

Valeur d'une variable

La valeur d'une variable est la raison pour laquelle celle-ci existe. Le rôle premier d'une variable étant en effet de contenir une valeur.

La valeur d'une variable n'est **pas attribuée à sa déclaration** (elle contient à ce moment là une valeur mais il ne faut rien supposer dessus).

On accède à la valeur d'une variable par son identificateur.

On modifie une variable par une **affectation**. L'occurrence de l'identificateur de la variable se trouve dans ce cas à gauche de l'opérateur `=`.

– Exemple –

```
int num;  
num = 23;  
num = num + 32;
```

L'occurrence de `num` en l. 2 est située à gauche du `=` : il s'agit d'une affectation. Il y en a deux en ligne 3 : la 1^{re} permet de modifier et la 2 de lire sa valeur.

L-values et R-values

Nous rencontrons une subtilité : un identificateur `x` de variable peut désigner soit :

1. la valeur de la variable `x`, p.ex., dans `x + 16` ;
2. soit la variable `x` elle-même, p.ex., dans `x += 8`.

La terminologie de valeur gauche (ou encore « *L-value* ») et valeur droite (ou encore « *R-value* ») permet de mettre en évidence cette différence.

Une valeur gauche est une expression qui peut se situer dans le membre gauche d'une affectation (l'expression peut **recevoir** une valeur).

Une valeur droite est une expression qui peut se situer dans le membre droit d'une affectation (une valeur peut être **lue** depuis l'expression).

Note 1 : c'est le contexte qui permet de dire si une expression est une valeur gauche ou une valeur droite.

Note 2 : toute valeur gauche peut-être une valeur droite (pour un contexte différent), mais pas l'inverse.

Adresse d'une variable

L'adresse d'une variable est une valeur entière spécifiant la position de la variable en mémoire.

L'adresse d'une variable est **attribuée à sa déclaration** par le système à l'**exécution**. Elle ne peut pas être choisie par le programmeur ni être modifiée.

On accède à l'adresse d'une variable par son identificateur précédé de l'opérateur `&`.

– Exemple –

```
int num;  
printf("%p\n", &num);
```

Une 1^{re} exécution de ces instructions affiche `0x7fff6a3014fc`. Une 2^e affiche `0x7fffbdc357dc`.
L'adresse de `num` varie d'une exécution à l'autre.

Portée lexicale d'une variable et variables locales

La portée lexicale d'une variable désigne la **zone du programme** dans laquelle la variable peut être utilisée.

Elle dépend de l'endroit dans lequel elle a été déclarée.

Sa portée lexicale s'étend aux instructions qui sont situées après sa déclaration dans le plus petit bloc d'instructions qui la contient.

– Exemple –

```
void f(int x) {  
    int a, b;  
    ...  
}  
  
int g(int y, int z) {  
    int c;  
    ...  
}
```

La portée lexicale des variables **a** et **b** s'étend aux instructions du corps de la fonction **f**. Elle ne s'étend pas aux instructions du corps de **g**. Les variables **a** et **b** sont des variables locales à la fonction **f** et invisibles ailleurs.

Le **paramètre** **x** de **f** a pour portée lexicale uniquement le corps de **f**.

Portée lexicale d'une variable et variables locales

– Exemple –

```
...  
int f() {...}  
...  
int taille = 31;  
...  
int g() {...}  
...  
int main() {...}
```

La portée lexicale de la variable `taille` s'étend à tout ce qui suit sa déclaration dans le programme. Elle est donc visible dans les fonctions `g` et `main` mais pas dans `f`. Étant donné qu'elle est déclarée en dehors de toute fonction, elle est qualifiée de variable globale.

Attention : l'utilisation de variables globales n'est ni élégante ni indispensable. Elle est également source d'erreurs car il est souvent difficile de comprendre un programme les utilisant.

Elle est bannie pour ces raisons.

On préfère utiliser des définitions préprocesseur pour représenter leur valeur.

Portée lexicale d'une variable et blocs d'instructions

– Exemple –

```
{  
    int a;  
    a = 15;  
    printf("%d", a);  
}  
printf("%d", a);
```

Il y a erreur de compilation : la portée lexicale de la variable `a` s'étend de la l. 2 à la l. 4. Elle n'est pas visible à la l. 6. Il n'existe pas de variable identifiée par `a` lorsque la l. 7 est évaluée.

– Exemple –

```
{  
    int a;  
    a = 15;  
    printf("%d", a);  
}  
{  
    printf("%d", a);  
}
```

De la même manière que dans l'exemple précédent, l'occurrence du symbole `a` dans le second bloc (l. 7) n'est pas résolue. Elle se situe dans un bloc qui n'est pas contenu par celui où le symbole `a` est déclaré.

Portée lexicale d'une variable et blocs d'instructions

– Exemple –

```
int a;  
a = 10;  
{  
    printf("%d ", a);  
}  
printf("%d\n", a);
```

Ces instructions produisent l'affichage **10 10**. En effet, la variable **a** est visible dans le bloc d'instructions dans lequel elle est définie, ainsi que dans les blocs d'instructions qui se trouvent à l'intérieur.

– Exemple –

```
int a;  
a = 10;  
{  
    int a;  
    a = 20;  
    printf("%d ", a);  
}  
printf("%d\n", a);
```

Ces instructions produisent l'affichage **20 10**. La variable identifiée par **a** dans le bloc d'instructions de la l. 3 à la l. 7 est celle déclarée en l. 4. La variable identifiée par **a** hors de ce bloc d'instructions est celle déclarée en l. 1.

Portée lexicale d'une variable et blocs d'instructions

– Exemples –

Comparons les instructions

```
int a;  
a = 10;  
{  
    int a;  
    a = 20;  
    printf("%d ", a);  
}  
printf("%d\n", a);
```

```
int a;  
a = 10;  
{  
    a = 20;  
    printf("%d ", a);  
}  
printf("%d\n", a);
```

Dans le cas de gauche (déjà vu), l'affectation `a = 20` n'a d'effet que sur la variable `a` déclarée à l'intérieur du bloc. Ceci affiche `20 10`.

En revanche, les instructions de droite affichent `20 20` car il n'y a pas de déclaration de `a` dans le bloc. L'affectation `a = 20` modifie la variable `a` déclarée en l. 1.

Plan

1. Bases

- 1.1 Généralités
- 1.2 Expressions et instructions
- 1.3 Constructions syntaxiques
- 1.4 Variables
- 1.5 Fonctions et pile**
- 1.6 Commandes préprocesseur

Fonctions

Une fonction est constituée

1. d'un identificateur (qui suit les mêmes contraintes que ceux des variables);
2. d'une signature (la liste de ses paramètres et de leurs types);
3. d'un type de retour (le type de la valeur renvoyée par la fonction);
4. d'instructions (qui forment le corps de la fonction).

La ligne constituée du type de retour, de l'identificateur et de la signature d'une fonction est son prototype.

– Exemple –

```
int produit(int a, int b, int c) {  
    return a * b * c;  
}
```

est une fonction d'identificateur `produit`, de signature `(int a, int b, int c)` et de type de retour `int`.

Définition vs déclaration

La définition d'une fonction consiste à fournir tous ses constituants.

La déclaration d'une fonction consiste à fournir son **prototype**.

Déclarer une fonction est utile si l'on souhaite s'en servir avant de l'avoir définie.

– Exemple –

```
#include <stdio.h>

/* Declarations. */
void flop(int nb);
void flip(int nb);

/* Definitions. */
int main() {
    flip(10);
    return 0;
}
```

```
void flip(int nb) {
    if (nb >= 1) {
        printf("flip\n");
        flop(nb - 1);
    }
}
```

```
void flop(int nb) {
    if (nb >= 1) {
        printf("flop\n");
        flip(nb - 1);
    }
}
```

Paramètres vs arguments

Il faut faire attention à bien distinguer les notions de paramètre et d'argument qui sont deux choses différentes.

On considère la fonction

```
int produit(int a, int b, int c) {  
    return a * b * c;  
}
```

Les symboles `a`, `b` et `c` de son prototype sont ses paramètres.

Lors de l'appel

```
produit(15, num, -3);
```

les expressions `15`, `num` et `-3` sont les arguments de l'appel.

Aide-mémoire : paramètre ↔ prototype ; argument ↔ appel.

Portée lexicale des paramètres

Les variables locales d'une fonction ont pour portée lexicale la fonction elle-même (déjà mentionné).

Il en est de même pour ses **paramètres** : leur portée lexicale est la fonction elle-même. On peut voir la déclaration des paramètres d'une fonction dans son en-tête comme une déclaration de variable.

– Exemple –

```
int doubl(int a) {  
    return 2 * a;  
}  
  
int main() {  
    int a;  
    a = 10;  
    a = doubl(a + 1);  
    return 0;  
}
```

Il y a plusieurs occurrences du symbole `a`.

Celui déclaré dans l'en-tête de `doubl` a une portée lexicale qui s'étend de la l. 1 à la l. 2.

Celui déclaré dans le `main` a pour portée lexicale le `main` tout entier.

Ce sont des variables différentes.

Pile

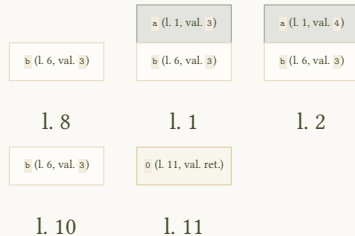
Lors de l'appel d'une fonction, les valeurs de ses arguments sont **recopiées** dans une zone de la mémoire appelée pile.

Conséquence très importante : toute modification des paramètres dans une fonction ne modifie pas les valeurs des arguments avec lesquels elle a été appelée.

– Exemple –

```
void incr(int a) {  
    a = a + 1;  
}  
  
int f() {  
    int b;  
  
    b = 3;  
    incr(b);  
    printf("%d\n", b);  
    return 0;  
}  
...  
f();
```

L'appel à `f` en l. 14 produit les configurations de pile



Pile

Les **variables locales** d'une fonction (c.-à-d. les variables déclarées dans le corps de la fonction) se situent dans la **pile**.

De plus, la **valeur renvoyée** (si son type de retour n'est pas `void`) se situe dans la **pile**.

Après avoir appelé une fonction, c.-à-d. juste après avoir renvoyé la valeur de retour, la pile se trouve dans le même état qu'avant l'appel.

Conséquence très importante : toute variable locale à une fonction est non seulement invisible mais n'existe plus en mémoire hors de la fonction et après son appel.

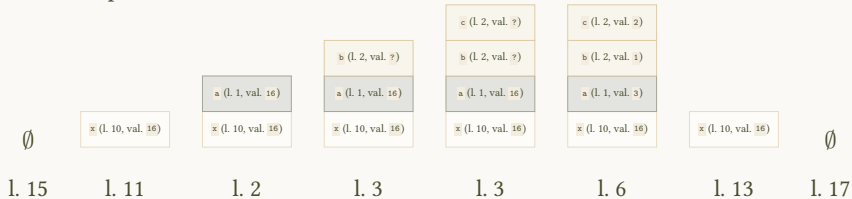
Pile

– Exemple –

```
void fct_1(int a) {  
    int b, c;  
    b = 1;  
    c = 2;  
    a = b + c;  
}  
  
void fct_2() {  
    int x;  
    x = 16;  
    fct_1(x);  
    printf("%d\n", x);  
}
```

fct_2();

Configurations de pile :



Pile et fonctions récursives

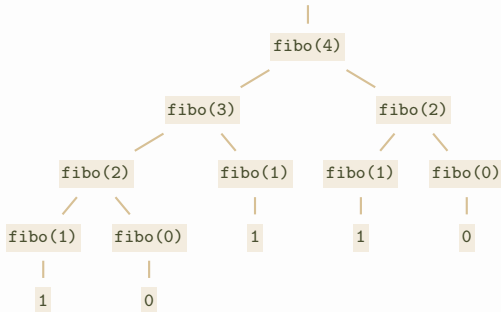
Soit la fonction

```
int fibo(int a) {  
    if (a <= 1)  
        return a;  
    return fibo(a - 1) + fibo(a - 2);  
}
```

et la suite d'instructions

```
int x;  
x = fibo(4);
```

On représente son exécution par un **arbre des appels** :



Fonctions à effet secondaire

Une **fonction** est à effet secondaire s'il existe au moins un jeu d'arguments qui fait que l'évaluation de l'appel à la fonction sur ce jeu d'arguments modifie la mémoire par rapport à son état d'avant l'appel.

– Exemple –

```
int f(int a, int b) {  
    return 21 * a + b;  
}
```

Cette fonction n'est pas à effet secondaire. Elle renvoie une valeur sans modifier la mémoire.

– Exemple –

```
float double_val(float *x) {  
    *x = 2 * (*x);  
    return *x;  
}
```

Cette fonction est à effet secondaire puisqu'elle modifie une zone de la mémoire (celle à l'adresse spécifiée par son argument).

Fonctions à effet secondaire

– Exemple –

```
int g(char c) {  
    int b;  
    b = 5;  
    return b + c;  
}
```

Cette fonction n'est pas à effet secondaire. La déclaration (et l'affectation) de `b` reste locale à la fonction. Après tout appel à `g`, la variable `b` n'existe plus.

– Exemple –

```
char *allouer(int n) {  
    char *res;  
    res = (char *) malloc(sizeof(char) * n);  
    return res;  
}
```

Cette fonction est à effet secondaire. Elle réserve en effet, par l'appel interne à la fonction `malloc`, une zone de la mémoire, ce qui modifie son état.

– Exemple –

```
int h(int a, int b) {  
    if (a * b == 0)  
        printf("z\n");  
    return a - b;  
}
```

Cette fonction est à effet de secondaire. En effet, l'appel à `h` avec, p.ex., les arguments `1` et `0` provoque un affichage sur la sortie standard, modifiant l'état de la mémoire.

1. Bases

- 1.1 Généralités
- 1.2 Expressions et instructions
- 1.3 Constructions syntaxiques
- 1.4 Variables
- 1.5 Fonctions et pile
- 1.6 Commandes préprocesseur**

Commandes préprocesseur

Une commande préprocesseur (ou directive préprocesseur) est une ligne qui commence par #.

Le préprocesseur est une unité qui intervient lors de la compilation. Son rôle est de traiter les commandes préprocesseur.

Il fonctionne en **construisant une nouvelle version** du programme en **remplaçant** chaque commande préprocesseur par des expressions en C adéquates.

Il existe plusieurs sortes de commandes préprocesseur :

- les inclusions de fichiers ;
- les définitions de symboles ;
- les macro-instructions à paramètres ;
- les macro-instructions de contrôle de compilation.

Inclusions de fichiers

La commande préprocesseur

```
#include <NOM.h>
```

permet d'**inclure** le fichier `NOM.h` dans le programme pour bénéficier des fonctionnalités qu'il apporte.

Le préprocesseur résout cette commande en recopiant le contenu de `NOM.h` à l'endroit où elle est invoquée.

Il est possible d'enchaîner les inclusions :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Habituellement, les inclusions sont réalisées au début du programme.

Définitions de symboles

La commande préprocesseur

```
#define SYMB EXP
```

permet de **définir un alias** `SYMB` pour l'expression `EXP`. Ceci autorise à faire référence à l'expression `EXP` par l'intermédiaire du symbole `SYMB`.

Le préprocesseur résout tout invocation `SYMB` en la remplaçant par `EXP`.

– Exemple –

À gauche (resp. à droite), des instructions avant (resp. après) le passage du préprocesseur :

```
#define NB 5
#define CHAINE "cba\n"
...
for (i = 1 ; i <= NB ; ++i) {
    printf("%s", CHAINE);
}
```

```
/* Rien. */
/* Rien. */
...
for (i = 1 ; i <= 5 ; ++i) {
    printf("%s", "cba\n");
}
```

Par convention, tout alias est constitué de lettres majuscules, de chiffres ou de tirets bas.

Macro-instructions à paramètres

La commande préprocesseur

```
#define SYMB(P1, P2, ..., Pn) EXP
```

permet de définir une **macro-instruction à paramètres** `SYMB`. Ceci autorise à faire référence à l'expression `EXP` par l'intermédiaire du symbole `SYMB` paramétrable par des paramètres `P1`, `P2`, ..., `Pn`.

Le préprocesseur résout toute invocation `SYMB(A1, A2, ..., An)` en la remplaçant par l'expression obtenue en substituant `Ai` à toute occurrence du paramètre `Pi` dans `EXP`.

– Exemple –

À gauche (resp. à droite), des instructions avant (resp. après) le passage du préprocesseur :

```
#define MAX(a, b) a > b ? a : b
...
int x;
x = MAX(10, 14);
```

```
/* Rien. */
...
int x;
x = 10 > 14 ? 10 : 14;
```

Macro-instructions à paramètres

Problème : étudions comment le préprocesseur transforme les instructions suivantes :

```
#define CARRE(a) a * a
...
x = 2;
y = 3;
z = CARRE(x + y);
```

La l. 5 est remplacée par `z = x + y * x + y;`.

Ainsi, la valeur `2 + 3 * 2 + 3 = 11` est affectée à `z` au lieu de `25` comme attendu.

Solution : il faut placer des parenthèses autour des paramètres des macro-instructions à paramètres :

```
#define CARRE(a) (a) * (a)
```

De cette façon, `CARRE(x + y)` est remplacée par `(x + y) * (x + y)` comme désiré.

Macro-instructions à paramètres

Problème : étudions comment le préprocesseur transforme les instructions suivantes :

```
#define DOUBLE(a) (a) + (a)
...
x = 3;
z = 5 * DOUBLE(x);
```

La l. 4 est remplacée par `z = 5 * (x) + (x);`.

Ainsi, la valeur `5 * 3 + 3 = 18` est affectée à `z` au lieu de `30` comme attendu.

Solution : il faut placer des parenthèses autour de l'expression toute entière :

```
#define DOUBLE(a) ((a) + (a))
```

De cette façon, `5 * DOUBLE(x)` est remplacée par `5 * ((x) + (x))` comme désiré.

Macro-instructions à paramètres

Attention, l'usage de macro-instructions à paramètres peut provoquer des **évaluations multiples** d'une même expression.

Considérons par exemple `#define MAX(a, b) ((a) > (b) ? (a) : (b))` et son emploi dans

```
res = MAX(f(t1), f(t2));
```

où `res` est une variable entière, `f` est une fonction qui renvoie un entier et `t1` et `t2` sont des arguments acceptés par `f`.

Cette dernière instruction est remplacée par le préprocesseur en

```
res = ((f(t1)) > (f(t2)) ? (f(t1)) : (f(t2)));
```

Problème : ceci réalise un appel

- à `f` avec l'argument `t1` (ce qui est normal);
- à `f` avec l'argument `t2` (ce qui est normal);
- à `f` avec l'argument `t1` ou `t2` (ce qui est de trop).

Macro-instructions de contrôle de compilation

Les **macro-instructions de contrôle de compilation** permettent d'ignorer, lors de la compilation, une partie du programme.

Ceci est utile pour **sélectionner** les parties à prendre en compte dans un programme, sans avoir à les (dé)commenter.

On dispose ainsi des constructions

```
#ifdef SYMB  
...  
#endif
```

```
#ifndef SYMB  
...  
#endif
```

```
#ifdef SYMB  
...  
#else  
...  
#endif
```

À gauche, le code ... n'est considéré que si l'alias **SYMB** est défini.

Au centre, le code ... n'est considéré que si l'alias **SYMB** n'est pas défini.

Macro-instructions de contrôle de compilation

– Exemple –

```
#include <stdio.h>

#define GAUCHE_DROITE

#ifdef GAUCHE_DROITE

int rechercher(char *tab,
               int n, char x) {
    int i;
    for (i = 0 ; i < n ; ++i)
        if (tab[i] == x)
            return i;
    return -1;
}
```

```
#else

int rechercher(char *tab,
               int n, char x) {
    int i;
    for (i = n - 1 ; i >= 0 ; --i)
        if (tab[i] == x)
            return i;
    return -1;
}

#endif
```

```
int main() {
    int res;
    char tab[] = "chaine de test";

    res = rechercher(tab, 14, 't');
    printf("%d\n", res);
}
```

Ici, on donne deux algorithmes pour localiser la première occurrence d'une lettre dans un tableau : de la gauche vers la droite, ou bien de la droite vers la gauche.

Ce programme affiche 10 ; si on renomme GAUCHE_DROITE (en l. 4), il affiche 13.

Plan

2. Habitudes

2. Habitudes

2.1 Mise en page

2.2 Gestion d'erreurs

2.3 Assertions d'entrée

Mise en page d'un programme

Pour écrire un programme de valeur, il faut soigner les points suivants :

1. l'indentation ;
2. l'organisation des espaces autour des unités lexicales ;
3. le choix des identificateurs ;
4. la documentation.

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au PYTHON, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter la lisibilité.

Règle : on place **quatre espaces** (pas de tabulation) avant chaque instruction d'un bloc et on incrémente cet espacement de quatre en quatre en fonction de la profondeur des blocs.

– Exemples –

```
/* Correct. */
a=_8;
if_(b_>=_0){
    printf("%d\n",_a);
    a=_0;
    for_(i=_0;_i<=_b;_++i)
        a/_=_2;
}
```

```
/* Incorrect. */
a=_8;
if_(b_>=_0){
    printf("%d\n",_a);
    a=_0;
    for_(i=_0;_i<=_b;_++i)
        a/_=_2;
}
```

Organisation des blocs

Nous avons vu qu'un bloc est une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

– Exemples –

```
/* Correct. */  
if (valeur >= 1) {  
    valeur -= 1;  
}
```

```
/* Incorrect. */  
if (valeur >= 1)  
{  
    valeur -= 1;  
}
```

– Exemples –

```
/* Correct. */  
valeur = 1;  
{  
    int a;  
    valeur = 10;  
}
```

```
/* Incorrect. */  
valeur = 1; {  
    int a;  
    valeur = 10;  
}
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

– Exemple –

/ Correct. */*

```
a_ = _b_*_2+_5;
```

/ Incorrect. */*

```
a_=_b*_2+_5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

– Exemple –

/ Correct. */*

```
f(a,_b,_c,_16);
```

/ Incorrect. */*

```
f(a,b,c,16);
```

On ne place **pas d'espace** après une **parenthèse ouvrante** ou avant une **parenthèse fermante**.

– Exemple –

/ Correct. */*

```
a_=(f(a,_3)+_a)*_2;
```

/ Incorrect. */*

```
a_=(f(_a,_3)+_a)*_2;
```

Choix des identificateurs

Les **identificateurs** doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

– Exemples –

```
/* Identificateur non explicite. */  
v
```

```
/* Identificateur trop long. */  
valeur_choisie_pour_le_nombre_parties
```

```
/* Identificateur acceptable. */  
nb_parties
```

On fixe la langue au **français** pour leur construction.

Choix des identificateurs

■ Identificateurs de variables.

Les seuls identificateurs d'une lettre autorisés sont `i`, `j`, `k`, etc., pour les indices de boucles. Les majuscules sont interdites dans les identificateurs. Dans un identificateur composé de plusieurs mots, ces derniers sont séparés par des sous-tirets.

– Exemples –

```
/* Correct. */  
nb_parties
```

```
/* Incorrect. */  
nbParties
```

■ Identificateurs de fonctions. Mêmes règles que pour les identificateurs de variables.

■ Identificateurs des macro-instructions.

Ils sont écrits exclusivement en majuscules et, s'ils sont composés de plusieurs mots, ils sont séparés par des sous-tirets.

– Exemple –

```
#define TAILLE_MAX 1024  
#define DEBUG
```

■ Identificateurs de types.

La première lettre commence par une majuscule et, s'ils sont composés de plusieurs mots, chaque première lettre de chaque mot commence par une majuscule.

– Exemple –

```
typedef ... Liste;  
typedef ... ArbreBinaire;
```

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre `/*` et `*/`.

On documente chaque **fichier de programme**, au tout début, par

- les prénoms et noms des auteurs ;
- la date de création (au format jour-mois-année) ;
- la date de modification (au format précédent).

On commentera le moins possible les instructions.

Il faut impérativement éviter les commentaires inutiles.

– Exemple –

```
/* Auteur : A. M. Turing  
 * Creation : 23-06-1912  
 * Modification : 07-06-1954 */
```

– Exemple –

```
/* Incorrect. */  
/* Affiche la valeur de 'a'. */  
printf("%d\n", a);
```

Documentation des fonctions

On documente la plupart des **fonctions** par des commentaires situés avant leur déclaration (ou leur définition).

Un commentaire de fonction explique

- le **rôle** de chaque **paramètre** ;
- ce que **renvoie** la fonction ;
- l'**effet** produit par la fonction.

– Exemples –

```
/* Correct. */  
/* Renvoie le plus grand entier  
 * parmi 'a' et 'b'. */  
int max(int a, int b) {  
    if (a >= b)  
        return a;  
    return b;  
}
```

```
/* Incorrect. */  
/* Calcule le maximum de deux  
 * entiers. */  
int max(int a, int b) {  
    if (a >= b)  
        return a;  
    return b;  
}
```