Programmation C

(Perfectionnement à la programmation en C)

Samuele Giraudo

Université Paris-Est Marne-la-Vallée

LIGM, bureau 4B055

samuele.giraudo@u-pem.fr

http://igm.univ-mlv.fr/~giraudo/

Introduction

L'objectif de ce module est d'approfondir les concepts de base et d'approcher certains concepts plus avancés de la programmation en C.

Celui-ci est organisé en trois axes.

1. **Axe 1** : écrire un projet maintenable.

   Bonnes habitudes de programmation, documentation, pré-assertions, gestion des erreurs, programmation modulaire, compilation.

2. **Axe 2** : comprendre les mécanismes de base.

   Pointeurs, allocation dynamique, types, types structurés, entrées/sorties.

3. **Axe 3** : utiliser quelques techniques avancées.

   Opérateurs bit à bit, mémoïsation, génération aléatoire, pointeurs de fonction, généricité.

Contenu du cours

Axe 1.

Axe 2.

Axe 3.

Pré-requis

Ce cours demande les pré-requis suivants :

- des bases en **programmation générale** (notion de programme, d'instruction, de compilation) ;
- des bases en **programmation impérative** (notion de variable, de structures de contrôle, de fonction) ;
- des bases en **algorithmique** (manipulation de structures de données élémentaires comme les tableaux, les chaînes de caractères, les listes) ;
- des bases en **programmation en** C (écriture de fonctions, manipulation de tableaux, de chaînes de caractères, connaissance des types numériques, des types structurés, notion d'allocation dynamique, de pointeur).

Quelques dates

Le langage C est apparu en 1972 dans les laboratoires Bell. Il a été créé par D. Ritchie et K. Thompson au même moment que l'apparition des systèmes UNIX.

Quelques dates

Le langage C est apparu en 1972 dans les laboratoires Bell. Il a été créé par D. Ritchie et K. Thompson au même moment que l'apparition des systèmes UNIX.

Il est influencé par le langage B qui, contrairement au C, ne possédait pas de système de type.

Quelques dates

Le langage C est apparu en 1972 dans les laboratoires Bell. Il a été créé par D. Ritchie et K. Thompson au même moment que l'apparition des systèmes UNIX.

Il est influencé par le langage B qui, contrairement au C, ne possédait pas de système de type.

Quelques dates sur l'évolution de la spécification du langage :

- 1978 : première description complète du langage (C traditionnel) ;
- 1989 : publication de la norme ANSI C (ou C89) ;
- 1995 : évolution du langage C94/95 ;
- 1999 : évolution du langage C99 ;
- 2011 : nouvelle version du standard C11.

Quelques dates

Le langage C est apparu en 1972 dans les laboratoires Bell. Il a été créé par D. Ritchie et K. Thompson au même moment que l'apparition des systèmes UNIX.

Il est influencé par le langage B qui, contrairement au C, ne possédait pas de système de type.

Quelques dates sur l'évolution de la spécification du langage :

- 1978 : première description complète du langage (C traditionnel) ;
- 1989 : **publication de la norme ANSI** C **(ou** C**89)** ;
- 1995 : évolution du langage C94/95 ;
- 1999 : évolution du langage C99 ;
- 2011 : nouvelle version du standard C11.

Quelques caractéristiques

Le C est un langage de bas niveau : il offre la possibilité de se placer « proche » de la machine. Il permet en effet de manipuler finement des données en mémoire, des adresses et des suites de bits.

Quelques caractéristiques

Le C est un langage de bas niveau : il offre la possibilité de se placer « proche » de la machine. Il permet en effet de manipuler finement des données en mémoire, des adresses et des suites de bits.

Ce langage a été initialement pensé pour la conception de systèmes d'exploitation.

Il est cependant suffisamment expressif pour s'adapter à un large éventail d'utilisations.

Quelques caractéristiques

Le C est un langage de bas niveau : il offre la possibilité de se placer « proche » de la machine. Il permet en effet de manipuler finement des données en mémoire, des adresses et des suites de bits.

Ce langage a été initialement pensé pour la conception de systèmes d'exploitation.

Il est cependant suffisamment expressif pour s'adapter à un large éventail d'utilisations.

Il se situe à la croisée des chemins du monde des langages de programmation : beaucoup de langages modernes sont traduits en C pour être compilés.

Quelques caractéristiques

Le C est un langage de bas niveau : il offre la possibilité de se placer « proche » de la machine. Il permet en effet de manipuler finement des données en mémoire, des adresses et des suites de bits.

Ce langage a été initialement pensé pour la conception de systèmes d'exploitation.

Il est cependant suffisamment expressif pour s'adapter à un large éventail d'utilisations.

Il se situe à la croisée des chemins du monde des langages de programmation : beaucoup de langages modernes sont traduits en C pour être compilés.

L'un de ses compilateurs, gcc, fruit de nombreuses optimisations, fait que le C est un langage d'une efficacité extrême.

Axe 1 : écrire un projet maintenable

Plan

Plan

La notion de programme

Un programme en C est avant tout un texte (contenu dans un **fichier**) qui suit certaines règles (dites de **syntaxe**).

La notion de programme

Un programme en C est avant tout un texte (contenu dans un **fichier**) qui suit certaines règles (dites de **syntaxe**).

C'est une collection de déclarations et de définitions de fonctions, de types, de variables globales, assorties de commandes pré-processeur.

La notion de programme

Un programme en C est avant tout un texte (contenu dans un **fichier**) qui suit certaines règles (dites de **syntaxe**).

C'est une collection de déclarations et de définitions de fonctions, de types, de variables globales, assorties de commandes pré-processeur.

Tout programme possède une **fonction principale** nommée `main`. C'est par elle que commence l'exécution du programme. On appelle ceci le point d'entrée du programme.

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

On compile un fichier `Prog.c` par la commande

```
gcc -o NOM Prog.c
```

Ceci produit un exécutable nommé `NOM`.

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

On compile un fichier `Prog.c` par la commande

```
gcc -o NOM Prog.c
```

Ceci produit un exécutable nommé `NOM`.

On compilera obligatoirement avec les **options** `-ansi`, `-pedantic` et `-Wall` au moyen de la commande

```
gcc -o NOM -ansi -pedantic -Wall Prog.c
```

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

On compile un fichier `Prog.c` par la commande

```
gcc -o NOM Prog.c
```

Ceci produit un exécutable nommé `NOM`.

On compilera obligatoirement avec les **options** `-ansi`, `-pedantic` et `-Wall` au moyen de la commande

```
gcc -o NOM -ansi -pedantic -Wall Prog.c
```

- `-ansi -pedantic` : empêche un programme non compatible avec la norme ANSI C de compiler ;

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

On compile un fichier `Prog.c` par la commande

```
gcc -o NOM Prog.c
```

Ceci produit un exécutable nommé `NOM`.

On compilera obligatoirement avec les **options** `-ansi`, `-pedantic` et `-Wall` au moyen de la commande

```
gcc -o NOM -ansi -pedantic -Wall Prog.c
```

- `-ansi -pedantic` : empêche un programme non compatible avec la norme ANSI C de compiler ;
- `-Wall` : active tous les messages d'avertissement.

Plan

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;
3. une combinaison d'expressions et d'opérateurs ;

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;
3. une combinaison d'expressions et d'opérateurs ;
4. un appel de fonction qui renvoie une valeur, c.-à-d., de type de retour autre que `void`.

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;
3. une combinaison d'expressions et d'opérateurs ;
4. un appel de fonction qui renvoie une valeur, c.-à-d., de type de retour autre que `void`.

Une expression n'est donc rien d'autre qu'un **assemblage de symboles** qui vérifie des règles **syntaxiques** et **sémantiques**.

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;
3. une combinaison d'expressions et d'opérateurs ;
4. un appel de fonction qui renvoie une valeur, c.-à-d., de type de retour autre que `void`.

Une expression n'est donc rien d'autre qu'un **assemblage de symboles** qui vérifie des règles **syntaxiques** et **sémantiques**.

Toute expression possède une valeur et un type. Le processus qui consiste à déterminer la valeur d'une expression se nomme l'**évaluation**.

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

- `b = 3`

  Valeur : `3`, type : `int`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

- `b = 3`

  Valeur : `3`, type : `int`

- `f(5)`

  Valeur : la valeur renvoyée par `f(5)`, type : le type de retour de `f`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

- `b = 3`

  Valeur : `3`, type : `int`

- `f(5)`

  Valeur : la valeur renvoyée par `f(5)`, type : le type de retour de `f`

- `(a == 0) && (b >= 3)`

  Valeur : `0` ou `1`, type : `int`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

- `b = 3`

  Valeur : `3`, type : `int`

- `f(5)`

  Valeur : la valeur renvoyée par `f(5)`, type : le type de retour de `f`

- `(a == 0) && (b >= 3)`

  Valeur : `0` ou `1`, type : `int`

- `a > 'a' ? a : -a`

  Valeur : `a` ou `-a`, type : le type de `a`

Expressions

P.ex., les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : `int`

- `'a'`

  Valeur : `97`, type : `int`

- `"abc"`

  Valeur : `"abc"`, type : `char *`

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : `int`

- `b = 3`

  Valeur : `3`, type : `int`

- `f(5)`

  Valeur : la valeur renvoyée par `f(5)`, type : le type de retour de `f`

- `(a == 0) && (b >= 3)`

  Valeur : `0` ou `1`, type : `int`

- `a > 'a' ? a : -a`

  Valeur : `a` ou `-a`, type : le type de `a`

- `1. + 7 * 2`

  Valeur : `15.`, type : `float`

Instructions

Une instruction est définie récursivement comme étant soit

1. une expression `E;` terminée par un point-virgule ;
2. un bloc `{ I }` où `I` est une instruction ;
3. une conditionnelle `if (E) I1 else I2`, où `E` est une expression et `I1` et `I2` sont des instructions ;
4. toute autre construction similaire (`switch`, `while`, `do while`, `for`).

Instructions

Une instruction est définie récursivement comme étant soit

1. une expression `E;` terminée par un point-virgule ;
2. un bloc `{ I }` où `I` est une instruction ;
3. une conditionnelle `if (E) I1 else I2`, où `E` est une expression et `I1` et `I2` sont des instructions ;
4. toute autre construction similaire (`switch`, `while`, `do while`, `for`).

À la différence des expressions, c'est souvent l'**effet** d'une instruction qui est sa raison d'être (et non plus uniquement sa valeur ou son type).

Instructions

Une instruction est définie récursivement comme étant soit

1. une expression `E;` terminée par un point-virgule ;
2. un bloc `{ I }` où `I` est une instruction ;
3. une conditionnelle `if (E) I1 else I2`, où `E` est une expression et `I1` et `I2` sont des instructions ;
4. toute autre construction similaire (`switch`, `while`, `do while`, `for`).

À la différence des expressions, c'est souvent l'**effet** d'une instruction qui est sa raison d'être (et non plus uniquement sa valeur ou son type).

En effet, une instruction peut p.ex. afficher un élément, allouer une zone mémoire, lire dans un fichier, *etc.* La valeur de l'expression sous-jacente à l'instruction est d'importance secondaire.

Instructions

P.ex., les entités suivantes sont des instructions :

- `;` (instruction vide)
- `1;`
- `a = b;`
- `while (1) a += 1;`
- `printf("abc\n");`
- `a++;`
- `malloc(64);`
- `int tab[64];`
- `tab[3] = 9;`
- `tab[3];`

Expressions à effet de bord

Une **expression** est à effet de bord (ou encore à effet secondaire) si son évaluation modifie la mémoire.

Expressions à effet de bord

Une **expression** est à effet de bord (ou encore à effet secondaire) si son évaluation modifie la mémoire.

P.ex.,

- `0`,
- `'c'`,
- `(a + 21) << 3`,
- `tab[8]`,

ne sont pas a effet de bord.

Expressions à effet de bord

Une **expression** est à effet de bord (ou encore à effet secondaire) si son évaluation modifie la mémoire.

P.ex.,

- `0`,
- `'c'`,
- `(a + 21) << 3`,
- `tab[8]`,

ne sont pas a effet de bord.

En revanche,

- `a = 31`,
- `printf("abc\n")`,
- `char c`,
- `malloc(64)`,

sont à effet de bord.

Expressions à effet de bord

En règle générale, ce sont les éléments suivants dans les expressions qui produisent des effets de bord :

- les affectations ;
- les allocations de mémoire ;
- la sollicitation au système de fichiers.

Expressions à effet de bord

En règle générale, ce sont les éléments suivants dans les expressions qui produisent des effets de bord :

- les affectations ;
- les allocations de mémoire ;
- la sollicitation au système de fichiers.

En revanche, les éléments suivants dans les expressions ne produisent pas d'effet de bord :

- lectures de constantes et de variables ;
- calculs arithmétiques, logiques ou bit à bit ;
- comparaisons.

Plan

Blocs

Un bloc est une suite d'instructions délimitée par des accolades. Il est constitué d'une partie consacrée à la déclaration de variables et d'une partie consacrée aux instructions.

Blocs

Un bloc est une suite d'instructions délimitée par des accolades. Il est constitué d'une partie consacrée à la déclaration de variables et d'une partie consacrée aux instructions.

```
{
  DECL
  INST
}
```

`DECL` : section de **déclarations**.

`INST` : section d'**instructions**.

Blocs

Un bloc est une suite d'instructions délimitée par des accolades. Il est constitué d'une partie consacrée à la déclaration de variables et d'une partie consacrée aux instructions.

```
{
  DECL
  INST
}
```

`DECL` : section de **déclarations**.

`INST` : section d'**instructions**.

Les parties `DECL` ou `INST` peuvent être vides.

Blocs

Un bloc est une suite d'instructions délimitée par des accolades. Il est constitué d'une partie consacrée à la déclaration de variables et d'une partie consacrée aux instructions.

```
{
  DECL
  INST
}
```

`DECL` : section de **déclarations**.

`INST` : section d'**instructions**.

Les parties `DECL` ou `INST` peuvent être vides.

Sachant qu'un bloc est une suite d'instructions, il est possible de placer un bloc dans la section d'instructions d'un bloc et ainsi d'**imbriquer** plusieurs blocs.

Blocs

```
int main() {
    int b;
    scanf("%d", &b);
    {
        int a;
        a = 1 + b;
    }
}
```

La partie rouge est un bloc au sein d'une fonction `main`.

Il y a des règles concernant la visibilité des variables (que nous verrons plus loin).

Blocs

```
int main() {
    int b;
    scanf("%d", &b);
    {
        int a;
        a = 1 + b;
    }
}
```

La partie rouge est un bloc au sein d'une fonction `main`.

Il y a des règles concernant la visibilité des variables (que nous verrons plus loin).

```
{
    int a;
    int b;
    scanf(" %d", &a);
    {
        printf("%d\n", a);
    }
    scanf(" %d", &b);
}
```

Ceci est un bloc constitué d'une section de déclarations et d'une section d'instructions.

Cette dernière partie contient un bloc (en vert) dont la section de déclarations est vide.

Opérateur de test `if`

L'opérateur de test se décline en deux versions :

```
if (EXP)
  BLOC

if (EXP)
  BLOC_1
else
  BLOC_2
```

Ici, `EXP` est une **expression booléenne** : elle est considérée comme fausse si elle s'évalue en zéro et comme vraie sinon. De plus, `BLOC`, `BLOC_1` et `BLOC_2` sont des blocs d'instructions.

Opérateur de test `if`

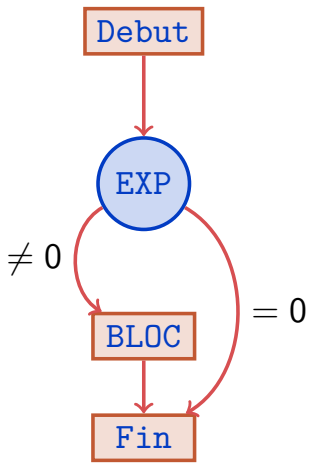
L'opérateur de test se décline en deux versions :

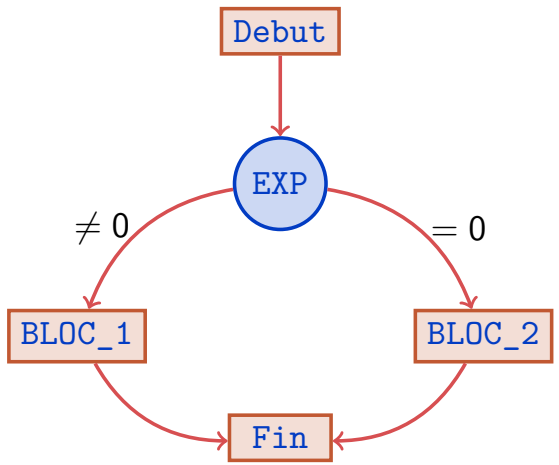
```
if (EXP)
  BLOC

if (EXP)
  BLOC_1
else
  BLOC_2
```

Ici, `EXP` est une **expression booléenne** : elle est considérée comme fausse si elle s'évalue en zéro et comme vraie sinon. De plus, `BLOC`, `BLOC_1` et `BLOC_2` sont des blocs d'instructions.

Diagrammes d'exécution :





Instruction de branchement `switch`

L'instruction de branchement admet la syntaxe

```
switch (EXP) {
  case EXP_1 : INSTR_1
  case EXP_2 : INSTR_2
  ...
  case EXP_N : INSTR_N
  default : INSTR_D
}
```

Ici, `EXP`, `EXP_1`, `EXP_2`, ..., `EXP_N` sont des expressions qui s'évaluent en des entiers.

Les `INSTR_1`, `INSTR_2`, ..., `INSTR_N` et `INSTR_D` sont des suites d'instructions.

Instruction de branchement `switch`

L'instruction de branchement admet la syntaxe

```
switch (EXP) {
  case EXP_1 : INSTR_1
  case EXP_2 : INSTR_2
  ...
  case EXP_N : INSTR_N
  default : INSTR_D
}
```

Ici, `EXP`, `EXP_1`, `EXP_2`, ..., `EXP_N` sont des expressions qui s'évaluent en des entiers.

Les `INSTR_1`, `INSTR_2`, ..., `INSTR_N` et `INSTR_D` sont des suites d'instructions.

L'exécution de cette instruction se passe ainsi. Soit `i` le plus petit entier (s'il existe) tel que les évaluations de `EXP` et `EXP_i` sont égales. Les instructions `INSTR_i`, ..., `INSTR_N` ainsi que `INSTR_D` sont exécutées.

Instruction de branchement `switch`

L'instruction de branchement admet la syntaxe

```
switch (EXP) {
  case EXP_1 : INSTR_1
  case EXP_2 : INSTR_2
  ...
  case EXP_N : INSTR_N
  default : INSTR_D
}
```

Ici, `EXP`, `EXP_1`, `EXP_2`, ..., `EXP_N` sont des expressions qui s'évaluent en des entiers.

Les `INSTR_1`, `INSTR_2`, ..., `INSTR_N` et `INSTR_D` sont des suites d'instructions.

L'exécution de cette instruction se passe ainsi. Soit `i` le plus petit entier (s'il existe) tel que les évaluations de `EXP` et `EXP_i` sont égales. Les instructions `INSTR_i`, ..., `INSTR_N` ainsi que `INSTR_D` sont exécutées.

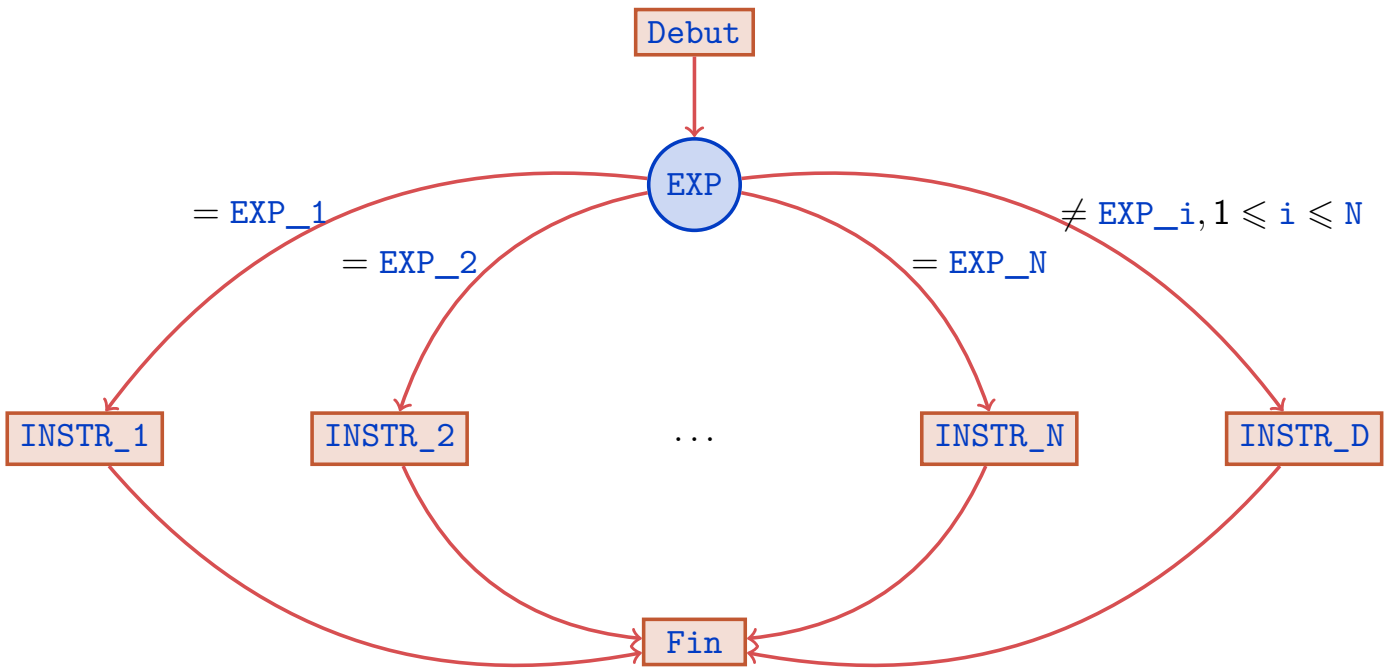
La ligne `default : INSTR_D` est facultative. Si elle est présente, `INSTR_D` est toujours exécutée.

Instruction de branchement `switch`

On s'impose de terminer chaque `INSTR_j` et `INSTR_D` par le mot-clé `break`.

Ceci permet de n'exécuter que le `INSTR_i` tel que les évaluations de `EXP` et `EXP_i` sont égales. Dans ce cas, `INSTR_D` n'est exécuté que si aucun des `EXP_i` ne l'a été.

On obtient ainsi le diagramme d'exécution



Instruction itérative `while`

L'instruction itérative `while` admet la syntaxe

```
while (EXP)
  BLOC
```

Ici, `EXP` est une expression booléenne et `BLOC` est un bloc d'instructions.

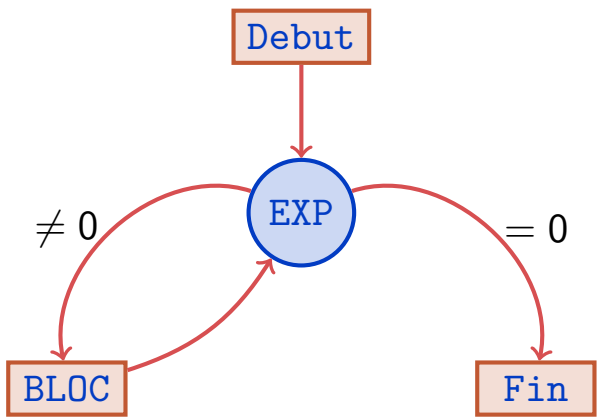
Instruction itérative `while`

L'instruction itérative `while` admet la syntaxe

```
while (EXP)
  BLOC
```

Ici, `EXP` est une expression booléenne et `BLOC` est un bloc d'instructions.

Diagramme d'exécution :



Instruction itérative `do while`

L'instruction itérative `do while` admet la syntaxe

```
do
  BLOC
while (EXP);
```

Ici, `EXP` est une expression booléenne et `BLOC` est un bloc d'instructions.

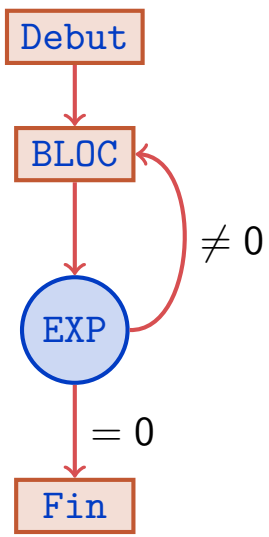
Instruction itérative `do while`

L'instruction itérative `do while` admet la syntaxe

```
do
  BLOC
while (EXP);
```

Ici, `EXP` est une expression booléenne et `BLOC` est un bloc d'instructions.

Diagramme d'exécution :



Instruction itérative `for`

L'instruction itérative `for` admet la syntaxe

```
for (EXP_1 ; EXP_2 ; EXP_3)
  BLOC
```

Ici, `EXP_2` est une expression booléenne (le **test**), `EXP_1` (l'**initialisation**) et `EXP_3` (l'**incrémentation**) sont des expressions et `BLOC` est un bloc d'instructions.

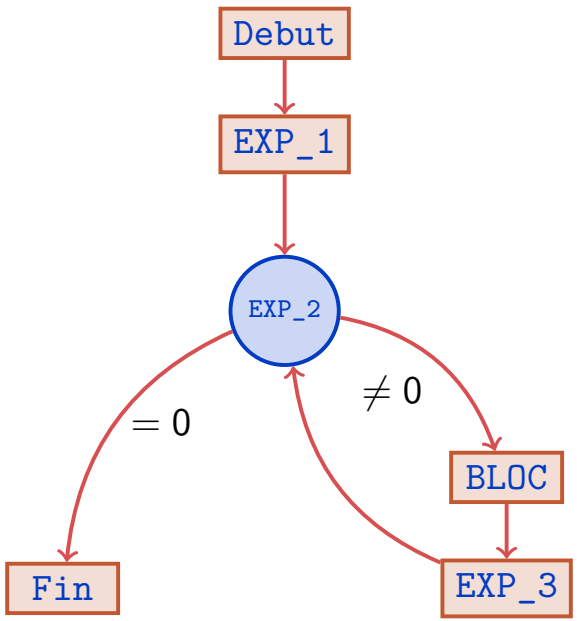
Instruction itérative `for`

L'instruction itérative `for` admet la syntaxe

```
for (EXP_1 ; EXP_2 ; EXP_3)
  BLOC
```

Ici, `EXP_2` est une expression booléenne (le **test**), `EXP_1` (l'**initialisation**) et `EXP_3` (l'**incrémentation**) sont des expressions et `BLOC` est un bloc d'instructions.

Diagramme d'exécution :



Instruction itérative `for`

```
for (i = 0; i < 3; ++i) {
    printf("a");
}
```

Affiche trois occurrences du caractère `'a'`.

Instruction itérative `for`

```
for (i = 0; i < 3; ++i) {
    printf("a");
}
```

Affiche trois occurrences du caractère `'a'`.

```
for (x = lst; x != NULL;
        x = x->suiv) {
    afficher(x->elem);
}
```

Parcours une liste simplement chaînée et l'affiche. La variable `x` est un pointeur sur la cellule courante.

Instruction itérative `for`

```
for (i = 0; i < 3; ++i) {
    printf("a");
}
```

Affiche trois occurrences du caractère `'a'`.

```
for (x = lst; x != NULL;
        x = x->suiv) {
    afficher(x->elem);
}
```

Parcours une liste simplement chaînée et l'affiche. La variable `x` est un pointeur sur la cellule courante.

```
for (; *str != '\0'; str++)) {
    putchar(*str);
}
```

Affiche la chaîne de caractères `str`. Il n'y a pas d'initialisation. L'incrémentation fait pointer `str` sur le prochain caractère.

Instruction itérative `for`

```
for (i = 0; i < 3; ++i) {
    printf("a");
}
```

Affiche trois occurrences du caractère `'a'`.

```
for (x = lst; x != NULL;
        x = x->suiv) {
    afficher(x->elem);
}
```

Parcours une liste simplement chaînée et l'affiche. La variable `x` est un pointeur sur la cellule courante.

```
for (; *str != '\0'; str++)) {
    putchar(*str);
}
```

Affiche la chaîne de caractères `str`. Il n'y a pas d'initialisation. L'incrémentation fait pointer `str` sur le prochain caractère.

```
for (; *str != '\0';
        putchar(*str++)) {
    /* Rien. */
}
```

Affiche la chaîne de caractères `str`. Il n'y a pas d'initialisation. L'affichage est réalisé comme effet de bord de l'incrémentation.

Instructions de court-circuit

Les instructions de court-circuit sont `break` et `continue`.

Instructions de court-circuit

Les instructions de court-circuit sont `break` et `continue`.

L'instruction `break` permet à l'exécution de sortir d'une structure `switch`, `while`, `do while` ou `for`. L'exécution continue alors aux instructions qui suivent cette structure.

Instructions de court-circuit

Les instructions de court-circuit sont `break` et `continue`.

L'instruction `break` permet à l'exécution de sortir d'une structure `switch`, `while`, `do while` ou `for`. L'exécution continue alors aux instructions qui suivent cette structure.

L'instruction `continue` permet à l'exécution de sauter à la fin du bloc `BLOC` d'une structure `while`, `do while` ou `for`. L'exécution continue alors à l'évaluation de l'expression test.

Instructions de court-circuit

Dans une boucle `for`, l'instruction `continue` fait que l'expression d'incrémentation est tout de même évaluée.

Instructions de court-circuit

Dans une boucle `for`, l'instruction `continue` fait que l'expression d'incrémentation est tout de même évaluée.

```
for (i = 1; i <= 7; ++i) {
    if (i == 4)
        continue;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit sept tours de boucle et l'affichage `1 2 3 5 6 7`. Lorsque `i == 4`, le `continue` relance l'exécution à l'incrémentation du `for`.

Instructions de court-circuit

Dans une boucle `for`, l'instruction `continue` fait que l'expression d'incrémentation est tout de même évaluée.

```
for (i = 1; i <= 7; ++i) {
    if (i == 4)
        continue;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit sept tours de boucle et l'affichage `1 2 3 5 6 7`. Lorsque `i == 4`, le `continue` relance l'exécution à l'incrémentation du `for`.

Ce n'est pas le cas pour le `break`.

Instructions de court-circuit

Dans une boucle `for`, l'instruction `continue` fait que l'expression d'incrémentation est tout de même évaluée.

```
for (i = 1; i <= 7; ++i) {
    if (i == 4)
        continue;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit sept tours de boucle et l'affichage `1 2 3 5 6 7`. Lorsque `i == 4`, le `continue` relance l'exécution à l'incrémentation du `for`.

Ce n'est pas le cas pour le `break`.

```
for (i = 1; i <= 7; ++i) {
    if (i == 4)
        break;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit quatre tours de boucle et l'affichage `1 2 3`. Lorsque `i == 4`, le `break` fait sortir de la boucle et `i` vaut `4`.

Plan

Variables

Une variable est une entité constituée des cinq éléments suivants :

Variables

Une variable est une entité constituée des cinq éléments suivants :

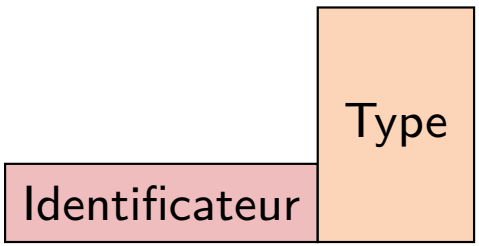
1. un identificateur ;

Identificateur

Variables

Une variable est une entité constituée des cinq éléments suivants :

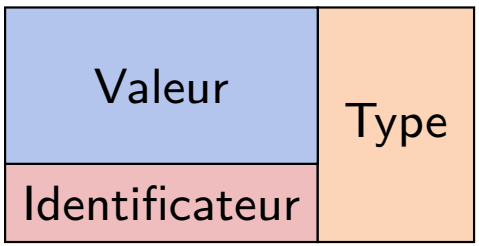
1. un identificateur ;
2. un type ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

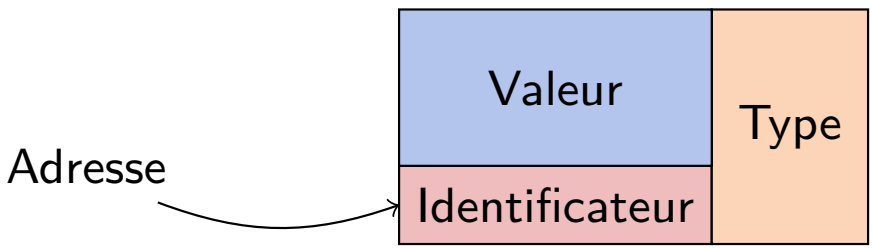
1. un identificateur ;
2. un type ;
3. une valeur ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

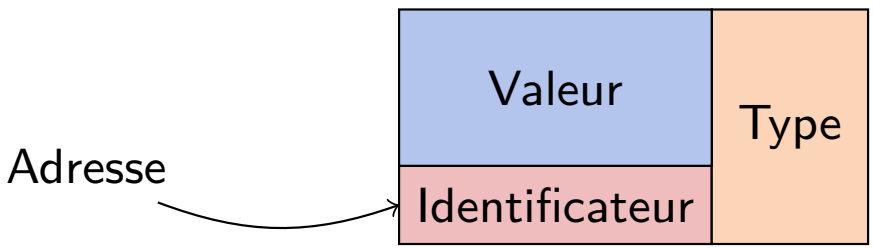
1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;



Variables

Une variable est une entité constituée des cinq éléments suivants :

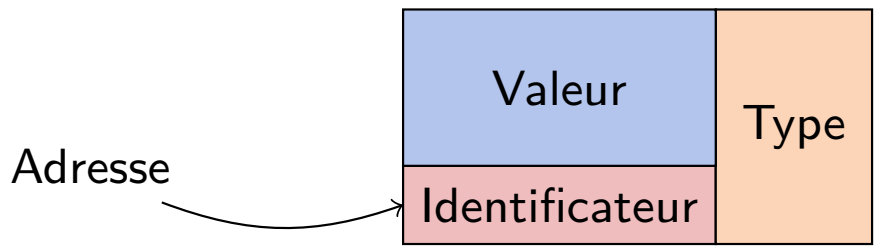
1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;
5. une portée lexicale.



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 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 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 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 : toute *L*-value est une *R*-value, 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.