sera l'option que l'on retiendra ce semestre. Dans ce cas, la définition doit être placée après la fonction `main`, en n'oubliant pas de déclarer la fonction avant.

Pour résumer, tout programme devra avoir la structure suivante:

```
/* Nom du programme
 * Date de réalisation
 * Auteur
 * Descriptif
 */
```

```
/* Appel des bibliothèques */
#include ...
...

/* Déclarations des fonctions et des macros*/
....
....

/* Fonction principale */
int main() {
...
}

/* Définitions des fonctions */
....
```

Les définitions des fonctions devront, de préférence, être écrites dans le même ordre que leurs déclarations.
Pour définir une fonction, il faut d'abord rappeler le prototype puis écrire le corps de la fonction, c'est à dire la liste des instructions qui la compose, entre accolades.

Exemple 8. Voici un exemple complet de programme

```
/* Illustration d'une fonction d'incrémentation
 * 26 Aout 2024
 * Jean-Gabriel Luque
 * Ce programme permet d'illustrer la notion de fonction
 * en développant l'exemple d'une fonction d'incrémentation
 */

/* Appel des bibliothèques */
#include<stdio.h>
#include<stdlib.h>

/* Déclarations des fonctions */
/* inc: ajoute 1 à la variable de type int située à l'adresse pa
 * Entrée: pa de type int *. C'est l'adresse à laquelle se trouve
 *         la valeur qui va être incrémentée.
 * Sortie: Aucune
 * AE: Aucune
 * AS: *pa == old(*pa) + 1
 */

void inc(int *pa);
```

```
/* Fonction principale */
int main(void) {
  int a = 0;
  printf("%d\n", a);
  inc(&a);
  printf("%d\n", a);
  return EXIT_SUCCESS;
}

/* Définitions des fonctions */
void inc(int *pa) {
  *pa += 1;
}
```

Remarquez l'utilisation de `&a` lors de l'appel à la fonction `inc` qui signifie que c'est l'adresse de la variable `a` qui est passée en paramètre.

2.2 Paramètres et type d'une fonction

Lorsque l'on écrit une fonction, il faut préciser quels sont ses paramètres et quel est le type de la valeur renvoyée (lorsqu'il y en a une).
Chaque paramètre doit être clairement décrit (nom, type et rôle) dans les commentaires précédant la déclaration.
Dans le prototype de la fonction, le type est placé à gauche du nom et la liste des paramètres est placée à droite du nom et entre parenthèses. Chaque paramètre est décrit par son type suivi de son nom. Les paramètres sont séparés par une virgule. Lors de l'appel à une fonction, il faut veiller à bien respecter l'ordre des paramètres. Les paramètres peuvent être de n'importe quel type y compris des pointeurs, ce qui peut être pratique lorsque l'on veut modifier le contenu d'une variable.
Une fonctions peut renvoyer une valeur et dans ce cas elle doit être du même type que celui indiqué devant le prototype. Dans le corps de la fonction, l'instruction `return` sert à renvoyer la valeur. Lorsqu'une instruction `return` est exécutée, la fonction est arrêtée, l'expression située à droite du `return` est renvoyée et vient se substituer au nom de la fonction dans l'expression où se situe l'appel (nous verrons cela plus en détail dans la section suivante).

Exemple 9. Considérons le programme suivant

```
...
/*
 * sub: soustrait le second paramètre au premier.
 * Entrées: a et b de type int.
 * Sortie: une valeur de type int
 * AE: Aucune
 * AS: sub == a - b
 */

int sub(int a, int b);
```

```
int main(void) {
  int x = 2;
  int y = 3;
  printf("%d", sub(x, y));
  return EXIT_SUCCESS;
}

int sub(int a, int b) {
 return a - b;
}
```

Lors de son exécution, il affiche `-1` à l'écran.
Pour exécuter l'instruction `printf("%d",␣sub(x,␣y))`, le compilateur évalue d'abord l'expression `sub(x,␣y)`:

- La valeur de chaque variable est substituée à son nom. Ce qui donne `sub(2,␣3)`.
- La fonction `sub` est appelée avec les affectations `a␣=␣2` et `b␣=␣3`.
- La fonction est exécutée et renvoie la valeur `-1` qui se substitue syntaxiquement à son appel. Le programme exécute donc la commande `printf("%d",␣-1)`. Ceci affiche `-1` à l'écran.

Notez que l'ordre des paramètres est important. Si l'on avait écrit `printf("%d",␣sub(y,␣x))`, le programme aurait affiché `1` et non `-1`.

Remarque 7. Attention, la lecture de `return` arrête la fonction; aucune instruction suivant le `return` ne sera exécutée.

Remarque 8. `main` est bien une fonction qui ne prend aucun paramètre[4] et renvoie une valeur de type `int`. C'est pour cette raison que la commande `return` apparait dans le corps de cette fonction. La valeur de retour permet de mentionner si l'exécution du programme s'est bien déroulée ou non. Elle peut être utilisée par des programmes externes. La commande système `echo␣$?` permet d'afficher la dernière valeur renvoyée par un programme exécuté (`$?` désigne une variable d'environnement).

Une fonction peut ne pas renvoyer de valeur. Dans ce cas, il faut faire précéder le nom de la fonction par le mot clef `void` dans la déclaration et la définition. Il ne faut pas écrire l'appel de la fonction dans une expression à évaluer et l'utilisation de `return` dans le corps de la fonction n'est pas nécessaire.

Exemple 10. Reprenons le programme donné dans l'exemple 8. La fonction `inc` ne renvoie aucune valeur. Le mot clef `return` n'est pas utilisé. Lors de l'appel à la fonction, l'adresse de la variable `a` est passée en paramètre; cette adresse est stockée dans la variable `pa` de type `int*` (pointeur sur un `int`). La ligne `*pa␣+=␣1` permet de modifier la valeur située à l'adresse stockée dans `pa`, c'est à dire la valeur stockée dans la variable `a` de la fonction `main`.
Dans la fonction `main`, l'appel à `inc` n'est pas placé dans une expression.

[4] Pour l'instant... nous verrons un peu plus tard que l'on peut lui en donner.

Remarque 9. Les fonctions ne peuvent pas renvoyer plus d'une valeur. Si on veut pouvoir récupérer plusieurs valeurs après exécution, il est possible d'utiliser des paramètres contenant des adresses.
Par exemple, considérons le programme suivant:

```
...
 /*  quot: Renvoie le résultat du quotient d'un entier x
 *       de type int et d'un entier non nul y
 *       de type int. Place le reste de la division dans la variable
 *       prem de type int*.
 *  Entrées: deux entiers x et y de type  int et un pointeur prem vers un type int
 *  Sortie: une valeur de type int
 *  AE:  y != 0
 *  AS:  quot == x / y && *prem == x % y
 */
int quot(int x, int y, int *prem);
...
int main(void) {
  int a = 14;
  int b = 4;
  int q;
  int r;
  q = quot(a, b, &r);
  printf("%d %d\n", q, r);
  return EXIT_SUCCESS;
}

...
int quot(int x, int y, int *prem) {
  *prem = x % y;
  return x / y;
}
```

Celui-ci affichera `3␣2`. En effet, le pointeur `prem` est utilisé pour récupérer la valeur du reste de la division.

3 Que se passe-t-il lors de l'appel d'une fonction?

3.1 Blocs mémoires et portée d'une variable

Lorsqu'un programme s'exécute, une zone de la mémoire lui est allouée afin qu'il puisse réaliser ses opérations. Le langage `C` permet de définir des variables ainsi que des blocs mémoires. Dans les deux cas, il s'agit de zone de la mémoire allouées pour réaliser certaines opérations. Nous avons déjà étudier ce que sont les variables dans un chapitre précédent. Les blocs mémoires sont délimités dans un programme par des accolades. Lorsque le programme "lit" une accolade ouvrante, il créé un bloc mémoire et, lorsque l'accolade fermante correspondante est lue, le bloc mémoire est détruit. À l'intérieur d'un bloc, on peut définir des variables. Ces variables seront utilisables à l'intérieur

de ce bloc et uniquement à l'intérieur de celui-ci. On peut créer des blocs mémoires emboités: un bloc mémoire peut contenir un autre bloc mémoire. Une variable déclarée dans un bloc mémoire pourra être utilisée n'importe lequel de ses sous blocs sauf dans le cas où le nom de la variable est réutilisé dans une déclaration à l'intérieur de ce dernier. Lorsqu'un bloc se termine (accolade fermante), plus aucun des noms de variables déclarées dans ce dernier n'est utilisable.

Remarque 10. On peut définir des variables en dehors des blocs (et donc des fonctions). Ces variables sont dites globales. Elles sont utilisables dans tous les blocs dans lesquels leurs noms n'est pas associé à une autre variable. Notez néanmoins qu'il est déconseillé de déclarer des variables globales.

Afin de pouvoir s'y retrouver dans l'exécution d'un programme, il est fortement recommandé d'adopter une démarche incluant une représentation graphique. Nous vous conseillons la suivante:

- Suivez l'exécution pas à pas de votre programme en numérotant les lignes lues et en commençant à l'accolade ouvrante suivant le nom de la fonction `main`.
- À chaque fois que vous rencontrez une accolade ouvrante vous dessinez un rectangle. À l'intérieur de ce bloc, vous dessinerez les variables et les sous blocs.
- Représentez les variables par des petits rectangle sous lesquels vous placerez leurs noms. Pour les variables de type pointeur, vous pourrez utiliser une flèche qui pointera vers la donnée située à l'adresse qu'elle contient.
- Lors de la création d'une variable remplissez l'espace de son rectangle avec un `?` tant que celle-ci n'a pas été initialisée.
- Lorsque le contenu d'une variable est modifié barrez l'ancien contenu et écrivez le nouveau à côté.
- Lorsque qu'un symbole `}` est lu, barrez le bloc correspondant pour signifier qu'aucune des variables définies dans celui-ci n'est accessible.
- Réservez un espace sur votre feuille pour dessinez ce qu'il se passe à l'écran (entrée/sortie).
- Numérotez chaque étape de votre dessin (mémoire et écran) avec la même numérotation que les lignes de votre programme.

Exemple 11. Considérons le programme suivant:

```
...
#include<assert.h>
...
int main(void) {
  int a,b;
  assert (scanf("%d%d", &a, &b) == 2);
  {
    int a = 0;
    a = a + b;
    int c = a + b;
```

```
    printf("%d %d\n", a, c);
  }
  printf("%d %d\n", a ,b);
  return EXIT_SUCCESS;
}
...
```

Nous allons simuler son exécution pour l'entrée 1␣2 saisie par l'utilisateur. Nous allons donc suivre pas à pas l'exécution du programme en numérotant les lignes, les dessins et l'affichage à l'écran.

```
...
int main(void) { (1)
  int a,b; (2)
  assert(scanf("%d%d", &a, &b) == 2); (3)
  { (4)
    int a = 0 (5);
    a = a + b; (6)
    int c = a + b; (7)
    printf("%d %d\n", a, c); (8)
  } (9)
  printf("%d %d\n", a, b); (10)
  return EXIT_SUCCESS; (11)
} (12)
...
```

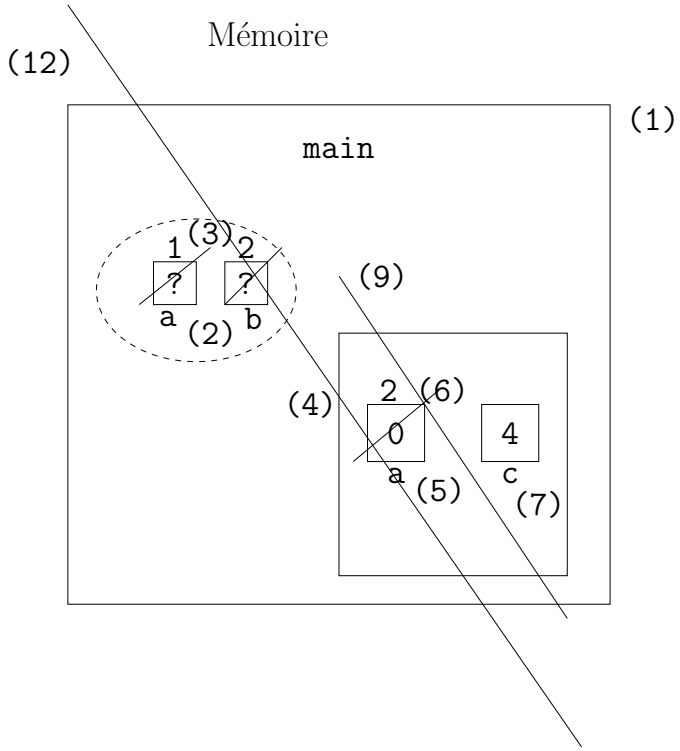
Écran

1 2 (3)

2 4 (8)

1 2 (10)

Mémoire



1. Création du bloc mémoire associé à la fonction `main`

2. Déclaration des variables `a` et `b` dans le bloc de `main`. Pour l'instant, aucune valeur ne leur a été affectée. Ceci est symbolisé grâce au caractère `?`

3. Saisie du contenu des variables `a` et `b` par l'utilisateur. Il y a deux actions: la saisie apparaît à l'écran et les variables sont affectées.

4. Création d'un bloc mémoire.

5. Déclaration de la variable `a` qui est interne au bloc `(4)-(9)`. On lui affecte la valeur `0`. Remarquez que la variable a le même nom que celle du bloc `main`

6. La valeur de la variable `a` est modifiée. Attention, il s'agit de la variable du bloc `(4)-(9)` et non de celle du bloc `main`. Cette dernière n'est pas modifiée.

7. Déclaration et affectation de la variable `c`. L'expression à droite utilise le nom de la variable `a` qui est celle du bloc `(4)-(9)` et celui de la variable `b` qui est celle du bloc `main`.

8. On affiche le contenu des variables `a` et `c`. Les deux sont celles définies dans le bloc `(4)-(9)`.

9. Destruction du bloc `(4)-(9)`. Les variables déclarées à l'intérieur de celui-ci ne seront plus accessibles. À partir de cette ligne, le nom `a` désignera la variable déclarée dans le bloc `main`.

10. Affichage des valeurs des variables `a` et `b` du bloc `main`. On constate que le bloc `(4)-(9)` ne les a pas modifiées.

11. Renvoie de la valeur `EXIT_SUCCESS` et terminaison de la fonction `main`.

12. Destruction du bloc associé à la fonction `main` et fin du programme.

Exemple 12. Considérons le programme suivant:

```
...
int main(void) {
  int a, b;
  assert(scanf("%d%d", &a, &b) == 2);
  {
    int *pc;
    pc = &b;
    a = a + b;
    *pc = a + b;
    printf("%d %d\n", a,* pc);
  }
  printf("%d %d\n", a, b);
  return EXIT_SUCCESS;
}
...
```

Nous allons simuler son exécution pour l'entrée `1␣2` saisie par l'utilisateur. Nous allons donc suivre pas à pas l'exécution du programme en numérotant les lignes, les dessins et l'affichage à l'écran.

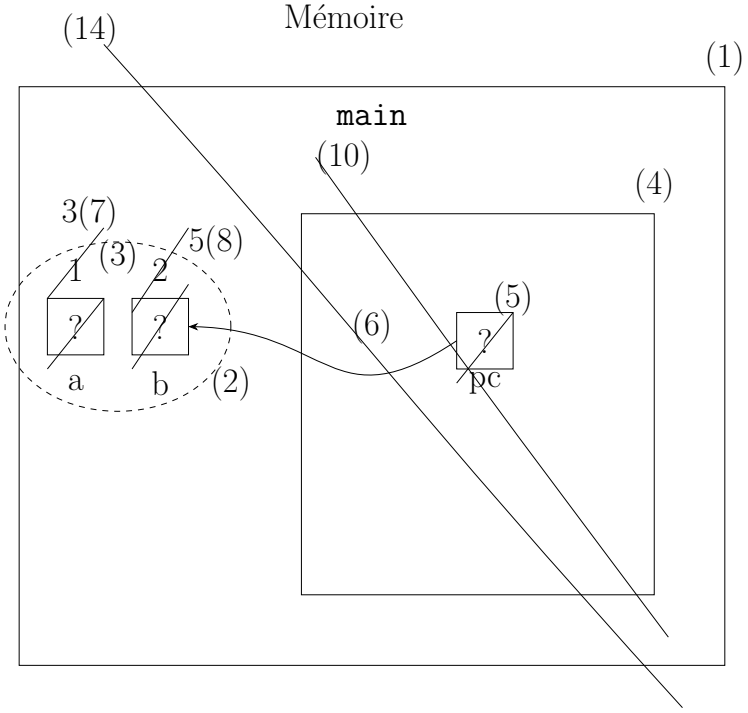
```
...
int main(void) { (1)
  int a, b; (2)
  assert(scanf("%d%d", &a, &b) == 2); (3)
  { (4)
    int *pc; (5)
    pc = &b; (6)
    a = a + b; (7)
    *pc = a + b; (8)
    printf("%d %d\n", a, *pc); (9)
  } (10)
  printf("%d %d\n", a, b); (11)
  return EXIT_SUCCESS; (12)
} (13)
...
```

Écran

1 2 (3)

3 5 (9)

3 5 (11)



1. Création du bloc mémoire associé à la fonction `main`
2. Déclaration des variables `a` et `b` dans le bloc de `main`. Pour l'instant, aucune valeur ne leur a été affectée. Ceci est symbolisé grâce au caractère `?`
3. Saisie du contenu des variables `a` et `b` par l'utilisateur. Il y a deux actions: la saisie apparaît à l'écran et les variables sont affectées.
4. Création d'un bloc mémoire.
5. Déclaration de la variable `pc` qui est interne au bloc `(4)-(9)`. C'est un pointeur vers un entier. Pour l'instant, aucune valeur ne lui a été affectée.

6. La valeur de la variable `pc` est modifiée. Elle contient maintenant l'adresse de la variable `b`.
7. Modification du contenu de la variable `a`. Il s'agit de la variable qui a été déclarée dans le bloc associé à la fonction `main`.
8. Modification de la valeur située à l'adresse `pc`. Cette adresse correspond à l'adresse de la variable `b`. C'est donc le contenu de `b` qui est modifié.
9. On affiche le contenu des variables `a` et `*pc`. Ce qui revient à afficher le contenu de `a` et `b`.
10. Destruction du bloc `(4)-(9)`. Les variables déclarées à l'intérieur de celui-ci ne seront plus accessibles.
11. Affichage des valeurs des variables `a` et `b` du bloc `main`. On constate que le bloc `(4)-(9)` les a modifiées.
12. Renvoie de la valeur `EXIT_SUCCESS` et terminaison de la fonction `main`.
13. Destruction du bloc associé à la fonction `main` et fin du programme.

3.2 Mécanisme d'appel d'une fonction

Lorsqu'une fonction est appelée, un bloc mémoire lui est associé dans lequel sont déclarées les variables dont les valeurs sont passées en paramètre. Les valeurs passées en appel sont affectées à ces variables. Les seules variables utilisables dans ce bloc sont les paramètres, les variables déclarées dans celui-ci et les variables globales.
Lorsqu'une fonction est terminée, ses variables ne sont plus accessibles et les valeurs qu'elles contenaient sont considérées comme perdues.
Une fonction se termine lorsque la lecture atteint le symbole `}` de fin de bloc. Si un mot clef `return` est lu, alors la fonction termine et la valeur obtenue par l'évaluation de l'expression placée à droite est renvoyée.
Pour s'y retrouver, il est conseillé d'adopter une représentation graphique similaire à celle présentée à la section précédente.

Exemple 13. Considérons le programme suivant

```
...
int sub(int a, int b);
...
int main(void) {
  int a, b, c;
  assert(scanf("%d%d%d",&a, &b, &c) == 3);
  printf("%d", sub(a, sub(b, c)));
  return EXIT_SUCCESS;
}
...
int sub(int a, int b) {
 return a - b;
}
...
```

Supposons que l'utilisateur saisisse 1␣2␣3. L'orde de lecture des lignes est le suivant

```
...
int sub(int a, int b);
...
int main(void) { (1)
  int a, b, c; (2)
  assert(scanf("%d%d%d", &a, &b, &c) == 3); (3)
  printf("%d", sub(a, sub( b, c))); (4)
  return EXIT_SUCCESS; (5)
} (6)
...
int sub(int a, int b) { (4.a) (4.d)
 return a - b; (4.b) (4.e)
} (4.c) (4.f)
...
```

Quelques remarques:

- Pour bien montrer que la fonction `sub` est appelée par la ligne (4), nous avons préfixé la numérotation des lignes par "`4.`".
- Les lignes de `sub` peuvent être exécutées plusieurs fois (ici deux fois chacune).
- Dans ce programme, l'ordre d'exécution des lignes ne dépend pas des valeurs saisies par l'utilisateur.

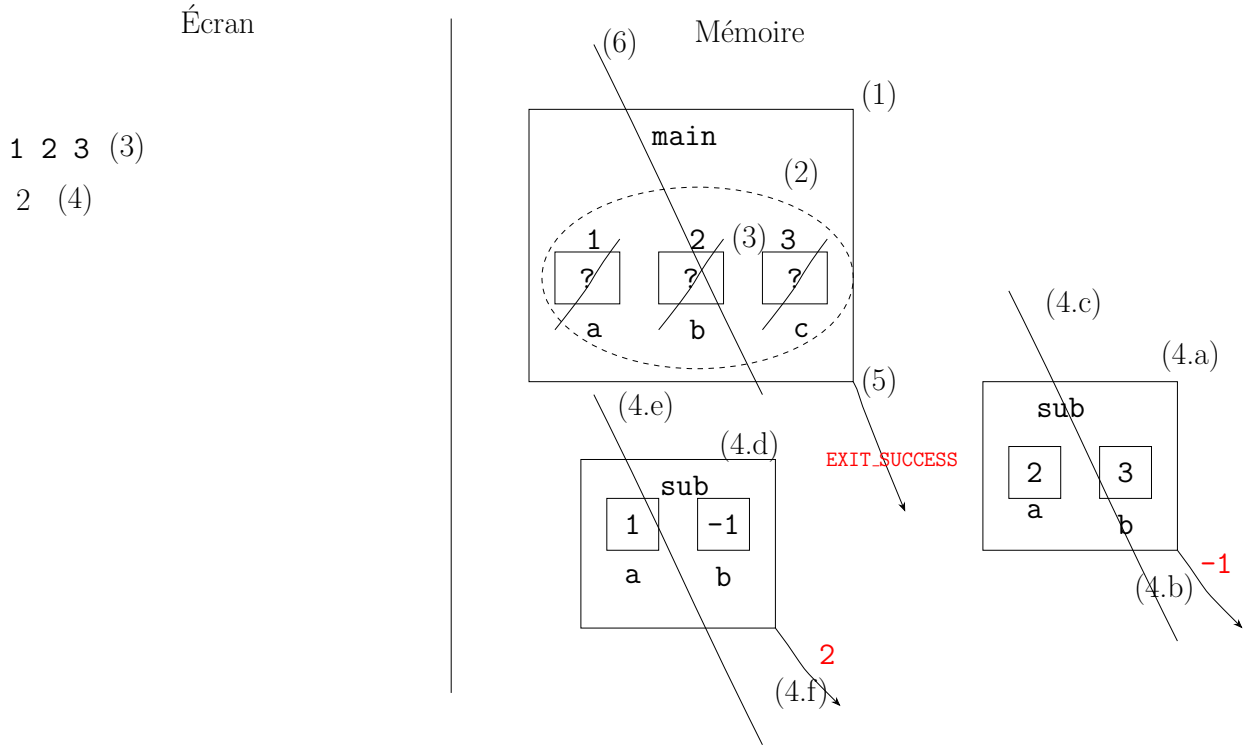
Voici un dessin représentant, de façon simplifiée, ce qui se passe en mémoire:

1. Création du bloc dédié à la fonction `main`
2. Déclaration des variables `a b c`.
3. Saisie du contenu des variables `a b c` par l'utilisateur.
4. (a) Création du bloc dédié au premier appel à la fonction `sub`. Attention, il y a deux appels imbriqués. La procédure d'évaluation demande d'abord à ce que l'appel le plus profond soit exécuté. Ici c'est la partie rouge de la commande `printf("%d\n",␣sub(a,␣sub(b,␣c)))` qui est évaluée. Les valeurs des variables `b` et `c` de la fonction `main` sont affectées aux paramètres `a` et `b` de la fonction `sub`.

   (b) La valeur `-1` est renvoyée et vient se substituer syntaxiquement à l'appel de la fonction. Le renvoie d'une valeur est symbolisé par une flêche sud-est surmontée de la valeur en rouge.
   La commande qui s'exécute "devient" `printf("%d\n",␣sub(a,␣-1))`.

   (c) Fin du premier appel à la fonction `sub`. La zone reservée à cette fonction n'est plus accessible.

   (d) Création du bloc dédié au second appel à la fonction `sub`. Le paramètre `a` de la fonction `sub` prend la valeur de la variable `a` de la fonction `main`. Le paramètre `b` de la fonction `sub` prend la valeur `-1`.



(e) La valeur `2` est renvoyée et vient se substituer syntaxiquement à l'appel de la fonction. La commande qui s'exécute "devient" `printf("%d\n",␣2)`

(f) Fin du second appel à la fonction `sub`. La zone reservée à cette fonction n'est plus accessible.

Affichage de `2` à l'écran.

5. Renvoie de la valeur `EXIT_SUCCESS`.

6. Fin de la fonction `main` et du programme. La zone reservée à `main` n'est plus accessible.

Remarquez que l'ordre d'exécution des lignes du programme ne dépend pas des valeurs saisies par l'utilisateur.

Exemple 14. Afin d'illustrer l'utilisation des pointeurs passés en paramètre des fonctions, nous allons considérer le programme suivant:

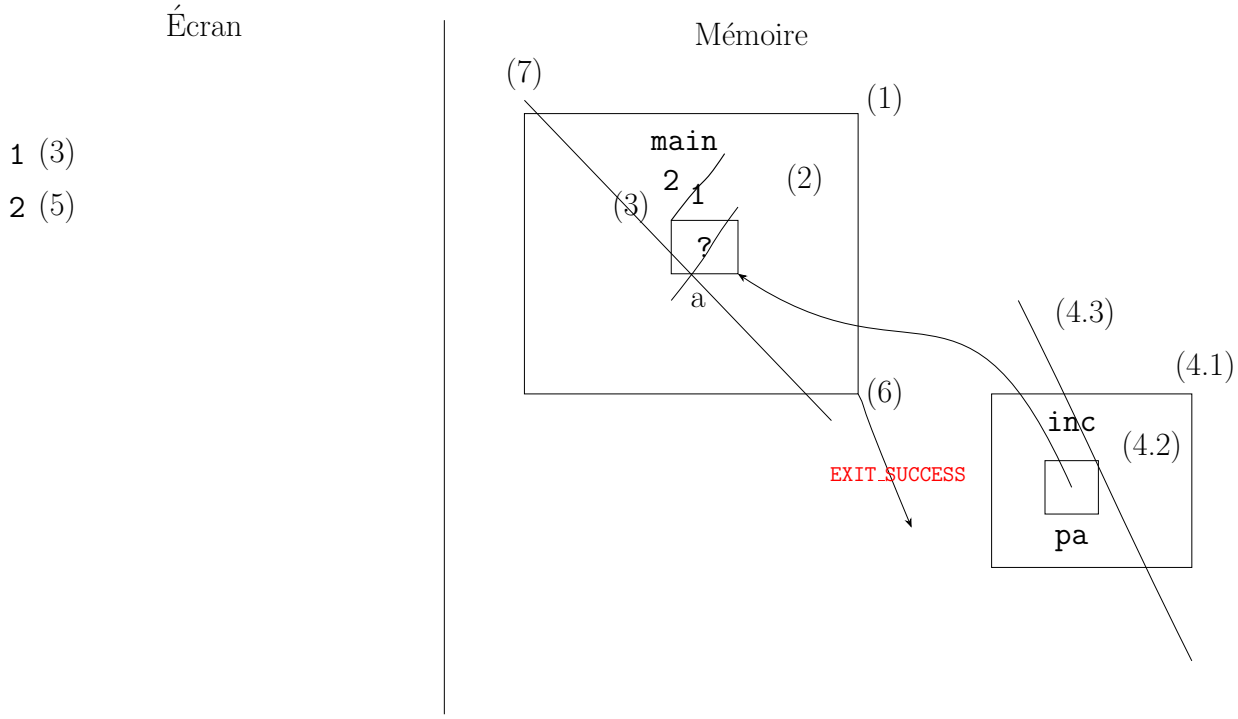
```
...
void inc(int *pa);
....
int main() {
  int a;
  assert(scanf("%d", &a) == 1);
  inc(&a);
  printf("%d", a);
  return EXIT_SUCCESS;
}
```

```
....
void inc(int *pa) {
  *pa+= 1;
}
```

Voici l'ordre dans lequel il s'exécute:

```
...
void inc(int *pa);
....
int main(void) {(1)
  int a; (2)
  assert(scanf("%d",&a) == 1); (3)
  inc(&a); (4)
  printf("%d", a); (5)
  return EXIT_SUCCESS; (6)
} (7)
....
void inc(int *pa) { (4.a)
  *pa+= 1; (4.b)
} (4.c)
```

Si on suppose que l'utilisateur saisisse la valeur `1`, le déroulement de l'exécution peut se schématiser de la façon suivante:



1. Création du bloc dédié à la fonction `main`
2. Déclaration de la variable `a`.
3. Saisie du contenu de la variable `a` par l'utilisateur (valeur `1`).

4. (a) Création du bloc dédié au premier appel à fonction `inc`. La valeur de la variable `pa` en paramètre est l'adresse de la variable `a` en mémoire.
   (b) La valeur contenue à l'adresse pointée par `pa` est augmentée de `1`. Autrement dit: on ajoute `1` au contenu de la variable `a` de la fonction `main`.
   (c) Fin de l'appel à la fonction `sub`. La zone reservée à cette fonction n'est plus accessible.

   Remarquez qu'il n'y a aucune valeur de retour à la fonction `inc`. Son appel n'est pas placé dans une expression.
5. Affichage de la valeur de la variable `a` qui contient `2`.
6. Renvoie de la valeur `EXIT_SUCCESS`.
7. Fin de la fonction `main` et du programme. La zone reservée à `main` n'est plus accessible.

3.3 Trace d'un programme

Les dessins, que l'on a introduits précédement, ne seront pas demandé en contrôle. Ils ne sont là qu'à titre indicatif. Ils permettent de se forger une représentation mentale de ce qui se passe en mémoire lors de l'exécution d'un programme.
En revanche, la notion de trace de programme est quelque chose de plus formalisé. Écrire la trace d'un programme consiste à décrire chaque étape et chaque transition de ligne et à notifier l'état des variables utilisées ainsi que les entrées/sorties dans un tableau.
Les lignes devront être numérotés et chaque variable (du moins les plus importantes) devra apparaître en colonne. Si une variable n'a pas été déclarée ou n'est pas accessible dans la bloc, on notera `ND` (ou rien si il n'y a pas d'ambiguité). On pourra néanmoins garder des variables qui ne sont pas dans la fonction courante mais dont la valeur est modifiée grâce à un pointeur.
Si c'est utile à la compréhension, on peut aussi faire figure des valeurs pointées. Si une variable a été déclarée mais qu'aucune valeur ne lui a été affectée, on le mentionnera en écrivant un "`?`".
On pourra aussi mentionner dans une colonne, la fonction qui est en train d'être lue. Lorsque deux variables ont le même nom, il faut préciser les blocs auxquels elles appartiennent. On pourra aussi utiliser le symbole `|` pour préciser que la variable existe mais n'est pas accessible dans le bloc courant ainsi qu'un code couleur pour désigner des zones mémoires identiques.

Exemple 15. Reprenons l'exemple 14. On numérote les lignes (on peut numéroter toutes les lignes du programme ou seulement celles que l'on va utiliser).

```
....
1 int main() {
2  int a;
3  assert(scanf("%d", &a) == 1);
4  inc(&a);
5  printf("%d", a);
6  return EXIT_SUCCESS;
7 }
....
8 void inc(int *pa) {
9  *pa+= 1;
10 }
```

On repère les variables qui vont nous être utiles. ici on a besoin de la variable `a` de la fonction `main` et la variable `pa` de la fonction `inc`. Nous allons décrire la trace du programme lorsque l'utilisateur a saisi `0` sur l'entrée standard.

a	pa	*pa	affichage	lignes de code	fonction courante
0	ND			1 2 3	main
0|	&a	0		8	inc
1|	&a	1		9	inc
1	ND			4	main
1	ND		1	5	main
1	ND			6 7	fin

Si le programme la commande `assert` provoque l'arrêt du programme, on le mentionnera en écrivant "échec".

Exemple 16. Si l'entrée standard est vide dans l'exemple précédent, la trace du programme est

a	pa	affichage	lignes de code	fonction courante
?	ND		1 2	main
?	ND			échec

Lorsque deux variables ont le même nom, il faut préciser les blocs auxquels elles appartiennent.

Exemple 17. Reprenons l'exemple 14 mais en utilisant le nom `a` comme paramètre de `inc`.

```
....
1 int main() {
2  int a;
3  assert(scanf("%d", &a) == 1);
4  inc(&a);
5  printf("%d", a);
6  return EXIT_SUCCESS;
7 }
....
8 void inc(int *a) {
9  *a+= 1;
10 }
```

La trace du programme pour la valeur `0` sur l'entrée standard est

$a^\dagger$	$a^\star$	$*a^\star$	affichage	lignes de code	fonction courante
0	ND			1 2 3	main
0|	$\&a^\dagger$	0		8	inc
1|	$\&a^\dagger$	1		9	inc
1	ND			4	main
1	ND		1	5	main
1	ND			6 7	fin

†: désigne une variable déclarée dans la fonction `main`.
*: désigne une variable déclarée dans la fonction `inc`.

Lorsqu'une fonction est appelée plusieurs fois et que cela ne donne lieu à aucune ambiguité, la même colonne peut être utilisée pour les variables pour les différents appels.

L'information sur nom de la fonction courrante peut-être retrouvée grâce au numéro de la ligne. Lorsqu'il y a beaucoup de colonnes, cette information peut être omise. On peut faire aussi apparaître le détail de l'évaluation d'expression lorsque celui-ci est en lien avec des valeurs renvoyées par des fonctions.

Exemple 18. Reprenons l'exemple 13.

```
...
int sub(int a, int b);
...
1 int main(void) {
2   int a, b, c;
3   assert(scanf("%d%d%d",&a, &b, &c) == 3);
4   printf("%d", sub(a, sub(b, c)));
5   return EXIT_SUCCESS;
6 }
...
7 int sub(int a, int b) {
8 return a - b;
9 }
...
```

Supposons que `1 2 3` soit lu sur l'entrée standard. On obtient alors la trace

a†	b†	c†	a⋆	b⋆	sub(b, c)†	sub(a,sub(b, c))†	affichage	lignes
1	2	3	ND	ND				1 2 3
|	|	|	2	3				7 8 9
1	2	3	ND	ND	-1			4
|	|	|	1	-1				7 8 9
1	2	3	ND	ND	-1	2		4
1	2	3	ND	ND	-1	2	2	4
1	2	3	ND	ND	-1	2		5 6

†: désigne une variable ou une expression de la fonction `main`.

⋆: désigne une variable ou une expression la fonction `sub`.