Plan

Variables

Une variable est une entité constituée des cinq éléments suivants :

Variables

Une variable est une entité constituée des cinq éléments suivants :

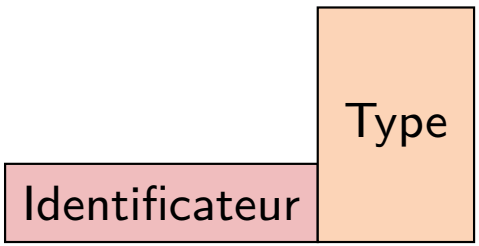
1. un identificateur ;

Identificateur

Variables

Une variable est une entité constituée des cinq éléments suivants :

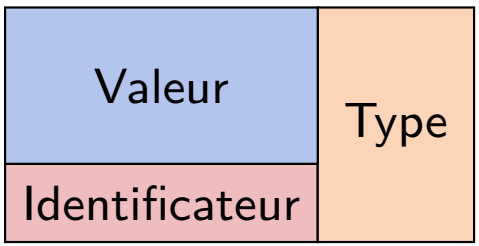
1. un identificateur ;
2. un type ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

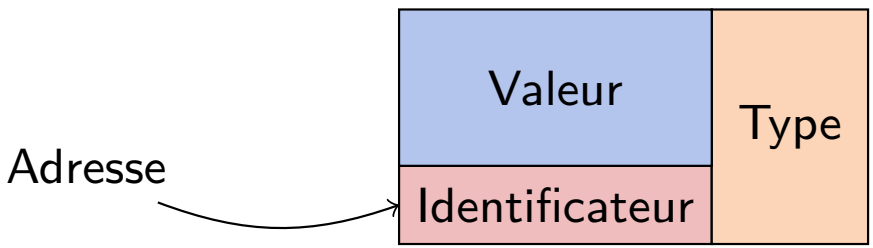
1. un identificateur ;
2. un type ;
3. une valeur ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

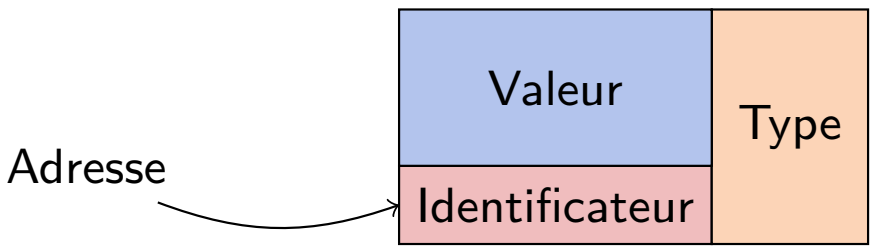
1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

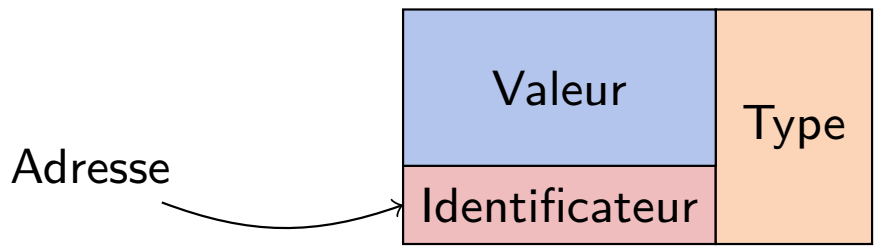
1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;
5. une portée lexicale.



Variables

Une variable est une entité constituée des cinq éléments suivants :

1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;
5. une portée lexicale.



Intuitivement, c'est une boîte qui peut contenir un objet (valeur) et qui dispose d'un nom (identificateur).

Une boîte ne peut contenir que des objets d'une certaine sorte (type).

Elle se situe de plus à un endroit bien précis dans la mémoire (adresse) et elle n'est visible qu'à partir de certains endroits du code (portée lexicale).

Identificateurs de variable

L'identificateur d'une variable est un mot commençant par une lettre ou bien '_', suivi par un nombre arbitraire de lettres, chiffres ou '_'.

De plus, aucun identificateur ne peut-être un mot réservé du langage. En voici la liste complète :

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

Identificateurs de variable

L'identificateur d'une variable est un mot commençant par une lettre ou bien `'_'`, suivi par un nombre arbitraire de lettres, chiffres ou `'_'`.

De plus, aucun identificateur ne peut-être un mot réservé du langage. En voici la liste complète :

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

L'identificateur d'une variable est attribué à sa déclaration.

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.

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).

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.

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 =.

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 =.

```
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 1re permet de modifier et la 2e 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` ;

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`.

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

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

Une *L*-value est une expression qui peut se situer dans le membre gauche d'une affectation (l'expression peut **recevoir** une 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 « *L*-value » (valeur gauche) et « *R*-value » (valeur droite) permet de mettre en évidence cette différence.

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

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

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

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

Une *R*-value 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 *L*-value ou une *R*-value.

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

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

Une *R*-value 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 *L*-value ou une *R*-value.

Note 2 : toute *L*-value peut-être une *R*-value (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.

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.

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 &.

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 &.

```
int num;

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

Une 1re exécution de ces instructions affiche `0x7fff6a3014fc`. Une 2e 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.

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.

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.

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.

```
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.

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.

```
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

```
...
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.

Portée lexicale d'une variable et variables locales

```
...
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 bannie pour ces raisons.

On préfère utiliser des définitions préprocesseur pour représenter leur valeur.

Portée lexicale d'une variable et blocs d'instructions

```
{
    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. 5. Elle n'est pas visible à la l. 6. Il n'existe pas de variable identifiée par `a` lorsque la l. 7 est évaluée.

Portée lexicale d'une variable et blocs d'instructions

```
{
    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. 5. Elle n'est pas visible à la l. 6. Il n'existe pas de variable identifiée par a lorsque la l. 7 est évaluée.

```
{
    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

```
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.

Portée lexicale d'une variable et blocs d'instructions

```
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.

```
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

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);
```

Portée lexicale d'une variable et blocs d'instructions

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`.

Portée lexicale d'une variable et blocs d'instructions

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

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).

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**.

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**.

P.ex.,

```
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.

Définition *vs* déclaration

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

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

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.

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.

Voici un 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.

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;
}
```

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.

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.

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 : **p**aramètre $\leftrightarrow$ **p**rototype ; **a**rgument $\leftrightarrow$ **a**ppel.

Portée lexicale des paramètres

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

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.

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.

```
int double(int a) {
    return 2 * a;
}

int main() {
    int a;
    a = 10;
    a = double(a + 1);
    return 0;
}
```

Il y plusieurs occurrences du symbole `a`.

Celui déclaré dans l'en-tête de `double` a une portée lexicale qui s'étend de la l. 1 à la l. 3.

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

Ce sont des variables différentes.

Pile

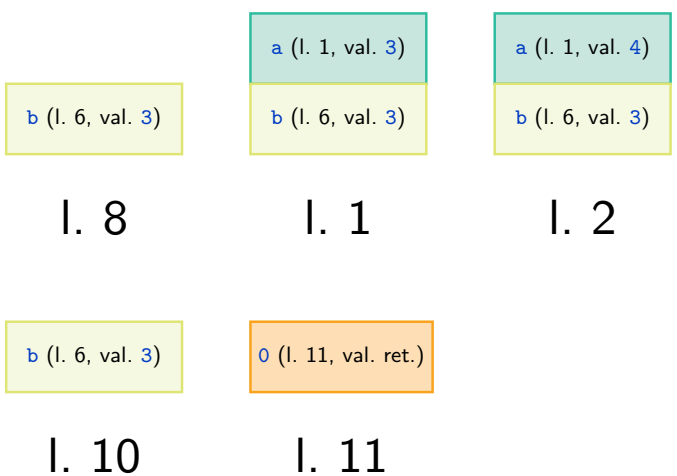
Lors de l'appel d'une fonction, les valeurs de ses arguments sont recopiées dans une zone de la mémoire appelée pile.

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

P.ex.,

```
1   void incr(int a) {
2       a = a + 1;
3   }
4
5   int f() {
6       int b;
7
8       b = 3;
9       incr(b);
10      printf("%d\n", b);
11      return 0;
12  }
13  ...
14  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.

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.

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.

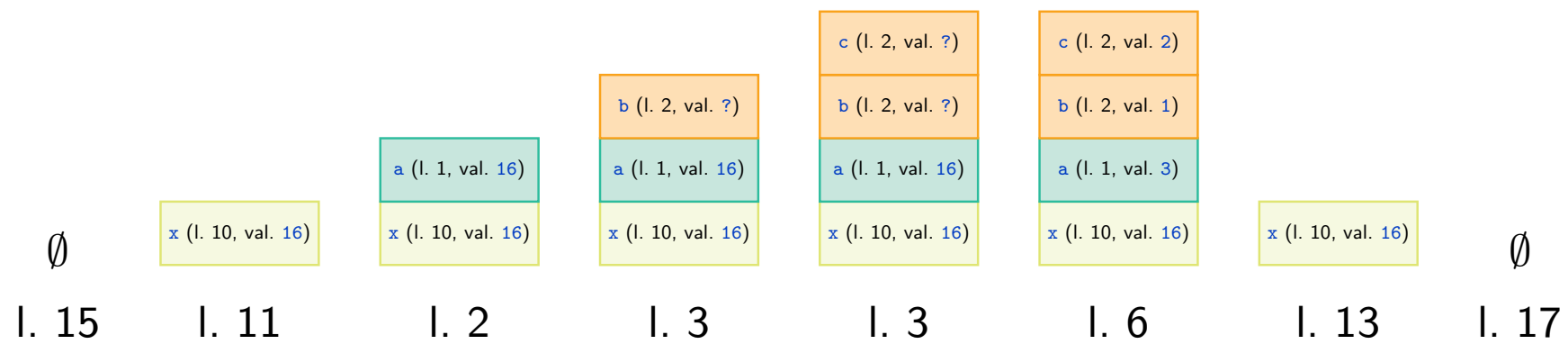
Pile

P.ex.,

```
1   void fct_1(int a) {
2       int b, c;
3       b = 1;
4       c = 2;
5       a = b + c;
6   }
7
8
```

```
9   void fct_2() {
10      int x;
11      x = 16;
12      fct_1(x);
13      printf("%d\n", x);
14  }
15  ...
16  fct_2();
```

Configurations de pile :



Pile et fonctions récursives

Soit la fonction

```
1  void fibo(int a) {
2      if (a <= 1)
3          return a;
4      return fibo(a - 1) + fibo(a - 2);
5  }
```

et la suite d'instructions

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

Dessiner l'arbre des appels et la pile au fur et à mesure de l'exécution.

Fonctions à effet de bord

Une **fonction** est à effet de bord 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.

Fonctions à effet de bord

Une **fonction** est à effet de bord 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.

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

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

Fonctions à effet de bord

Une **fonction** est à effet de bord 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.

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

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

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

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

Fonctions à effet de bord

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

Fonctions à effet de bord

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

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

Fonctions à effet de bord

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

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

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

Fonctions à effet de bord

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

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

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

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

Fonctions à effet de bord

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

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

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

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

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

Fonctions à effet de bord

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

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

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

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

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

Cette fonction est à effet de bord. 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.

Plan

Commandes préprocesseur

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

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.

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.

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.

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>
```

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`.

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`.

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`.

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");
}
```

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`.

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`.

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`.

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`.

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);
```

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.

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)
```

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);
```

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.

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))
```

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 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

```
1  #include <stdio.h>
2
3
4  #define GAUCHE_DROITE
5
6  #ifdef GAUCHE_DROITE
7
8  int rechercher(char *tab,
9          int n, char x) {
10     int i;
11     for (i = 0 ; i <= n ; ++i)
12         if (tab[i] == x)
13             return i;
14     return -1;
15 }
16
17 #else
18
19
```

```
20 int rechercher(char *tab,
21         int n, char x) {
22     int i;
23     for (i = n - 1 ; i >= 0 ; --i)
24         if (tab[i] == x)
25             return i;
26     return -1;
27 }
28
29 #endif
30
31 int main() {
32     int res;
33     char tab[] = "chaine de test";
34
35     res = rechercher(tab, 14, 't');
36     printf("%d\n", res);
37 }
```

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

Plan

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 caractères ;
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 sa 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.

```c
/* Correct. */
a = 8;
if (b >= 0) {
    printf("%d\n", a);
    a = 0;
    for (i = 0; i <= b; ++i)
        a /= 2;
}
```

```c
/* 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 des 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.

```
1  /* Correct. */
2  if (valeur >= 1) {
3      valeur -= 1;
4  }
```

```
1  /* Incorrect. */
2  if (valeur >= 1)
3  {
4      valeur -= 1;
5  }
```

```
1  /* Correct. */
2  valeur = 1;
3  {
4      int a;
5      valeur = 10;
6  }
```

```
1  /* Incorrect. */
2  valeur = 1; {
3      int a;
4      valeur = 10;
5  }
```

Organisation des espaces

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

```
1 /* Correct. */
2 a = b * 2 + 5;
```

```
1 /* Incorrect. */
2 a = b*2 + 5;
```

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

```
1 /* Correct. */
2 f(a, b, c, 16);
```

```
1 /* Incorrect. */
2 f(a,b,c,16);
```

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

```
1 /* Correct. */
2 a = (f(a, 3) + a) * 2;
```

```
1 /* Incorrect. */
2 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**.

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

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

```
1 /* Identificateur acceptable. */
2 nb_parties
```

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

Choix des identificateurs

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.

```
1  /* Correct. */
2  nb_parties
```

```
1  /* Incorrect. */
2  nbParties
```

Exception : les identificateurs de constantes (définies par une instruction pré-processeur) sont écrits exclusivement en majuscules.

```
1  /* Correct. */
2
3  #define TAILLE_MAX 1024
4  #define DEBUG
```

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).

```
1  /* Auteur : L. W. Polsfuss
2   * Creation : 01-01-1952
3   * Modification : 12-08-2009 */
```

On commentera le moins possible les instructions.

Il faut éviter les commentaires inutiles.

```
1  /* Incorrect . */
2  /* Affiche la valeur de 'a'. */
3  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.

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

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