Types élémentaires

Le type vide: **void**

Le type caractère: **char**

Ex: **char** var ='a', Var= 'B', foo='\n', Foo='\014';

Les types entiers: **short, int, long**

Ex: **short** si=120; int i=-10, j=0123; **long** l=0xfc, k=1234555;

entiers positifs en préfixant les types précédents par **unsigned**

Ex: **unsigned int** ui; **unsigned char** uc=0;

Les types réels: **float, double**

Ex: **float** f = 1.23, fb=1.2345e+12, fbb= 1.;

double d = 1.23, db=1.23456789012345e+12;

Booléens

<u>En C</u>: il n’y a pas de ***type booléen***, par contre, les expressions booléennes existent. Par convention, toute valeur *entière* peut être considérée comme une valeur booléenne. Tel que:

***Faux* est codé par l'entier nul**

***Vrai* est codé par un entier non nul**

<u>En C++ ANSI</u>: un nouveau type **bool** est ajouté, les valeurs de ce type sont *true* et *false*.

- la conversion d'une valeur numérique en un **bool** rend *fals*e pour 0 et *true* pour n'importe quelle valeur non nulle,
- une valeur de type **bool**, convertie en **int** est égale à 0 pour *false* et à 1 pour *true*,
- les instructions **if** et **while** sont converties en **bool**.

LES DECLARATIONS DE CONSTANTES (1)

Le langage C autorise 2 méthodes pour définir des constantes.

1-ère méthode: déclaration d'une variable, dont la valeur sera constante pour tout le programme:

Exemple:

```
void main()
{
    const float pi = 3.14159;
    float perimetre, rayon = 8.7;
    perimetre =  2*rayon*pi;
    ....
}
```

Dans ce cas, le compilateur réserve de la place en mémoire (ici 4 octets), pour la variable *pi*.

La valeur de *pi* ne peut changer dans le programme.

LES DECLARATIONS DE CONSTANTES (2)

2-ème méthode: **définition d'un symbole** à l'aide de la directive de compilation **#define.**

Exemple:

```
#define PI  3.14159
void main()
{
    float perimetre,rayon = 8.7;
    perimetre =  2*rayon*PI;
    ....
}
```

Les constantes déclarées par #define s'écrivent traditionnellement en majuscules, mais ce n'est pas une obligation.

Expressions

Une expression se compose d'une ou de plusieurs opérations. Différents types d'opérations:

- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateur d'affectation
- Opérateurs relationnels
- Opérateurs d'accès aux objets

Conversion de type: l'ordre d'évaluation dans une expression est déterminé par des règles de *priorité* et d'*associativité*.

Important: le type de donnée du résultat est déterminé par le type des opérandes.

Déclarations

Les régles:

- TOUTE VARIABLE doit être déclarée AVANT son utilisation.

  a**) En C++**, cette déclaration peut avoir lieu n'importe où dans un bloc,

  b) **En C**, cette déclaration doit être en début de bloc.

- TOUTE FONCTION doit être déclarée AVANT son appel (avec la liste des types de ses arguments).

Exemples:

```
int r=0, c, d=1;
static int K;
char chaine[300] = "ceci est un exemple";
int fonc(char *, int);
int a;
int  b=0;
for (int i=0; i<10; i++) b += i*i;
```

Opérateurs arithmétiques

Opérateur	Rôle	Usage
+	addition (binaire)	expr + expr + ...
-	soustraction (unaire)	- expr
-	soustraction (binaire)	expr - expr
*	multiplication (binaire)	expr * expr
/	division (binaire)	expr / expr
%	modulo (binaire)	expr % expr

Fonctions supplémentaires sont disponibles dans la bibliothèque mathématique **pow**, **exp**, **log**, **sin**, etc.

Fichier d'en-tête: <**math.h**> - bibliothèque: **libm.a**.

Opérateurs d'incrémentation/décrémentation

L'opérateur d'incrémentation **(++)** et l'opérateur de décrémentation **(--)** fournissent un moyen efficace pour *compacter des écritures.*

[**Post/Pré] [incr/décr]émentation:**

x++ ; y-- ; post-incr(décr)émentation

++x ; --y ; pré-incr(décr)émentation

Exemples:

```
pile[x++] = val ;   équivalent à:

                                  pile[x] = val ;
                                  x = x + 1 ;

val = pile[--x] ;   équivalent à:

                                  x = x - 1 ;
                                  val = pile[x] ;
```

Opérateurs bits à bits

Opérateur	Rôle	Usage
&	ET	op1 & op2
|	OU (inclusif)	op1 | op2
^	OU exclusif	op1 ^ op2
~	Négation	~ op1
<<	décalage à gauche (× 2)	op1 << op2
>>	décalage à droite (÷ 2)	op1 >> op2

- On se sert du ET (&) pour masquer certains bits:

  n = n & 0177 ;

- On se sert de OU (|) pour mettre des bits à un:

  x = x | MISE_A_UN ;

- L'opérateur unaire (~) met à un les bits qui sont à zéro et vice-versa:

  ~ 0

  x >> (p + 1) & ~ (~0 << n) ;

Opérateurs relationnels

Opérateur	Rôle	Usage
==	égalité	expr == expr
!=	inégalité	expr != expr
>	supérieur	expr > expr
>=	supérieur ou égal	expr >= expr
<	inférieur	expr < expr
<=	inférieur ou égal	expr <= expr
!	non logique	!expr
&&	et logique	expr && expr
||	ou logique	expr || expr

Le résultat est une expression booléenne.

Attention: arrêt de l'évaluation dès que le premier opérande est faux (dans le cas &&) ou vrai (dans le cas ||).

Conversion de type

Le compilateur fait de lui-même des conversions lors de l'évaluation des expressions. Pour cela il applique des règles de conversion implicite. Ces règles ont pour but la perte du minimum d'information dans l'évaluation de l'expression. Ces règles sont décrites dans la table ci-dessous.

Le type dans lequel le calcul d'une expression à deux opérandes doit se faire est donné par les règles suivantes:

- si l'un des deux opérandes est du type **long double** alors le calcul doit être fait dans le type **long double**;
- sinon, si l'un des deux opérandes est du type **double** alors le calcul doit être fait dans le type **double**;
- sinon, si l'un des deux opérandes est du type **float** alors le calcul doit être fait dans le type **float**;
- sinon, appliquer la règle de **promotion des entiers**.

Exemples de conversion implicite

float f; **double** d; **int** i; **long** li;

li = f + i ; i est transformé en float puis additionné à f,

d = li + i ; i est transformé en long puis additionné à li,

i = f + d ; f est transformé en double, additionné à d.

Opérateur cast

Il est possible de forcer la conversion d'une variable (ou d'une expression) dans un autre type avant de l'utiliser par une conversion implicite. Cette opération est appelée "**cast**". Elle se réalise de la manière suivante:

i = (**int**) f + (**int**) d ;

f et d sont convertis en int, puis additionnés.

Boucles et tests

- Boucle **for**
- Boucle **while**
- Boucle **do** {…} **while**
- Test **if**...**then**…**else**
- Test **switch**

Exemples:

for (i=0; i < 10; i++)

for (y = *p, p += 2; p < stop; y = *p, p += 2)

while (nb < argc) { printf ("argument %d : %s\n", nb, argv[nb]); nb++; }

do { printf ("argument %d : %s\n", nb, argv[nb]); nb++; }**while** (nb < argc);

if (!recu) **printf** ("rien reçu\n");