Chapitre 2

Variables, types, opérateurs et expressions

Les variables

- Les variables servent à stocker les valeurs des données utilisées pendant l'exécution d'un programme

- Les variables doivent être **déclarées** avant d'être utilisées, elles doivent être caractérisées par :

  - un nom (**Identificateur**)
  - un **type** (entier, réel, …)

  (Les types de variables en C seront discutés par la suite)

Les identificateurs

Le choix d'un identificateur (nom d'une variable ou d'une fonction) est soumis à quelques règles :

- doit être constitué uniquement de lettres, de chiffres et du caractère souligné _ (Eviter les caractères de ponctuation et les espaces)
  correct: PRIX_HT, prixHT **incorrect: PRIX-HT, prix HT, prix.HT**

- doit commencer par une lettre (y compris le caractère souligné)
  correct : A1, _A1 **incorrect: 1A**

- doit être différent des mots réservés du langage : **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**

Remarque : C distingue les majuscules et les minuscules. NOMBRE et nombre sont des identificateurs différents

Les types de base

- Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre et le nombre d'octets à lui réserver en mémoire

- En langage C, il n'y a que deux types de base *les entiers* et *les réels* avec différentes variantes pour chaque type

Remarques:

- Un type de base est un type pour lequel une variable peut prendre une seule valeur à un instant donné contrairement aux types agrégés

- Le type caractère apparaît en C comme cas particulier du type entier (un caractère est un nombre entier, il s'identifie à son code ASCII)

- En C il n'existe pas de type spécial pour chaînes de caractères. Les moyens de traiter les chaînes de caractères seront présentés aux chapitres suivants

- Le type booléen n'existe pas. Un booléen est représenté par un entier (un entier non nul équivaut à vrai et la valeur zero équivaut à faux)

Types Entier

4 variantes d'entiers :

- **char** : caractères (entier sur 1 octet : - 128 **à** 127)
- **short ou short int** : entier court (entier sur 2 octets : - 32768 **à** 32767)
- **int** : entier standard (entier sur 2 ou 4 octets)
- **long ou long int** : entier long (4 octets : - 2147483648 **à** 2147483648)

Si on ajoute le préfixe **unsigned** à la définition d'un type de variables entières, alors la plage des valeurs change:

- **unsigned char** : 0 à 255
- **unsigned short** : 0 à 65535
- **unsigned int** : dépend du codage (sur 2 ou 4 octets)
- **unsigned long** : 0 à 4294967295

Remarque : Une variable du type **char** peut subir les mêmes opérations que les variables du type **short**, **int** ou **long**

Types Réel

3 variantes de réels :

- **float** : réel simple précision codé sur 4 octets de
  $-3.4*10^{38}$ à $3.4*10^{38}$
- **double** : réel double précision codé sur 8 octets de
  $-1.7*10^{308}$ à $1.7*10^{308}$
- **long double** : réel très grande précision codé sur 10 octets de **$-3.4*10^{4932}$ à $3.4*10^{4932}$**

Déclaration des variables

- Les *déclarations* introduisent les variables qui seront utilisées, fixent leur type et parfois aussi leur valeur de départ (initialisation)

- Syntaxe de déclaration en C

  <Type> <NomVar1>,<NomVar2>,...,<NomVarN>;

- Exemple:

```
int   i, j,k;
float  x, y ;
double z=1.5; // déclaration et initialisation
short compteur;
char c=`A`;
```

Déclaration des constantes

- Une constante conserve sa valeur pendant toute l'exécution d'un programme
- En C, on associe une valeur à une constante en utilisant :
  - la directive *#define :*
    #define nom_constante valeur
    Ici la constante ne possède pas de type.
    exemple: *#define Pi 3.141592*
  - le mot clé *const* :
    const *type nom = expression* ;
    Dans cette instruction la constante est typée
    exemple : *const float Pi =3.141592*

(Rq: L'intérêt des constantes est de donner un nom parlant à une valeur, par exemple NB_LIGNES, aussi ça facilite la modification du code)

Constantes entières

On distingue 3 formes de constantes entières :

- forme décimale *:* c'est l'écriture usuelle. Ex : 372, 200
- forme octale (base 8) : on commence par un 0 suivi de chiffres octaux. Ex : 0477
- forme hexadécimale (base 16) : on commence par 0x (ou 0X) suivis de chiffres hexadécimaux (0-9 a-f). Ex : 0x5a2b, 0Xa9f

Remarques sur les constantes entières

- Le compilateur attribue automatiquement un type aux constantes entières. Il attribue en général le type le plus économique parmi (int, unsigned int, long int, unsigned long int)

- On peut forcer la machine à utiliser un type de notre choix en ajoutant les suffixes suivants:
  - u ou U pour unsigned int, Ex : 100U, 0xAu
  - l ou L pour long, Ex : 15l, 0127L
  - ul ou UL pour unsigned long, Ex : 1236UL, 035ul

Constantes réelles

On distingue 2 notations :

- notation décimale Ex : 123.4, .27, 5.
- notation exponentielle Ex : 1234e-1 ou 1234E-1

Remarques :

- Les constantes réelles sont par défaut de type double
- On peut forcer la machine à utiliser un type de notre choix en ajoutant les suffixes suivants:
  - f ou F pour le type float, Ex: 1.25f
  - l ou L pour le type long double, EX: 1.0L

Les constantes caractères

- Se sont des constantes qui désignent un seul caractère, elles sont toujours indiquées entre des apostrophes, Ex : ‘b’, ‘A’, ‘?’
- La valeur d’une constante caractère est le code ASCII du caractère
- Les caractères constants peuvent apparaître dans des opérations arithmétiques ou logiques
- Les constantes caractères sont de type int

Expressions et opérateurs

- Une *expression* peut être une valeur, une variable ou une opération constituée par des valeurs, des constantes et des variables reliées entre eux par des *opérateurs*

  exemples: 1, b, a*2, a+ 3*b-c, …

- Un *opérateur* est un symbole qui permet de manipuler une ou plusieurs variables pour produire un résultat. On distingue :
  - les *opérateurs binaires* qui nécessitent deux opérandes (ex : a + b)
  - les *opérateurs unaires* qui nécessitent un seul opérande *(ex: a++)*
  - *l'opérateur conditionnel ?:* , le seul qui nécessite trois opérandes

- Une expression fournit une seule valeur, elle est évaluée en respectant des règles de priorité et d'associativité

Opérateurs en C

- Le langage C est riche en opérateurs. Outre les opérateurs standards, il comporte des opérateurs originaux d'affectation, d'incrémentation et de manipulation de bits

- On distingue les opérateurs suivants en C :

  - **les opérateurs arithmétiques : +, -, *, /, %**
  - **les opérateurs d'affectation : =, +=, -=,*=,/=,…**
  - **les opérateurs logiques : &&, ||, !**
  - **les opérateurs de comparaison** : **==, !=, <, >, <=, >=**
  - **les opérateurs d'incrémentation et de décrémentation : ++, --**
  - **les opérateurs sur les bits : <<, >>, &, |, ~, ^**
  - **d'autres opérateurs particuliers :** ?:, sizeof, cast

Opérateurs arithmétiques

- binaires : + - * / et % (modulo) et unaire : -
- Les opérandes peuvent être des entiers ou des réels sauf pour % qui agit uniquement sur des entiers
- Lorsque les types des deux opérandes sont différents il y'a conversion implicite dans le type le plus fort
- L'opérateur / retourne un quotient entier si les deux opérandes sont des entiers (5 / 2 ➔ 2). Il retourne un quotient réel si l'un au moins des opérandes est un réel (5.0 / 2 ➔ 2.5)

Conversions implicites

- Les types short et char sont systématiquement convertis en int indépendemment des autres opérandes

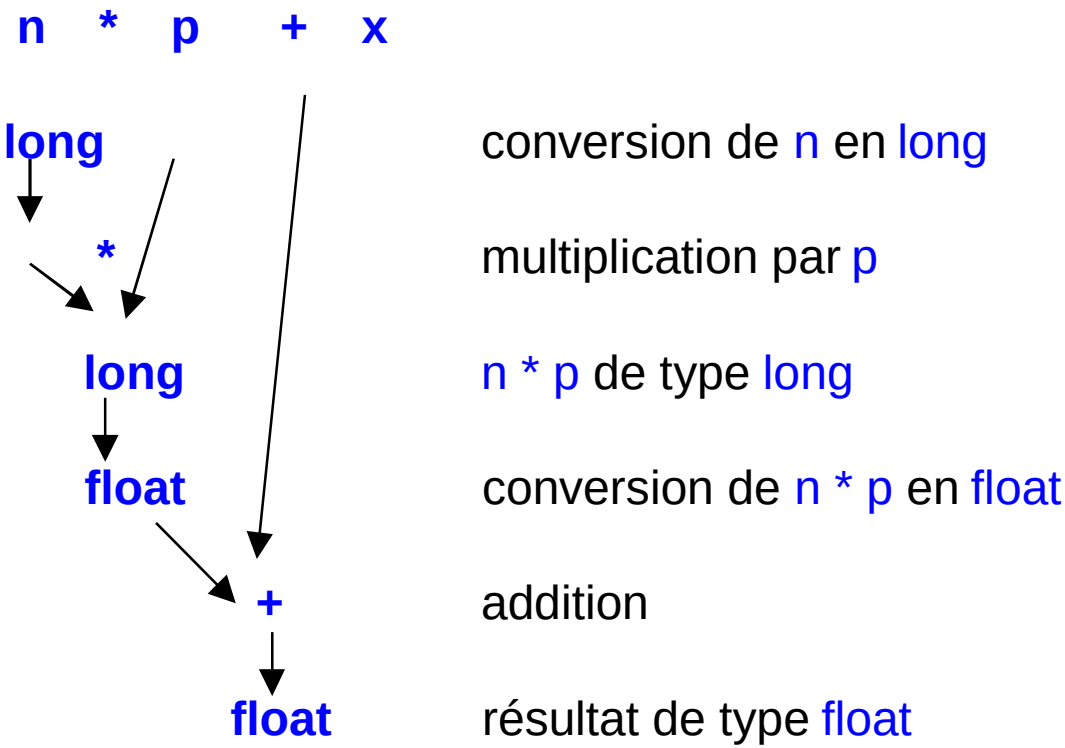
- La conversion se fait en général selon une hiérarchie qui n'altère pas les valeurs int → long → float → double → long double

- **Exemple1 :** n * x + p (int n,p; float x)
  - exécution prioritaire de n * x : conversion de n en float
  - exécution de l'addition : conversion de p en float

- **Exemple2 :** p1 * p2 + p3 * x (char p1, short p2, p3 ; float x)
  - p1, p2 et p3 d'abord convertis en int
  - p3 converti en float avant multiplication

Exemple de conversion

Exemple : n * p + x (int n ; long p ; float x)



Opérateur d'affectation simple =

- L'opérateur = affecte une valeur ou une expression à une variable
  - Exemple: double x,y,z; x=2.5; y=0.7; z=x*y-3;
- Le terme à gauche de l'affectation est appelé *lvalue* (left value)
- L'affectation est interprétée comme une expression. La valeur de l'expression est la valeur affectée
- On peut enchainer des affectations, l'évaluation se fait de droite à gauche
  - exemple : i = j = k= 5 (est équivalente à k = 5, j=k et ensuite i=j)
- La valeur affectée est toujours convertie dans le type de la lvalue, même si ce type est plus faible (ex : conversion de float en int, avec perte d'information)

Opérateurs relationnels

- **Opérateurs**
  - < : inférieur à
  - <= : inférieur ou égal à
  - > : supérieur à
  - >= : supérieur ou égal à
  - == : égal à
  - != : différent de

- Le résultat de la comparaison n'est pas une valeur booléenne, mais 0 si le résultat est faux et 1 si le résultat est vrai

- Les expressions relationnelles peuvent donc intervenir dans des expressions arithmétiques

- Exemple: a=2, b=7, c=4
  - b==3 ➔ 0 (faux)
  - a!=b ➔ 1(vrai)
  - 4*(a<b) + 2*(c>=b)➔ 4

Opérateurs logiques

- && : ET logique || : OU logique ! : négation logique
- && retourne vrai si les deux opérandes sont vrais (valent 1) et 0 sinon
- || retourne vrai si l'une des opérandes est vrai (vaut 1) et 0 sinon
- Les valeurs numériques sont acceptées : toute valeur non nulle correspond à vraie et 0 correspond à faux
  - Exemple : 5 && 11 ➔ 1
    !13.7 ➔ 0

Évaluation de && et ||

- Le 2ème opérande est évalué uniquement en cas de nécessité
  - a && b : b évalué uniquement si a vaut vrai (si a vaut faux, évaluation de b inutile car a && b vaut faux)
  - **a** || b : b évalué uniquement si a vaut faux (si a vaut vrai, évaluation de b inutile car a || b vaut vrai)

- **Exemples**
  - if ((d != 0) && (n / d == 2)) : pas de division si d vaut 0
  - if ((n >=0) && (sqrt(n) < p)) : racine non calculée si n < 0

- L'intérêt est d'accélérer l'évaluation et d'éviter les traitements inappropriés

Incrémentation et décrémentation

- Les opérateurs ++ et -- sont des opérateurs unaires permettant respectivement d'ajouter et de retrancher 1 au contenu de leur opérande
- Cette opération est effectuée après ou avant l'évaluation de l'expression suivant que l'opérateur suit ou précède son opérande
  - k = *i*++ *(post-incrémentation)* affecte d'abord la valeur de i à k et incrémente après (k = i++ ; ⇔ k = i ; i = i+1 ;)
  - k = ++*i (pré-incrémentation*) incrémente d'abord et après affecte la valeur incrémentée à k (k = ++i ; ⇔ i = i+1 ; k = i ;)
- Exemple : i = 5 ; n = ++i - 5 ;

  i vaut 6 et n vaut 1

  i = 5 ; n = i++ - 5 ;

  i vaut 6 et n vaut 0
- Remarque : idem pour l'opérateur de décrémentation --

Opérateurs de manipulations de bits

- **opérateurs arithmétiques bit à bit :**

  & : ET logique **|** : OU logique ^ : OU exclusif ~ : négation

- Les opérandes sont de type entier. Les opérations s'effectuent bit à bit suivant la logique binaire

b1	b2	~b1	b1&b2	b1 | b2	b1^b2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

- Ex : 14= 1110 , 9=1001➔ 14 & 9= 1000=8, 14 | 9 =1111=15

Opérateurs de décalage de bits

- Il existe deux opérateurs de décalage **:**

  >> : décalage à droite **<<** : décalage à gauche

- L'opérande gauche constitue l'objet à décaler et l'opérande droit le nombre de bits de décalage

- Dans le cas d'un décalage à gauche les bits les plus à gauche sont perdus. Les positions binaires rendues vacantes sont remplies par des 0

  Ex : char x=14; (14=00001110) ➔ 14<<2 = 00111000 = 56

  char y=-7; (-7=11111001) ➔ -7<<2= 11100100**= -28**

- Rq : un décalage à gauche de k bits correspond (sauf débordement) à la multiplication par 2^k

Opérateurs de décalage de bits

- Lors d'un décalage à droite les bits les plus à droite sont perdus.
  - si l'entier à décaler est non signé, les positions binaires rendues vacantes sont remplies par des 0
  - s'il est signé le remplissage dépend de l'implémentation (en général le remplissage se fait par le bit du signe)

  Ex : char x=14; (14=00001110) ➔ 14>>2 = **00**000011 = 3

- Remarque : un décalage à droite (n >> k) correspond à la division entière par 2^k si n est non signé

Opérateurs d'affectation combinés

- Soit un opérateur de calcul ***op*** et deux expressions *exp1* et *exp2*. L'expression *exp1= exp1 op exp2* peut s'écrire en général de façon équivalente sous la forme *exp1 op= exp2*

- Opérateurs utilisables :
  - += -= *= /= %=
  - <<= >>= &= ^= |=

- Exemples :
  - a=a+b s'écrit : a+=b
  - n=n%2 s'écrit : n%=2
  - x=x*i s'écrit : x*=i
  - p=p>>3 s'écrit : p>>=3

Opérateur de forçage de type (cast)

- Il est possible d'effectuer des conversions explicites ou de forcer le type d'une expression
  - Syntaxe : (<type>) <expression>
  - Exemple : int n, p ;
    (double) (n / p); convertit l'entier n / p en double

- Remarque : la conversion (ou casting) se fait après calcul

  (double) (n/p) ≠ (double) n / p ≠ (double) (n) / (double) (p)

  - float n = 4.6, p = 1.5 ;
    (int) n / (int) p = 4 / 1 = 4
    (int) n / p = 4 / 1.5 = 2.66
    n / (int) p = 4.6 / 1 = 4.6
    n / p = 4.6 / 1.5 = 3.06

Opérateur conditionnel ? :

- **Syntaxe: exp1 ? exp2 : exp3**

  exp1 est évaluée, si sa valeur est non nulle c'est exp2 qui est exécutée, sinon exp3

- **Exemple1 :** max = a > b ? a : b

  Si a>b alors on affecte à max le contenu de exp2 càd a sinon on lui affecte b

- **Exemple2 :** a>b ? i++ : i--;

  Si a>b on incrémente i sinon on décrémente i

Opérateur séquentiel ,

- Utilité : regrouper plusieurs sous-expressions ou calculs en une seule expression
- Les calculs sont évalués en séquence de gauche à droite
- La valeur de l'expression est celle de la dernière sous-expression
- Exemples
  - i++ , i + j; // on évalue i++ ensuite i+j (on utilise la valeur de i incrémentée)
  - i++ , j = i + k , a + b; // la valeur de l'expression est celle de a+b
  - for (i=1 , k=0 ; … ; …) { }

Opérateur SIZEOF

- **Syntaxe : sizeof (<type>) ou sizeof (<variable>)**

  fournit la taille en octets d'un type ou d'une variable

- **Exemples**
  - int n;
  - printf ("%d \n",sizeof(int)); // affiche 4
  - printf ("%d \n",sizeof(n)); // affiche 4

Priorité et associativité des opérateurs

- Une expression est évaluée en respectant des règles de priorité et d'associativité des opérateurs
  - Ex: * est plus prioritaire que +, ainsi 2 + 3 * 7 vaut 23 et non 35
- Le tableau de la page suivante donne la priorité de tous les opérateurs. La priorité est décroissante de haut en bas dans le tableau.
- Les opérateurs dans une même ligne ont le même niveau de priorité. Dans ce cas on applique les règles d'associativité selon le sens de la flèche. Par exemple: 13%3*4 vaut 4 et non 1
- Remarque: en cas de doute il vaut mieux utiliser les parenthèses pour indiquer les opérations à effectuer en priorité. Ex: (2 + 3) * 7 vaut 35

Priorités de tous les opérateurs

Catégorie	Opérateurs	Associativité
référence	() [] -> .	→
unaire	+ - ++ -- ! ~ * & (cast) sizeof	←
arithmétique	* / %	→
arithmétique	+ -	→
décalage	<< >>	→
relationnel	< <= > >=	→
relationnel	== !=	→
manip. de bits	&	→
manip. de bits	^	→
manip. de bits	|	→
logique	&&	→
logique	||	→
conditionnel	? :	→
affectation	= += -= *= /= %=	←
	&= ^= |= <<= >>=	←
séquentiel	,	→