**Chapitre 2. Manipulation de données en C**

# Les variables

## Qu’est-ce qu’une variable ?

Une variable est un emplacement mémoire qui sert à stocker une valeur qui peut changer pendant l'exécution d'un programme. Elle est définie par cinq éléments :

* + - **L'identificateur**: c'est le nom que l'on donne à la variable.
    - **Le type:**il détermine la nature de l’information (nombre entier,nombre réel,caractère, …).
    - **La taille:** c'est le nombre d'octets occupés en mémoire, elle est en fonction du type.
    - **La valeur:** c'est la valeur que l'on attribue à la variable.
    - **L'adresse**: c'est l’emplacement où est stocké la valeur de la variable.

**Adresse**

**FF00**

**Mémoire centrale**

**x Nom de la variable**



**15**

**Valeur**

## Classification des types de données simples

Le tableau suivant présente tous les types de données simples qui nous permettront de définir les variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type de donnée** | **Signification** | **Taille (en octets)** | **Valeurs limites** |
| **char** | caractère | 1 | -128 à 127 |
| **unsigned char** | caractère non signé | 1 | 0 à 255 |
| **short int** | entier court | 2 | -32 768 à +32 767 |
| **unsigned short int** | entier court non signé | 2 | 0 à 65 535 |
| **int** | entier | 2 (sur processeur 16 bits)  4 (sur processeur 32 bits) | -32 768 à +32 767  -2 147 483 648 à +2 147 483 647 |
| **unsigned int** | entier non signé | 2 (sur processeur 16 bits)  4 (sur processeur 32 bits) | 0 à 65 535  0 à 4 294 967 295 |
| **long int** | entier long | 4 | -2 147 483 648 à +2 147 483 647 |
| **unsigned long int** | entier long non signé | 4 | 0 à 4 294 967 295 |
| **float** | flottant (réel) | 4 | 3.4\*10-38 à 3.4\*1038 |
| **double** | flottant double | 8 | 1.7\*10-308 à 1.7\*10308 |
| **long double** | flottant double long | 10 | 3.4\*10-4932 à 3.4\*104932 |

## Identificateur d’une variable

Il existe un certain nombre de limites pour choisir l’identificateur d’une variable (son nom) :

* + - un identificateur peut contenir des lettres minuscules ou majuscules, des chiffres, ou le caractère spécial de soulignement « **\_** ». Par contre, il ne doit pas commencer par un chiffre ou posséder des lettres accentuées.
    - les espaces ne sont pas admis dans l’identificateur ;
    - un identificateur ne doit pas comporter plus de 32 caractères ;
    - les majuscules sont distinguées des minuscules, ainsi : Montant et montant désignent deux noms différents.
    - un identificateur ne peut pas être un mot réservé du langage :

|  |  |  |  |
| --- | --- | --- | --- |
| Auto | double | int | struct |
| Break | else | long | switch |
| Case | enum | register | typedef |
| Char | extern | return | union |
| Const | float | short | unsigned |
| Continue | for | signed | void |
| Default | goto | sizeof | volatile |
| Do | if | static | while |

## Identificateurs valides :

x x1 \_x

## Identificateurs non valides :

**2**eme commence par un chiffre

a**#**b caractère non autorisé (#) num**-**employe caractère non autorisé (-)

racine carree caractère non autorisé (espace)

## Déclaration des variables

Avant d’utiliser une variable dans un programme C, il faut la déclarer c'est-à-dire réserver son emplacement mémoire.

Pour déclarer une variable, on doit :

1. spécifier le type de donnée;
2. indiquer l’identificateur de la variable.

Une variable peut être initialisée lors de sa déclaration.

**Syntaxe** : Type **identificateur** [= valeur\_initiale];

## Exemples :

int x=-10, y, z=20 ; char touche = ‘A' ; float hauteur, largeur;

# Constantes littérales

Quand une valeur comme **1** apparaît dans un programme, elle est appelée constante littérale : littérale car on ne peut parler d'elle qu'à travers sa valeur, et constante car sa valeur ne peut être changée. Chaque littéral possède un type associé. Par exemple, **0** est de type entier, **3.1459** est une constante littérale de type double.

Le tableau suivant présente les quatre types de constantes littérales du langage C : les constantes entières, réelles, caractères et chaîne de caractères.

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Exemples** |
| **Entière** | Décimale | -12, 20 |
| Octale | 015,027 |
| Hexadécimale | 0x1A5, 0xFFFF |
| **Réelle** | Décimale | 15.25, 1.53**e**2, 314**E**-2 |
| **Caractère** | Imprimable | ‘a' ..... ‘2'.....‘,' ‘ ‘ (espace) |
| Non-imprimable | * **\n** : saut de ligne (Line Feed) * **\t** : tabulation horizontale (Horizontal Tab) * **\v** : tabulation verticale (Vertical Tab) * **\b** : retour arrière (Backspace) * **\r** : retour chariot (Carriage Return) * **\f** : saut de page (Form Feed) * **\a** : sonnerie ou bip (Alert) |
| **Chaîne de caractère** |  | **"**Bonjour**"** |

# Constantes symboliques : const

Une constante est une variable dont l'initialisation est obligatoire et dont la valeur ne pourra pas être modifiée en cours d'exécution. Elle est déclarée avec le mot clé : **const** qui doit précéder le type.

**Syntaxe** : **const** type **identificateur** = valeur\_initiale;

**Exemples** : **const** double PI = 22.0 / 7.0;

**const** float TVA = 0.2;

**const** float remise ;

remise = 0.4; **Erreur !**

# Opérateurs et expressions

Une *expression* peut être une variable, une constante, un appel d’une fonction (avec retour de valeur) ou d'une combinaison de chacun de ces éléments par des *opérateurs*. Toute expression à un type et une valeur.

## Exemples :

* b\*b – 4\*a\*c
* 'A' + 1
* (-b + sqrt(b\*b – 4\*a\*c)) / (2\*a)

## Classification et description

Les tableaux suivants présentent la liste de tous les opérateurs disponibles en C.

|  |  |  |
| --- | --- | --- |
| **Opérateurs arithmétiques** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Addition | **+** |  |
| Soustraction | **-** |  |
| Multiplication | **\*** |  |
| Division | **/** |  |
| Modulo | **%** | Reste d’une division entre entiers : **7/5 = 1**, reste **2**; donc **7%5 = 2** |

|  |  |  |
| --- | --- | --- |
| **Opérateurs de relation** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Plus grand que | **>** |  |
| Plus petit que | **<** |  |
| Plus grand ou égal à | **>=** |  |
| Plus petit ou égal à | **<=** |  |
| Egal à | **==** | Attention! Trop souvent confondu avec l’opérateur d’affectation = |
| Différent de | **!=** |  |
| **Note :** Ces opérateurs servent à comparer des expressions. Le résultat des opérations est VRAI ou FAUX. | | |

# &&

|  |  |  |
| --- | --- | --- |
| Opérateurs logiques | | |
| **Fonction** | **Symbole** | **Remarques** |
| ET | **&&** | **( (a>2) && (a<10) )**  **VRAI** si **a = 6**, **FAUX** si **a = 16** |
| OU | **||** | **( (a<2) || (a>10) )**  **FAUX** si **a = 6**, **VRAI** si **a = 16** |
| Négation | **!** | **!( (a<2) || (a>10) )**  **VRAI** si **a = 6**, **FAUX** si **a = 16** |
| **Note :** Ces opérateurs servent à tester l’état logique d’une expression : **VRAI** ou **FAUX**, **OUI** ou **NON**, **0** ou **1**. **Une**  **valeur numérique peut également être testée : VRAI** si différente de **0, FAUX** si égale à **0**. | | |

|  |  |  |
| --- | --- | --- |
| **Opérateurs de manipulation de bits** | | |
| **Fonction** | **Symbole** | **Remarques** |
| ET | **&** | **c = a & b**;  si **a = 1001 0011** et **b = 0101 0110**, **c = 0001 0010** |
| OU | **|** | **c = a | b**;  si **a = 1001 0011** et **b = 0101 0110**, **c = 1101 0111** |
| OU exclusif | **^** | **c = a ^ b**;  si **a = 1001 0011** et **b = 0101 0110**, **c = 1100 0101** |
| Décalage à droite | **>>** | **(a>>2)** décale la valeur de a de **2** bits vers la droite |
| Décalage à gauche | **<<** | **(a<<2)** décale la valeur de a de **2** bits vers la gauche |
| Complément à un | **~** | **c = ~a**  si **a = 1001 0011**, **c = 0110 1100** |

**Note :** Ces opérateurs ne modifient pas la valeur de la variable, sauf si le résultat de l’opérateur affecte celle-ci : **a >> 2** ne change pas la valeur de **a**, **a = a >> 2** change la valeur de **a**.

|  |  |  |
| --- | --- | --- |
| **Opérateurs d’affectation** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Affectation simple | **=** |  |
| Ajoute et affecte | **+=** | **a += b**; équivaut à **a = a + b** ; |
| Soustrait et affecte | **-=** | **a -= b**; équivaut à **a = a - b** ; |
| Multiplie et affecte | **\*=** | **a \*= b**; équivaut à **a = a \* b** ; |
| Divise et affecte | **/=** | **a /= b**; équivaut à **a = a / b** ; |
| Modulo et affecte | **%=** | **a %= b**; équivaut à **a = a % b** ; |
| OU (bit) et affecte | **|=** | **a |= b**; équivaut à **a = a | b** ; |
| OU exclusif (bit) et affecte | **^=** | **a ^= b**; équivaut à **a = a ^ b** ; |
| ET (bit) et affecte | **&=** | **a &= b**; équivaut à **a = a & b** ; |
| Décale à droite et affecte | **>>=** | **a >>= b**; équivaut à **a = a >> b** ; |
| Décale à gauche et affecte | **<<=** | **a <<= b**; équivaut à **a = a << b** ; |
| **Note :** Ces opérateurs modifient la valeur des variables. | | |

|  |  |  |
| --- | --- | --- |
| **Opérateur d’évaluation séquentielle** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Evaluation séquentielle | **,** | **int a, b; b = 3, a = 94;** |

|  |  |  |
| --- | --- | --- |
| **Opérateur conditionnel** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Evaluation conditionnelle | **?:** | **min = (a<b) ?a :b;**  si **(a<b) est VRAI, min=a ;.**sinon **min=b ;** |
| **Note** : L’utilisation de cet opérateur fait partie des thèmes traités au chapitre 4. | | |

|  |  |  |
| --- | --- | --- |
| **Opérateurs unaires** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Incrémentation | **++** | **c=a + (b++)** ; **b=b + 1 après** exécution de l’instruction  **c=a + (++b)** ; **b=b + 1 avant** exécution de l’instruction |
| Décrémentation | **- -** | **c=a + (b--)** ; **b=b - 1 après** exécution de l’instruction  **c=a + (--b)** ; **b=b - 1 avant** exécution de l’instruction |
| Conversion explicite | **(nom Type)** | **char a; int b; b=(int)a**; une copie de la valeur de **a** est transformée en  **int** avant d’être déposé dans **b**. |
| Adressage indirect | **\*** | **\*ptr=a** ;  copie la valeur de **a** à l’adresse pointée par **ptr.** |
| Adresse de | **&** | **&a** est l’adresse de **a.** |
| Evalue grandeur mémoire | **sizeof** | **sizeof(expr). sizeof(nom Type).** |
| Plus unaire | **+** | force l’évaluation d’une expression avant une autre à cause de son niveau  de priorité. |
| Moins unaire | **-** | **-a** est le **complément à 2** de **a**. |
| **Note :** Un opérateur unaire est un opérateur qui nécessite 1 seul opérande : **a++.**  Un opérateur binaire est un opérateur qui nécessite 2 opérandes : **a + b**. | | |

|  |  |  |
| --- | --- | --- |
| **Opérateurs primaires** | | |
| **Fonction** | **Symbole** | **Remarques** |
| Parenthèse | **( )** |  |
| Expression d’indice | **[ ]** | **char tableau[8] ; tableau[3] = a + tableau[5]** ; |
| Sélecteur de membre | **->** | Sélection par pointeur : **ptrstruct->mois = 12 ;** |
| Sélecteur de membre | **.** | Sélection par la structure : **date.mois = 12 ;** |

## Niveau de priorité des opérateurs

Le niveau de priorité des opérateurs détermine dans quel ordre ceux-ci seront appliqués lors de l’évaluation d’une expression : les opérateurs de niveau plus élevé seront toujours évalués en premier.

Lorsque plusieurs opérateurs de même niveau de priorité se trouvent dans une expression, c’est l’associativité (gauche à droite *ou* droite à gauche), qui détermine si les opérations seront effectuées de droite à gauche ou de gauche à droite.

|  |  |  |  |
| --- | --- | --- | --- |
| **Priorité des opérateurs** | | | |
| **Niveau de priorité** | **Type d’opérateur** | **Opérateurs** | **Associativité** |
| 15 | Primaire | **( ) [ ] . ->** | gauche à droite |
| 14 | Unaire | **++ -- (nom Type) \* & sizeof + - ! ~** | **droite à gauche** |
| 13 | Arithmétique | **\* / %** | gauche à droite |
| 12 | Arithmétique | **+ -** | gauche à droite |
| 11 | De manipulation de bits | **>> <<** | gauche à droite |
| 10 | De relation | **> < >= <=** | gauche à droite |
| 9 | De relation | **== !=** | gauche à droite |
| 8 | De manipulation de bits | **&** | gauche à droite |
| 7 | De manipulation de bits | **^** | gauche à droite |
| 6 | De manipulation de bits | **|** | gauche à droite |
| 5 | Logique | **&&** | gauche à droite |
| 4 | Logique | **||** | gauche à droite |
| 3 | Conditionnel | **?:** | **droite à gauche** |
| 2 | D’affectation | **= += -= \*= /= |= ^= &= >>= <<=** | **droite à gauche** |
| 1 | D’évaluation séquentielle | **,** | gauche à droite |

# Les Instructions

* Une instruction simple est soit une expression terminée par *un point virgule*, soit une instruction de contrôle (traitée au chapitre 4) soit un bloc d’instructions délimité par **{** et **}**. Exemple : a=a +1**;**
* Les instructions composées (ou blocs) qui permettent de considérer une succession d'instructions comme étant une seule instruction :
  + elles commencent par "{" et finissent par "}"
  + l’accolade ouvrante est l’équivalente du début en algorithme
  + l’accolade fermante est l’équivalente de la fin en algorithme
  + par exemple :

## {

float tauxConversion=6.55957, valeurEnFranc; int valeurEnEuro=50;

valeurEnFranc = valeurEnEuro \* tauxConversion;

## }