Algorithmique et Structures de données
Langage C

Dr Abdelkamel, Ben Ali

Université Echahid Hamma Lakhdar - El Oued

Décembre 2020

Licence 2 d'informatique

Langage C — Instructions

Syntaxe

instruction

- *instruction-bloc*
- *instruction-expression*
- *instruction-goto*
- *instruction-if*
- *instruction-while*
- *instruction-do*
- *instruction-for*
- *instruction-break*
- *instruction-continue*
- *instruction-switch*
- *instruction-return*
- *instruction-vide*
- *identificateur : instruction*

Syntaxe (2)

- *instruction-bloc*
  - { *déclaration ... déclaration*
    instruction ... instruction }
- *instruction-expression*
  - *expression*
- *instruction-goto*
  - `goto` identif ;
- *instruction-if*
  - `if` (*expression*) *instruction* `else` *instruction*
  - `if` (*expression*) *instruction*

Syntaxe (3)

- *instruction-while*
  - `while` (*expression*) *instruction*
- *instruction-do*
  - `do` *instruction* `while` (*expression*) ;
- *instruction-for*
  - `for` ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *instruction*
- *break*
  - `break` ;
- *continue*
  - `continue` ;

Syntaxe (4)

- *instruction-switch*
  - `switch` (*expression*) {*instruction-ou-case ... instruction-ou-case*}
- *instruction-ou-case*
  - `case` *expression-constante : instruction*
  - `default` *: instruction*
  - *instruction*
- *return*
  - `return` $expresion_{opt}$
- *instruction-vide*
  - ;
- C et le point virgule : en C le point-virgule n'est pas un séparateur d'instructions mais un terminateur de certaines instructions. (il appartient à la syntaxe de certaines instructions.)

Syntaxe (5)

- *instruction-while*
  - `while` (*expression*) *instruction*
- *instruction-do*
  - `do` *instruction* `while` (*expression*) ;
- *instruction-for*
  - `for` ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *instruction*
- *break*
  - `break` ;
- *continue*
  - `continue` ;

Présentation détaillée des instructions (1)

Instruction-bolc

Présentation détaillée des instructions (2)

Instruction-expression

Syntaxe :

expression ;

- Écrire un point-virgule derrière n'importe quelle expression pour faire en une instruction ;
- Exemple :

```
123;
i++;
x = 2 * x + 3;
printf("%d\n", n);
sacnf("%d", &x);
z = Sum(a, b);
Trier(&Tab);
```

Présentation détaillée des instructions (3)

Etiquettes et instruction `goto`

Syntaxe :

```
goto identificateur;
```

- L'utilisation de l'instruction `goto` transférer le contrôle d'exécution à l'instruction étiquetée par `identificateur`.
- Les étiquettes sont toujours terminiées par deux points :.
- L'instruction `goto` et l'instruction cible du `goto` doivent se trouver dans la même procédure (branchements locaux).
- Exemple :

```
Encore:        /* Ceci est l'étiquette */

;
...

goto Encore;
```

Présentation détaillée des instructions (4)

Instruction `if ... else`

Syntaxe :

```
if (expression)
    instruction1
else
    instruction2
```

et

```
if (expression)
    instruction1
```

- L'expression conditionnelle doit se figurer entre parenthèses.
- Chaque `else` se rapporte au dernier `if`.

Présentation détaillée des instructions (5)

Instructions `while` et `do ... while`

Syntaxe :

```
while (expression)
    instruction
```

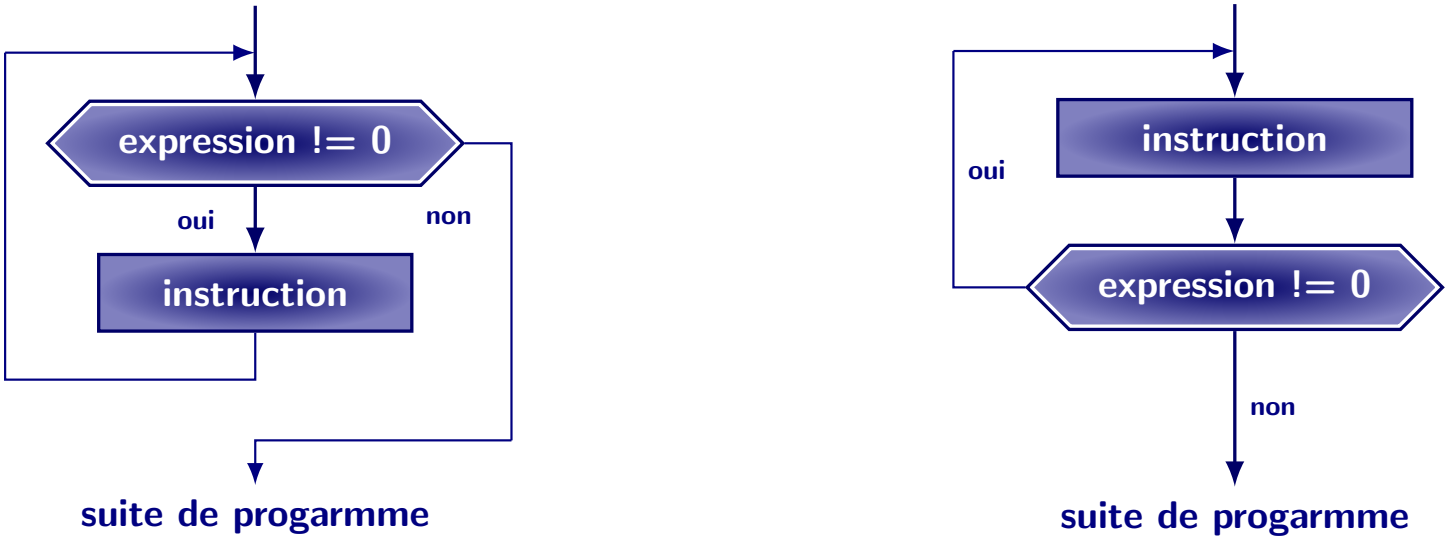
et

```
do
    instruction
while (expression) ;
```

- La condition doit se figurer entre parenthèses.

Présentation détaillée des instructions (5)

Le fonctionnement de `while` et `do ... while`



Instruction while (à gauche) et do...while (à droite)

Présentation détaillée des instructions (6)

Instructions `for`

Syntaxe :

```
for (exp1 ; exp2 ; exp3)
    instruction
```

équivaut strictement à celle-ci :

```
exp1 ;
while (expr2)
{
    instruction
    expr3 ;
}
```

Présentation détaillée des instructions (6)

Instruction `for` (suite)

- Les expressions *expr1* et *expr3* peuvent être absentes (les points-virgules doivent cependant apparaitre).

  `for` (; *exp2* ;)
  instruction

  équivaut à :

  `while` (*exp2*)
  instruction

- la boucle indéfinie peut se programmer :

  `for` (; ;)
  instruction

Présentation détaillée des instructions (7)

Instruction `switch`

Syntaxe :

`switch` (*expression*)
corps

- Le corps de l'instruction `switch` prend la forme d'un bloc ... renfermant une suite d'instructions entre lesquelles se trouvent des constructions de la forme
  `case` expression-constante :
  ou bien
  `default` :
- Lorsqu'il y a branchement réussi à un énonce case, toutes les instructions qui le suivent sont exécutées, jusqu'à la fin du bloc. Pour obtenir un comportement similaire à celui du `case...of...` de Pascal, on doit utiliser l'instruction `break`,

Présentation détaillée des instructions (7)

Instruction `switch` – Exemples

```
j = 0;
switch (i)
{
     case 3:
          j++;
     case 2:
          j++;
     case 1:
          j++;
}
```

```
j = 0;
switch (i)
{
     case 3:
          j++;
          break;
     case 2:
          j++;
          break;
     case 1:
          j++;
          break;
     default : ;
}
```

Question. Valeur de j après `suitch` ?

Présentation détaillée des instructions (8)

Instructions `break` et `continue`

Syntaxe :

```
break;
continue;
```

- L'instruction `break` provoque l'abandon de la structure de contrôle (instructions `for`, `while`, `do` et `switch`) et le passage à l'instruction écrite immédiatement derrière.
- Dans la portée d'une structure de contrôle de type boucle (`while`, `do` ou `for`), l'instruction `continue` produit l'abandon de l'itération courante et le démarrage de l'itération suivante.
- Lorsque plusieurs boucles sont imbriquées, c'est la plus profonde qui est concernée par les instructions `break` et `continue`.

Présentation détaillée des instructions (8)

Instructions `break` et `continue` ...

Par exemple la construction

```
for (expr1 ; expr2 ; expr3)
{
    ...
    break;
    ...
}
```

équivaut à

```
{
    for (expr1 ; expr2 ; expr3)
    {
        ...
        goto sortie;
        ...
    }
    sortie: ;
}
```

Présentation détaillée des instructions (9)

Instruction `return`

Syntaxe :

`retrun` *expression* ;

et

`retrun` ;

- `return` provoque l'abandon de la fonction en cours et le retour à la fonction appelante.
- Dans la première forme expression est évaluée ; son résultat est la valeur que la fonction renvoie à la fonction appelante.
- Dans la deuxième forme, la valeur retournée par la fonction reste indéterminée.
- Absence d'instruction `return` dans une fonction. Lorsque la dernière instruction d'une fonction est terminée, le contrôle est rendu également à la procédure appelante.

Langage C — Syntaxe, Déclaration ...

Structure générale d'un programme

- La transformation d'un texte écrit en langage C en un programme exécutable par l'ordinateur se fait en deux étapes : la *compilation* et l'*édition de liens*.
  - *Compilation* : Traduction des fonctions écrites en C en des procédures équivalentes écrites dans un langage machine.
  - Le compilateur lit toujours un fichier, appelé fichier source, et produit un fichier, dit fichier objet.
  - Chaque fichier objet est incomplet, insuffisant pour être exécuté, car il contient des appels de fonctions ou des références à des variables qui ne sont pas définies dans le même fichier (Exemple, la fonction `printf`)
  - L'*éditeur de liens* (ou *linker*) prend en entrée plusieurs fichiers objets et bibliothèques (une variété particulière de fichiers objets) et produit un unique fichier exécutable.
  - L'éditeur de liens est largement indépendant du langage de programmation.

Structure générale d'un programme (2)

La version C du célèbre programme –qui dit– bonjour

```c
#include <stdio.h>

int main()
{
    printf("Bonjour\n");
    return 0;
}
```

```c
#include <stdio.h>

void main()
{
    printf("Bonjour\n");
}
```

Considérations lexicales (1)

Présentation du **texte du programme**

- Des blancs, des tabulations et des sauts à la ligne peuvent être places à tout endroit
  - où cela ne coupe pas un identificateur, un nombre ou un symbole composé.
- Les commentaires commencent par `/*` et se terminent par `*/` :
  - `/* Ce texte est un commentaire et sera donc ignoré par le compilateur */`
    - Les commentaires ne peuvent pas être imbriques
      le texte `/* voici un grand /* et un petit */ commentaire */` est erroné, car seul `/* voici un grand /* et un petit */` sera vu comme un commentaire par le compilateur
- Le caractère anti-slash \ précédant immédiatement un saut à la ligne masque ce dernier
  - Par exemple, le texte

```
message = "anti\
                constitutionnellement";
```

    est compris comme ceci :

```
message = "anti constitutionnellement";
```

Considérations lexicales (2)

Mots clés

Les mots suivants sont réservés. Leur fonction est prévue par la syntaxe de C et ils ne peuvent pas être utilises dans un autre but :

`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`
`volative`	`while`				

Considérations lexicales (3)

Identificateurs

- Un identificateur est une suite de lettres et chiffres contigus, dont le premier est une lettre.
- en toute circonstance une lettre majuscule est tenue pour différente de la lettre minuscule correspondante.
- dans les identificateurs, le nombre de caractères discriminants est au moins de 31.
- Le caractère _ (appelé "blanc souligné") est considéré comme une lettre

Considérations lexicales (4)

Opérateurs

Symboles simples

```
(   )   [   ]   .   !   ~   <   >   ?   :
=   ,   +   -   *   /   %   |   &   ^
```

Symboles composés

```
->   ++   --   <=   >=   ==   !=   &&   ||   <<   >>
+=   -=   *=   /=   %=   <<=   >>=   |=   &=   ^=
```

- Tous ces symboles sont reconnus par le compilateur comme des operateurs.
- Il est interdit d'insérer des caractères blancs à l'intérieur d'un symbole composé.
- il est conseillé d'encadrer par des blancs toute utilisation d'un opérateur

Constantes littérales

Nombres entiers

- Les constantes littérales numériques entières ou réelles suivent les conventions habituelles, avec quelques particularités.
- Sans signe : -123, le calcul est fait pendant la compilation en appliquant l'opérateur unaire - à la constante 123 .
- Il n'existe pas d'operateur + unaire en C original : la notation +123 est interdite.
- On peut écrire les constantes entières en octal et en hexadécimal :
  - en octal (base 8) : on commence l'écriture par 0 (zéro) ;
  - en hexadécimal (base 16) : on commence l'écriture par 0x ou 0X.
    Exemple : trois manières d'écrire le même nombre :

    27 033 0x1B
- Détail à retenir : on ne doit pas écrire de zéro non significatif : 0123 ne représente pas la même valeur que 123.

Constantes littérales (2)

Nombres entiers

- Le type d'une constante entière est le plus petit type dans lequel sa valeur peut être représentée. Ou, plus exactement :
  - si elle est décimale : si possible `int`, sinon `long int`, sinon `unsigned long int` ;
  - si elle est octale ou hexadécimale : si possible `int`, sinon `unsigned int`, sinon `long int`, sinon `unsigned long int`.
- Certains suffixes permettent de changer cette classification :
  - U, u : indique que la constante est d'un type unsigned ;
  - L, l : indique que la constante est d'un type long.
    Exemples : `1L`, `0x7FFFU`.
    On peut combiner ces deux suffixes : `16UL`.

Constantes littérales (3)

Nombres flottants

- Une constante littérale est l'expression d'un nombre flottant si elle présente, dans l'ordre :
  - la partie entière : une suite de chiffres décimaux,
  - la virgule décimale : un point,
  - la partie fractionnaire : une suite de chiffres décimaux,
  - L'exposant
    - une des deux lettres E ou e, éventuellement un signe + ou -,
    - une suite de chiffres décimaux.

  Exemples : `123.456E-78`
- On peut omettre :
  - la partie entière ou la partie fractionnaire, mais pas les deux,
  - le point ou l'exposant, mais pas les deux.
    Exemples : `.5e7`, `5.e6`, `5000000.`, `5e6`

Constantes littérales (4)

Nombres flottants

- Une constante flottante est supposée de type `double`, à moins de comporter un suffixe explicite :
  - les suffixes F ou f indiquent qu'elle est du type `float` ;
  - les suffixes L ou l indiquent qu'elle est du type `long double`.

  Exemples : `1.0L`, `5.0e4f`

Constantes littérales (5)

Caractères et chaînes de caractères

- Une constante de type caractère

  `'A'  '2'  '"'`

- Une constante de type chaînes de caractères

  `"A"  "Bonjour à tous !"  ""  "'"`

- Séquences d'échappement :
  - `\n` nouvelle ligne (LF)
  - `\t` tabulation (HT)
  - `\b` espace-arrière (BS)
  - `\r` retour-chariot (CR)
  - `\f` saut de page (FF)
  - `\a` signal sonore (BEL)
  - `\\` \
  - `\'` '
  - `\"` "
  - `\d3d2d1` le caractère qui a pour code le nombre octal `d3d2d1`. On peut aussi le noter `\d2d1` ou `\d1`, s'il commence par un ou deux zéros.

Constantes littérales (6)

Caractères et chaînes de caractères

- Par exemple, la chaîne suivante définit la suite des 9 caractères A, espace (de code ASCII 27), B, ", C, saut de page, D, \et E :

```
"A\033B\"C\f D\\E"
```

- Une constante de type caractère appartient au type char, c'est-a-dire entier représenté sur un octet. [ASCII, UNICODE]
- Une constante de type chaîne de caractères représente une suite finie de caractères, de longueur quelconque.
  - les caractères de la chaîne sont rangés en mémoire, de manière contigüe, dans l'ordre ou ils figurent dans la chaîne ;
  - un caractère nul est ajouté immédiatement après le dernier caractère de la chaîne, pour en indiquer la fin ;
  - la constante chaîne représente alors, à l'endroit où elle est écrite, l'adresse de la cellule où a été rangé le premier caractère de la chaîne.

`"bonjour"`

		b	o	n	j	o	u	r	\0	

Constantes littérales (7)

Expressions constantes

Une expression constante est une :

- toute constante littérale ; exemples : 1, `'A'`, `"HELLO"`, `1.5e-2` ;
- une expression correcte formée par l'application d'un operateur courant (arithmétique, logique, etc.) à une ou deux expressions constantes ; exemples : `-1`, `'A' - 'a'`, `2 * 3.14159265`, `"HELLO" + 6` ;
- l'expression constituée par l'application de l'operateur `&` (operateur de calcul de l'adresse, voyez le cours ..) à une variable statique, à un champ d'une variable statique de type structure ou à un élément d'un tableau statique dont le rang est donné par une expression constante ; exemples : `&x`, `&fiche.nom`, `&table[50]` ;
- l'expression constituée par l'application de l'operateur `sizeof` à un descripteur de type. Exemples : `sizeof(int)`, `sizeof(char *)` ;
- l'expression constituée par l'application de l'operateur `sizeof` à une expression quelconque, qui ne sera pas évaluée ; exemples : `sizeof x`, `sizeof(2 * x + 3)`.

- En C, il est garanti que toute expression constante (et donc toute sous-expression constante d'une expression quelconque) sera effectivement évaluée avant que l'exécution ne commence (évaluation gratuite).

Constantes littérales (7)

Expressions constantes

Une expression constante est une :

- toute constante littérale ; exemples : 1, `'A'`, `"HELLO"`, `1.5e-2` ;
- une expression correcte formée par l'application d'un operateur courant (arithmétique, logique, etc.) à une ou deux expressions constantes ; exemples : `-1`, `'A' - 'a'`, `2 * 3.14159265`, `"HELLO" + 6` ;
- l'expression constituée par l'application de l'operateur `&` (operateur de calcul de l'adresse, voyez le cours ..) à une variable statique, à un champ d'une variable statique de type structure ou à un élément d'un tableau statique dont le rang est donné par une expression constante ; exemples : `&x`, `&fiche.nom`, `&table[50]` ;
- l'expression constituée par l'application de l'operateur `sizeof` à un descripteur de type. Exemples : `sizeof(int)`, `sizeof(char *)` ;
- l'expression constituée par l'application de l'operateur `sizeof` à une expression quelconque, qui ne sera pas évaluée ; exemples : `sizeof x`, `sizeof(2 * x + 3)`.

- En C, il est garanti que toute expression constante (et donc toute sous-expression constante d'une expression quelconque) sera effectivement évaluée avant que l'exécution ne commence (évaluation gratuite).

Types fondamentaux du langage C

Types de base

Nombres entiers

Anonymes

Petite taille

- signes `char`
- non signés `unsigned char`

Taille moyenne

- signes `short`
- non signés `unsigned short`

Grande taille

- signes `long`
- non signés `unsigned long`

Nommés

Types fondamentaux du langage C

Types de base

Nombres flottants

- Simples `float`
- Grande précision `double`
- Précision encore plus grande `long double`

Types dérivés

- Tableaux [ ]
- Fonctions ( )
- Pointeurs *
- Structures `struct`
- Unions `union`

Les types dérivés obtenus en appliquant quelques procédés récursifs de construction soit à des types fondamentaux soit à des types dérivés définis de la même manière.

Nombres entiers et caractères

- Deux critères de classification :
  - Nombres positifs / négatifs : type signé / non signé
  - Taille requise par leur représentation

    Rappal

    Avec N chiffres binaires (ou bits) on peut représenter :
    - soit les 2^N nombres positifs $0, 1, 2 \dots 2^N - 1$ (cas non signé) ;
    - soit les 2^N nombres positifs et négatifs $-2^{N-1} \dots 2^{N-1} - 1$ (cas non signé) ;

- Le TYPE CARACTÈRE. Un objet de type char peut être défini, au choix, comme :
  - Un nombre entier pouvant représenter toute caractère du jeu de caractères
  - Un nombre entier occupant une cellule mémoire un octet (08 bits)
    - Souvent, char est un entier signé .... un unsigned char est alors un entier non signé. Certains compilateur permettent traitent les deux cas ...

Nombres entiers et caractères (2)

- LES ENTIERS COURTS ET LONGS : `short` $\subset$ `long`

  De nos jours on trouve souvent :

`unsigned short`	16 bits	0 .. 65.535
`short`	16 bits	-32.768 .. 32.767
`unsigned long`	32 bits	0 .. 4.294.967.296
`long`	32 bits	-2.147.483.648 .. 2.147.483.647

- LE TYPE INT : correspond à la taille d'entier la plus efficace (machine utilisée) ; `int` = `short` / `int` = `long`
  - Conseil : n'utiliser le type int que pour les variables locales destinées à contenir des valeurs raisonnablement petites (inferieures en valeur absolue à 32767)

Nombres entiers et caractères (3)

- A PROPOS DES BOOLÉENS.

  En C il n'existe donc pas de type booléen spécifique.

  - Une expression booléenne sera tenue pour vraie si elle est non nulle, elle sera considérée fausse sinon.
    `expr` (c-à-d, `expr` "vraie") équivalent à `expr != 0`
  - Un opérateur booléen (égalité, comparaison, etc.) produit 0 pour faux et 1 pour vrai.
  - À retenir, le fichier `<types.h>` comporte les déclarations :

```
enum {false, true};
typedef unsigned char Boolean;
```

Types énumérés

- **Énumération** : une famille finie de nombres entiers, chacun associé à un identificateur qui en est le nom.

```
enum jour_semaine {lundi, mardi, mercredi, jeudi,
vendredi, samedi, dimanche};
```

lundi valant 0, mardi valant 1, mercredi valant 2, etc. Ainsi, les expressions mardi + 2 et jeudi représentent la même valeur.

Nombres flottants

Trois types de flottants :

`float`	simple précision
`double`	double précision
`long double`	précision étendue

- `float` M/Eb/SM
  - 32 bits $23 + 8 + 1$
  - 1.18 10^-38 $< |X| <$ 3.40 10^38
  - Scientific (7-digit precision)
- `double`
  - 64 bits $52 + 11 + 1$
  - 2.23 10^-308 $< |X| <$ 1.79 10^308
  - Scientific (15-digit precision)
- `long`
  - 80 bits $64 + 15 + 1$
  - 3.37 10^-4932 $< |X| <$ 1.18 10^4932
  - Financial (18-digit precision)

Langage C — Variables

Syntaxe des déclarations

La forme simple de la déclaration d'une variable :

spécification var-init, var-init, ... var-init;

où spécification est de la forme :

$$\left\{\begin{array}{c}\texttt{auto}\\ \texttt{register}\\ \texttt{static}\\ \texttt{extern}\\ \textit{rien}\end{array}\right\}\ \left\{\begin{array}{c}\texttt{const}\\ \texttt{volative}\\ \textit{rien}\end{array}\right\}\ \left\{\begin{array}{c}\left\{\begin{array}{c}\texttt{signed}\\ \texttt{unsigned}\\ \textit{rien}\end{array}\right\}\ \left\{\begin{array}{c}\texttt{char}\\ \texttt{short}\\ \texttt{long}\\ \texttt{int}\end{array}\right\}\\ \texttt{float}\\ \texttt{double}\\ \texttt{long double}\end{array}\right\}$$

et chaque *var-init* est de la forme :

$$identificateur = \left\{\begin{array}{c}\texttt{= expression}\\ \textit{rien}\end{array}\right\}$$

Syntaxe des déclarations (2)

Exemples :

```
int x, y = 0, z;
extern float a, b;
static unsigned short cpt = 1000;
```

Les déclarations de variables peuvent se trouver :

- en dehors de toute fonction, il s'agit alors de **variables globales** ;
- à l'intérieur d'un bloc, il s'agit alors de **variables locales** ;
- dans l'en-tête d'une fonction, il s'agit alors d'arguments formels, placés
  - soit dans les parenthèses de l'en-tête (fonction définie en syntaxe ANSI avec un prototype),
  - soit entre le nom de la fonction et le { initial (fonction définie en syntaxe originale ou sans prototype).

Syntaxe des déclarations (3)

Exemples :

Avec prototype :

```
long i = 1;

int my_function(int j)
{
      short k;
      ...
}
```

Sans prototype :

```
long i = 1;
int my_function(j)
int j;
{
     short k;
     ...
}
```

Ci-dessus, i est une variable globale, k une variable locale et j un argument formel d'une fonction.

Visibilité des variables

Quels sont les identificateurs auxquels on peut faire référence en un point d'un programme ?

- En C :
  - Les fonctions ne peuvent pas être imbriquées les unes dans les autres.
  - Tout bloc peut comporter ses propres définitions de variables locales. Un bloc est une suite de déclarations et d'instructions encadrée par une accolade ouvrante "{" et l'accolade fermante "}" correspondante. Le corps d'une fonction est lui-même un bloc, mais d'autres blocs peuvent être imbriques dans celui-là.

Variables locales

- Une **variable locale** ne peut être référencé que depuis l'intérieur du bloc où elle est définie ;
- Le nom d'une variable locale masque toute variable de même nom définie dans un bloc englobant le bloc en question.
- Toutes les déclarations de variables locales à un bloc doivent être écrites au début du bloc, avant la première instruction.

Visibilité des variables (2)

Arguments formels

- Un argument formel est accessible de l'intérieur de la fonction, partout où une variable locale plus profonde ne le masque pas.
- En aucun cas on ne peut y faire référencé depuis l'extérieur de la fonction.

Variables globales

- Le nom d'une variable globale ou d'une fonction peut être utilisé depuis n'importe quel point compris entre sa déclaration (pour une fonction : la fin de la déclaration de son en-tête) et la fin du fichier où la déclaration figuré, sous réserve de ne pas être masquée par une variable locale ou un argument formel de même nom.

Allocation et durée de vie des variables

- Les variables globales sont toujours **statiques** : elles existent pendant toute la durée de l'exécution. Le système d'exploitation se charge, immédiatement avant l'activation du programme, de les allouer dans un espace mémoire de taille adéquate.
- Les variables locales et les arguments formels des fonctions sont **automatiques** : l'espace correspondant est alloué lors de l'activation de la fonction ou du bloc en question et il est rendu au système lorsque le contrôle quitte cette fonction ou ce bloc.
- **Remarque**. Une grande similitude entre les variables locales et les arguments formels des fonctions : visibilité et durée de vie. En réalité : les arguments formels sont de vraies variables locales avec l'unique particularité d'être automatiquement initialises (par les valeurs des arguments effectifs) lors de l'activation de la fonction.

Initialisation des variables

Variables statiques. la déclaration d'une variable statique peut indiquer une valeur initiale à ranger dans la variable. Cela est vrai y compris pour des variables de types complexes (tableaux ou structures).

Exemple :

```
double x = 0.5e3;
int t[5] = { 11, 22, 33, 44, 55 };
```

- La valeur initiale doit être définie par une expression constante (calculable durant la compilation, avant l'exécution) ;
- Les variables statiques pour lesquelles aucune valeur initiale n'est indiquée sont remplies de zéros. L'interprétation de ces zéros dépend du type de la variable.

Initialisation des variables (2)

Variables automatiques

- Les arguments formels des fonctions sont automatiquement initialisés lors de leur création (au moment de l'appel de la fonction) par les valeurs des arguments effectifs.
- La déclaration d'une variable locale peut elle aussi comporter une initialisation. Mais l'initialisation représente ici une affectation tout à fait ordinaire. Ainsi, la construction

```
int i = exp; /* déclaration + initialisation */
```

équivaut au couple

```
int i; /* déclaration */
...
i = exp ; /* affectation */
```

  - l'expression qui donne la valeur initiale est évaluée à l'exécution, chaque fois que la fonction ou le bloc est activé ;
- Les variables automatiques pour lesquelles aucune valeur initiale n'est indiquée sont allouées avec une valeur imprévisible.
- **Remarque**. Dans le C original, une variable automatique ne peut être initialisée que si elle est simple (c.-à-d.. autre que tableau ou structure). Cette limitation ne fait pas partie du C ANSI.

Variables locales statiques

- Le qualificaeur `static`, placé devant la déclaration d'une *variable locale*, produit une variable qui est
  - pour sa visibilité, locale ;
  - pour sa durée de vie, statique (c.-à-d permanente).
- Elle est créée au début de l'activation du programme et elle existe aussi longtemps que dure l'exécution de celui-ci.
  Exemple :
  Sans prototype :

```
void bizarre1(void)
{
    static int cpt = 1000;
    printf("%d ", cpt);
    cpt++;
}
```

- L'initialisation de telle variable est effectuée une seule fois avant l'activation du programme.
- Une variable locale statique conserve sa valeur entre deux activations consécutives de la fonction.

Variables locales statiques (2)

Comparer des appels consécutifs des fonctions :

```
void bizarre1(void)
{
    static int cpt = 1000;
    printf("%d␣", cpt);
    cpt++;
}

int cpt = 1000;
void bizarre2(void)
{
    printf("%d␣", cpt);
    cpt++;
}

void bizarre3(void)
{
    int cpt = 1000;
    printf("%d␣", cpt);
    cpt++;
}
```

- **Remarque**. On utilise les variables locales statiques (au lieu de globales) pour éviter la modification inconsidérée par une autre fonction, ou entrer en conflit avec un autre objet de même nom.

Variables critiques

- Le qualifieur `register` précédant une déclaration de variable informe le compilateur que la variable en question est très fréquemment accédée pendant l'exécution du programme.
  - dans certains calculateurs de telles variables sont logées (si possible) dans un registre de l'unité centrale de traitement (CPU) plutôt que dans la mémoire centrale.
  - Les variables ainsi déclarées doivent être locales et d'un type simple. Elles sont automatiquement initialisées à zéros chaque fois qu'elles sont créées.