***Récapitulatif du vocabulaire*** :
Les *variables* et les *constantes* sont les données principales qui peuvent être manipulées par un programme. Les *déclarations* introduisent les variables qui sont utilisées, fixent leur type et parfois aussi leur valeur de départ. Les *opérateurs* contrôlent les actions que subissent les valeurs des données. Pour produire de nouvelles valeurs, les variables et les constantes peuvent être combinées à l'aide des opérateurs dans des *expressions*. Le *type* d'une donnée détermine l'ensemble des valeurs admissibles, le nombre d'octets à réserver en mémoire et l'ensemble des opérateurs qui peuvent y être appliqués.

## I) Les types simples :

### Les types entiers :

Avant de pouvoir utiliser une variable, nous devons nous intéresser à deux caractéristiques de son type numérique :

- le domaine des valeurs admissibles,
- le nombre d'octets qui est réservé pour une variable.

| *Définition* | *Description* | *Domaine Min* | *Domaine Max* | *Nombre D'octets* |
|---|---|---|---|---|
| **char** | caractère | -128 | 127 | 1 |
| **short** | entier court | -32768 | 32767 | 2 |
| **int** | Entier standard | -32768 | 32767 | 2 |
| **long** | entier long | -2147483648 | 2147483647 | 4 |

- **char** : caractère

  Une variable du type **char** peut contenir une valeur entre -128 et 127 et elle peut subir les mêmes opérations que les variables du type **short**, **int** ou **long**.

- **int** : entier standard

  Sur chaque machine, le type **int** est le type de base pour les calculs avec les entiers. Le codage des variables du type **int** est donc dépendant de la machine. Sur les IBM-PC sous MS-DOS, une variable du type **int** est codée dans deux octets.

- **short** : entier court

  Le type **short** est en général codé dans 2 octets. Comme une variable **int** occupe aussi 2 octets sur notre système, le type **short** devient seulement nécessaire, si on veut utiliser le même programme sur d'autres machines, sur lesquelles le type standard des entiers n'est pas forcément 2 octets.

  - Les modificateurs **signed/unsigned**:

| Définition | Description | Domaine Min | Domaine Max | Nombre d'octets |
|---|---|---|---|---|
| unsigned char | caractère | 0 | 255 | 1 |
| unsigned short | entier court | 0 | 65535 | 2 |
| unsigned int | entier standard | 0 | 65535 | 2 |
| unsigned long | entier long | 0 | 4294967295 | 4 |

***Remarques :***

**1.** Le calcul avec des entiers définis comme **unsigned** correspond à l'arithmétique ***modulo $2^n$***. Ainsi, en utilisant une variable X du type **unsigned short**, nous obtenons le résultat suivant :

Affectation : X = 65500 + 100 * [$2^{15}$] */

Résultat : X=64

**2.** Par défaut, les types entiers **short**, **int**, **long** sont munis d'un signe. Le type par défaut de **char** est dépendant du compilateur et peut être **signed** ou **unsigned**. Ainsi, l'attribut **signed** a seulement un sens en liaison avec **char** et peut forcer la machine à utiliser la représentation des caractères avec signe (qui n'est cependant pas très usuelle).

**3.** Les valeurs limites des différents types sont indiquées dans le fichier *header* <**limits.h**>.

**4.** En principe, on peut dire que :

```
sizeof(short) <= sizeof(int) <= sizeof(long)
```

Ainsi sur certaine architecture on peut avoir

```
short = 2 octets, int = 2 octets, long = 4 octets
```

et sur d'autre

```
short = 2 octets, int = 4 octets, long = 4 octets (
```

Il sera intéressant de voir l'implémentation dans un environnement 64 bits! )

## 2) Les types rationnels :

En informatique, les rationnels sont souvent appelés des 'flottants'. Ce terme vient de *'en virgule flottante'* et trouve sa racine dans la notation traditionnelle des rationnels :

| | <+\|-> <***mantisse***> * 10$^{<exposant>}$ |
|---|---|
| <+ -> | *est le signe positif ou négatif du nombre* |
| <*mantisse*> | *est un décimal positif avec un seul chiffre devant la virgule.* |
| <*exposant*> | *est un entier relatif* |

***Exemples :***

```
3.14159*100      1.25003*10-12
4.3001*10321     -1.5*103
```

En C, nous avons le choix entre trois types de rationnels : **float**, **double** et **long double**. Dans le tableau ci-dessous, vous trouverez leurs caractéristiques :

| | |
|---|---|
| *min et max* | *représentent les valeurs minimales et maximales **positives**. Les valeurs négatives peuvent varier dans les mêmes domaines.* |
| *Mantisse* | *indique le nombre de chiffres significatifs de la mantisse.* |

| *Définition* | *Domaine min* | *Domaine max* | *Nombre d'octets* |
|---|---|---|---|
| **Float** | 3.4 * $10^{-38}$ | 3.4 * $10^{38}$ | 4 |
| **Double** | 1.7 * $10^{-308}$ | 1.7 * $10^{308}$ | 8 |
| **Long double** | 3.4 * $10^{-4932}$ | 1.1 * $10^{4932}$ | 10 |

***Remarque avancée :***

Les détails de l'implémentation sont indiqués dans le fichier header <**float.h**>.

## II) La déclaration des variables simples :

Maintenant que nous connaissons les principaux types de variables, il nous faut encore la syntaxe pour leur déclaration :

***Déclaration de variables en langage algorithmique :***

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

***Déclaration de variables en C :***

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

Prenons quelques déclarations du langage descriptif,

```
entier    COMPTEUR,X,Y
réel    HAUTEUR,LARGEUR,MASSE_ATOMIQUE
caractère    TOUCHE
booléen    T_PRESSEE
```

et traduisons-les en des déclarations du langage C :

```
int    compteur, X, Y;
float    hauteur, largeur;
double    masse_atomique;
char    touche;
int    t_pressee;
```

***Langage algorithmique --> C :***

En général, nous avons le choix entre plusieurs types et nous devons trouver celui qui correspond le mieux au domaine et aux valeurs à traiter. Voici quelques règles générales qui concernent la traduction des déclarations de variables numériques du langage algorithmique en C :

- La ***syntaxe*** des déclarations en C ressemble à celle du langage algorithmique. Remarquez quand même les points virgules à la fin des déclarations en C.

**Entier** : Nous avons le choix entre tous les types entiers (inclusivement **char**) dans leurs formes **signed** ou **unsigned**. Si les nombres deviennent trop grands pour **unsigned long**, il faut utiliser un type rationnel (p.ex : **double**)

**Réel** : Nous pouvons choisir entre les trois types rationnels en observant non seulement la grandeur maximale de l'exposant, mais plus encore le nombre de chiffres significatifs de la mantisse.

**Caractère** : Toute variable du type **char** peut contenir un (seul) caractère. En C, il faut toujours être conscient que ce 'caractère' n'est autre chose qu'un nombre correspondant à un code (ici : code ASCII). Ce nombre peut être intégré dans toute sorte d'opérations algébriques ou logiques ...

**Chaîne** : En C il n'existe pas de type spécial pour chaînes de caractères. Les moyens de traiter les chaînes de caractères seront décrits ultérieurement.

**Booléen** : En C il n'existe pas de type spécial pour variables booléennes. Tous les types de variables numériques peuvent être utilisés pour exprimer des opérations logiques :

| |
|---|
| valeur logique faux <==> valeur numérique zéro |
| valeur logique vrai <==>toute valeur différente de zéro |

Si l'utilisation d'une variable booléenne est indispensable, le plus naturel sera d'utiliser une variable du type **int**.

Les opérations logiques en C retournent toujours des résultats du type **int** :

0 pour faux

1 pour vrai

### 1) Les constantes numériques :

En pratique, nous utilisons souvent des valeurs constantes pour calculer, pour initialiser des variables, pour les comparer aux variables, etc. Dans ces cas, l'ordinateur doit attribuer un type numérique aux

valeurs constantes. Pour pouvoir prévoir le résultat et le type exact des calculs, il est important pour le programmeur de connaître les règles selon lesquelles l'ordinateur choisit les types pour les constantes.

### a) Les constantes entières :

#### *Type automatique :*

Lors de l'attribution d'un type à une constante entière, C choisit en général la solution la plus économique :

Le type des constantes entières :

* Si possible, les constantes entières obtiennent le type **int.**
* Si le nombre est trop grand pour **int** (par ex : -40000 ou +40000) il aura automatiquement le type **long.**
* Si le nombre est trop grand pour **long**, il aura le type **unsigned long.**
* Si le nombre est trop grand pour **unsigned long**, la réaction du programme est imprévisible.

#### *Type forcé :*

Si nous voulons forcer l'ordinateur à utiliser un type de notre choix, nous pouvons employer les suffixes suivants :

| suffixe | type | Exemple |
|---|---|---|
| u ou U | unsigned (int ou long) | 550u |
| l ou L | long | 123456789L |
| ul ou UL | unsigned long | 12092UL |

#### *Exemples*

| | |
|---|---|
| **12345** | |
| **52000** | type **long** |
| **-2** | type **int** |
| **0** | type **int** |
| **1u** | type **unsigned int** |
| **52000u** | type **unsigned long** |
| **22lu** | ***Erreur !*** |

#### *Base octale et hexadécimale :*

Il est possible de déclarer des constantes entières en utilisant la base ***octale*** ou ***hexadécimale*** :

* Si une constante entière commence par **0** (zéro), alors elle est interprétée en base octale.
* Si une constante entière commence par **0x** ou **0X** , alors elle est interprétée en base hexadécimale.

***Exemples*** :

| base décimale | base octale | base hexadécimale | représentation binaire |
|---|---|---|---|
| 100 | 0144 | 0X64 | 1100100 |
| 255 | 0377 | 0xff | 11111111 |
| 65536 | 0200000 | 0X10000 | 10000000000000000 |
| 12 | 014 | 0XC | 1100 |
| 4040 | 07710 | 0xFC8 | 111111001000 |

### b) Les constantes rationnelles :

Les constantes rationnelles peuvent être indiquées :

* *en notation décimale*, c'est-à-dire à l'aide d'un point décimal :

***Exemples :***

```
123.4        -0.001        1.0
```

* *en notation exponentielle*, c'est-à-dire à l'aide d'un exposant séparé du nombre décimal par les caractères 'e' ou 'E' :

***Exemples :***

```
1234e-1-1E-30.01E2
```

l'ordinateur reconnaît les constantes rationnelles au point décimal ou au séparateur de l'exposant ('e' ou 'E'). Par défaut, les constantes rationnelles sont du type **double**.

***Le type des constantes rationnelles :***

* Sans suffixe, les constantes rationnelles sont du type **double**.
* Le suffixe **f** ou **F** force l'utilisation du type **float**.
* Le suffixe **l** ou **L** force l'utilisation du type **long double**.

### c) Les caractères constants :

Les constantes qui désignent un (seul) caractère sont toujours indiquées entre des apostrophes : par exemple **'x'**. La valeur d'un caractère constant est le code interne de ce caractère. Ce code (ici : le code ASCII) est dépendant de la machine.

Les caractères constants peuvent apparaître dans des opérations arithmétiques ou logiques, mais en général ils sont utilisés pour être comparés à des variables.

### d) Séquences d'échappement :

Une séquence d'échappement est un couple de symboles dont le premier est le *signe d'échappement* '\'. Au moment de la compilation, chaque séquence d'échappement est traduite en un caractère de contrôle dans le code de la machine. Comme les séquences d'échappement sont identiques sur toutes les machines, elles nous permettent d'écrire des programmes portables, c'est-à-dire : des programmes qui ont le même effet sur toutes les machines, indépendamment du code de caractères utilisé.

***Séquences d'échappement :***

| | | | |
|---|---|---|---|
| **\a** | *sonnerie* | **** | *trait oblique* |
| **\b** | *curseur arrière* | **\?** | *point d'interrogation* |
| **\t** | *tabulation* | **\'** | *apostrophe* |
| **\n** | *nouvelle ligne* | **\"** | *guillemets* |
| **\r** | *retour au début de ligne* | **\f** | *saut de page (imprimante)* |
| **\0** | *NUL* | **\v** | *tabulateur vertical* |

***Le caractère NUL :***

La constante **'\0'** qui a la valeur numérique zéro (ne pas à confondre avec le caractère '0' !!) a une signification spéciale dans le traitement et la mémorisation des chaînes de caractères : En C le caractère '\0' définit **la fin d'une chaîne** de caractères.

## 2) Initialisation des variables :

***Initialisation :***

En C, il est possible d'initialiser les variables lors de leur déclaration :

```
int   MAX = 1023;
char   TAB = '\t';
float   X = 1.05e-4;
```

***const :***

En utilisant l'attribut **const**, nous pouvons indiquer que la valeur d'une variable ne change pas au cours d'un programme :

```
        const int    MAX = 767;
const double e = 2.71828182845905;
const char NEWLINE = '\n';
```

## III) Les opérateurs standard :

***Affectation en langage algorithmique :***

<NomVariable> ← <Expression>

***Affectation en C :***

**<NomVariable> = <Expression>;**

### 1) Exemples d'affectations :

Reprenons les exemples du cours d'algorithmique pour illustrer différents types d'affectations :

a) **- L'affectation avec des valeurs constantes**

| algorithmique | C | Type de la constante |
|---|---|---|
| LONG ← 141 | LONG = 141; | (const. entière) |
| PI ← 3.1415926 | PI = 3.1415926; | (const. réelle) |
| NATION ← "L" | NATION = 'L'; | (caractère const.) |

**b) - L'affectation avec des valeurs de variables :**

| algorithmique | C |
|---|---|
| VALEUR ← X1A | VALEUR = X1A; |
| LETTRE ← COURRIER | LETTRE =COURRIER; |

c) **L'affectation avec des valeurs d'expressions :**

| algorithmique | C |
|---|---|
| AIRE ←PI*$R^2$ | AIRE = PI*pow(R,2); |
| MOYENNE ← (A+B)/2 | MOYENNE = (A+B)/2; |
| $U_N$ ← $\sin^2(X)+\cos^2(X)$ | $U_N$=pow(sin(X),2)+pow(cos(X),2); |
| RES ← 45+5*X | RES = 45+5*X; |
| PLUSGRAND ← (X>Y) | PLUSGRAND = (X>Y); |
| CORRECT ← ('a'='a') | CORRECT = ('a' == 'a'); |

### d) Observations :

Les opérateurs et les fonctions arithmétiques utilisées dans les expressions seront introduites dans la suite du chapitre. Observons déjà que :

* il n'existe pas de fonction standard en C pour calculer le carré d'une valeur; on peut se référer à la fonction plus générale **pow(x,y)** qui calcule $\mathbf{x^y}$.
* le test d'égalité en C se formule avec *deux* signes d'égalité == , l'affectation avec *un seul* = .

## 2) Les opérateurs connus :

Avant de nous lancer dans les 'spécialités' du langage C, retrouvons d'abord les opérateurs correspondant à ceux que nous connaissons déjà en langage python.

### a) Opérateurs arithmétiques :

| | |
|---|---|
| + | **addition** |
| - | **soustraction** |
| * | **multiplication** |
| / | **division (entière et rationnelle!)** |
| % | **modulo (reste d'une div. entière)** |

### *b)* Opérateurs logiques :

| | |
|---|---|
| && | **et logique (and)** |
| \|\| | **ou logique (or)** |
| ! | **négation logique (not)** |

### c) Opérateurs de comparaison

| | |
|---|---|
| == | **égal à** |
| != | **différent de** |
| <, <=, >, >= | **plus petit que, ...** |

### d) Opérations logiques :

Les résultats des opérations de comparaison et des opérateurs logiques sont du type **int** :

- - la valeur 1 correspond à la valeur booléenne vrai
- - la valeur 0 correspond à la valeur booléenne faux

Les opérateurs logiques considèrent toute valeur différente de zéro comme vrai et zéro comme faux :

| | |
|---|---|
| **32 && 2.3** | **1** |
| **!65.34** | **0** |
| **0\|\|!(32 > 12)** | **0** |

## 3) Les opérateurs particuliers de C :

### a) Les opérateurs d'affectation :

#### *Opération d'affectation :*

En pratique, nous retrouvons souvent des affectations comme :

**i = i + 2**

En C, nous utiliserons plutôt la formulation plus compacte :

**i += 2**

L'opérateur += est un *opérateur d'affectation*.
Pour la plupart des expressions de la forme :

**expr1 = (expr1) op (expr2)**

il existe une formulation équivalente qui utilise un opérateur d'affectation :

**expr1 op= expr2**

***Opérateurs d'affectation***

| | |
|---|---|
| **ajouter à** | **+=** |
| **diminuer de** | **-=** |
| **multiplier par** | ***=** |
| **diviser par** | **/=** |
| **modulo** | **%=** |

## b) Opérateurs d'incrémentation et de décrémentation :

Les affectations les plus fréquentes sont du type :

```
I = I + 1        et        I = I - 1
```

En C, nous disposons de deux opérateurs inhabituels pour ces affectations :

| | | |
|---|---|---|
| **I++ ou ++I** | pour l'incrémentation | (augmentation d'une unité) |
| **I-- ou --I** | pour la décrémentation | (diminution d'une unité) |

Les opérateurs ++ et -- sont employés dans les cas suivants :

incrémenter/décrémenter une variable (par ex : dans une boucle). Dans ce cas il n'y a pas de différence entre la notation ***préfixe*** (**++I --I**) et la notation ***postfixe*** (**I++ I--**).

incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe :

| | |
|---|---|
| **X=I++** | passe d'abord la valeur de I à X et ***incrémente après*** |
| **X=I--** | passe d'abord la valeur de I à X et ***décrémente après*** |
| **X=++I** | ***incrémente d'abord*** et passe la valeur incrémentée à X |
| **X=--I** | ***décrémente d'abord*** et passe la valeur décrémentée à X |

***Exemple :***

Supposons que la valeur de N est égal à 5 :

| | | |
|---|---|---|
| Incrém. postfixe : | **X = N++;** | Résultat : N=6 et X=5 |
| Incrém. préfixe : | **X = ++N;** | Résultat : N=6 et X=6 |

## IV) Les expressions et les instructions :

### 1) Expressions :

La formation des expressions est définie par récurrence :

Les *constantes* et les *variables* sont des expressions.

Les expressions peuvent être combinées entre elles par des *opérateurs* et former ainsi des expressions plus complexes.

Les expressions peuvent contenir des appels de fonctions et elles peuvent apparaître comme paramètres dans des appels de *fonctions.*

***Exemples :***

```
i =0
i++
X=pow(A,4)
printf(" Bonjour !\n")
a=(5*x+10*y)*2
(a+b)>=100
position!=limite
```

### 2) Instructions :

Une expression comme **I=0** ou **I++** ou **printf(...)** devient une ***instruction***, si elle est suivie d'un point-virgule.

***Exemples :***

```
i=0;
i++;
X=pow(A,4);
printf(" Bonjour !\n");
a=(5*x+10*y)*2;
```

### 3) Évaluation et résultats :

En C toutes les expressions sont évaluées et retournent une valeur comme résultat :

| | |
|---|---|
| **(3+2==5)** | retourne la valeur 1 (vrai) |
| **A=5+3** | retourne la valeur 8 |

Comme les affectations sont aussi interprétées comme des expressions, il est possible de profiter de la valeur rendue par l'affectation :

```
((A=sin(X)) == 0.5)
```

## V) Les priorités des opérateurs :

L'ordre de l'évaluation des différentes parties d'une expression correspond en principe à celle que nous connaissons des mathématiques.

***Exemple :***

Supposons pour l'instruction suivante : A=5, B=10, C=1

```
X = 2*A+3*B+4*C;
```

L'ordinateur évalue d'abord les multiplications :

```
2*A ==> 10, 3*B ==> 30, 4*C ==> 4
```

Ensuite, il fait l'addition des trois résultats obtenus :

```
10+30+4 ==> 44
```

A la fin, il affecte le résultat général à la variable :

```
X=44
```

## 1) Priorité d'un opérateur :

On dit alors que la multiplication ***a la priorité*** sur l'addition et que la multiplication et l'addition ont la priorité sur l'affectation.

Si nous voulons forcer l'ordinateur à commencer par un opérateur avec une priorité plus faible, nous devons (comme en mathématiques) entourer le terme en question par des *parenthèses.*

***Exemple :***

Dans l'instruction :

```
X = 2*(A+3)*B+4*C;
```

L'ordinateur évalue d'abord l'expression entre parenthèses, ensuite les multiplications, ensuite l'addition et enfin l'affectation.

Entre les opérateurs que nous connaissons jusqu'ici, nous pouvons distinguer les classes de priorités suivantes :

## 2) Classes de priorités :

| | |
|---|---|
| Priorité 1(la plus forte) : | () |
| Priorité 2 | **! ++ --** |
| Priorité 3 | *** / %** |
| Priorité 4 | **+ -** |
| Priorité 5 | **< <= > >=** |
| Priorité 6 | **== !=** |
| Priorité 7 | **&&** |
| Priorité 8 | **\|\|** |
| Priorité 9 | **=+ =- =* =/ =% =** |

## 3) Evaluation d'opérateurs de la même classe :

--> Dans chaque classe de priorité, les opérateurs ont la même priorité. Si nous avons une suite d'opérateurs binaires de la même classe, l'évaluation se fait en passant *de la gauche vers la droite* dans l'expression.

<-- Pour les opérateurs unaires (**!,++,--**) et pour les opérateurs d'affectation (**=,+=,-=,*=,/=,%=**), l'évaluation se fait *de droite à gauche* dans l'expression.

***Exemples :***

L'expression 10+20+30-40+50-60 sera évaluée comme suit :

**10+20** ==> 30
30+**30** ==> 60
60-**40** ==> 20
20+**50** ==> 70
70-**60** ==> 10

Pour A=3 et B=4, l'expression **A *= B += 5** sera évaluée comme suit :

Pour A=1 et B=4, l'expression **!--A==++!B** sera évalué comme suit :

***Les parenthèses***

Les parenthèses sont seulement nécessaires si nous devons forcer la priorité, mais elles sont aussi permises si elles ne changent rien à la priorité. En cas de parenthèses imbriquées, l'évaluation se fait de l'intérieur vers l'extérieur.

***Exemple :***

En supposant à nouveau que A=5, B=10, C=1 l'expression suivante s'évaluera à 134 :

```
X = ((2*A+3)*B+4)*C
```

Observez la priorité des opérateurs d'affectation :

```
X*=Y+1    X=X*(Y+1)

X*=Y+1     X=X*Y+1
```

# VI) Les fonctions arithmétiques standard :

Les fonctions suivantes sont prédéfinies dans la bibliothèque standard <*math*>. Pour pouvoir les utiliser, le programme doit contenir la ligne :

```
#include <math.h>
```

## 1) Type des données :

Les arguments et les résultats des fonctions arithmétiques sont du type **double**.

***Fonctions arithmétiques :***

| Fonction C | Explication | Algorithmique |
|---|---|---|
| **exp(X)** | Fonction exponentielle | $e^X$ |
| **log(X)** | Logarithme naturel | **ln(X)**, X>0 |
| **log10(X)** | Logarithme à base 10 | $\log_{10}(X)$, X>0 |
| **pow(X,Y)** | X exposant Y | $X^Y$ |
| **sqrt(X)** | Racine carrée de X | pour X>0 |
| **fabs(X)** | Valeur absolue de X | \|X\| |

| | |
|---|---|
| **sin(X), cos(X), tan(X)** | **sinus, cosinus, tangente de X** |
| **asin(X), acos(X), atan(X)** | **arcsin(X), arccos(X), arctan(X)** |
| **sinh(X), cosh(X), tanh(X)** | **sinus, cosinus, tangente hyperboliques de X** |

***Remarque avancée :***

La liste des fonctions ne cite que les fonctions les plus courantes. Pour la liste complète et les constantes prédéfinies voir **<math.h>**.

# VII) Les conversions de type :

La grande souplesse du langage C permet de mélanger des données de différents types dans une expression. Avant de pouvoir calculer, les données doivent être converties dans un même type. La plupart de ces conversions se passent automatiquement, sans l'intervention du programmeur, qui doit quand même prévoir leur effet. Parfois il est nécessaire de convertir une donnée dans un type différent de celui que choisirait la conversion automatique; dans ce cas, nous devons forcer la conversion à l'aide d'un opérateur spécial ("cast").

## 1) Les conversions de type automatiques :

### a) Calculs et affectations :

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun. Ces manipulations implicites convertissent en général des types plus 'petits' en des types plus 'larges'; de cette façon on ne perd pas en précision.

Lors d'une affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité. Dans ce cas, il peut y avoir perte de précision si le type de la destination est plus faible que celui de la source.

***Exemple :***

Considérons le calcul suivant :

```
int I = 8;
float X = 12.5;
double Y;
Y=I*X;
```

Pour pouvoir être multiplié avec *X*, la valeur de *I* est convertie en **float** (le type le plus large des deux). Le résultat de la multiplication est du type **float**, mais avant d'être affecté à *Y*, il est converti en **double**. Nous obtenons comme résultat :

```
Y = 100.00
```

### b) Appels de fonctions :

Lors de l'appel d'une fonction, les paramètres sont automatiquement convertis dans les types déclarés dans la définition de la fonction.

***Exemple :***

Au cours des expressions suivantes, nous assistons à trois conversions automatiques :

```
int A = 200;
int RES;
RES = pow(A, 2);
```

A l'appel de la fonction **pow**, la valeur de A et la constante 2 sont converties en **double**, parce que **pow** est définie pour des données de ce type. Le résultat (type **double**) retourné par **pow** doit être converti en **int** avant d'être affecté à RES.

### c) Règles de conversion automatique :

Conversions automatiques lors d'une opération avec,

- (1) deux entiers :
  D'abord, les types **char** et **short** sont convertis en **int**. Ensuite, l'ordinateur choisit le plus large des deux types dans l'échelle suivante :

  **int, unsigned int, long, unsigned long**

- (2) un entier et un rationnel :
  Le type entier est converti dans le type du rationnel.

- (3) deux rationnels :
  L'ordinateur choisit le plus large des deux types selon l'échelle suivante :

  **float, double, long double**

- (4) affectations et opérateurs d'affectation :
  Lors d'une affectation, le résultat est toujours converti dans le type de la destination. Si ce type est plus faible, il peut y avoir une perte de précision.

***Exemple :***

Observons les conversions nécessaires lors d'une simple division :

```
int X;
float A=12.48;
char B=4;
X=A/B;
```

*B* est converti en **float** (règle 2). Le résultat de la division est du type **float** et sera converti en **int** avant d'être affecté à *X* (règle 4), ce qui conduit au résultat $X=3$ .

### d) Phénomènes imprévus ... :

Le mélange de différents types numériques dans un calcul peut inciter à ne pas tenir compte des phénomènes de conversion et conduit parfois à des résultats imprévus ...

***Exemple :***

Dans cet exemple, nous divisons 3 par 4 à trois reprises et nous observons que le résultat ne dépend pas seulement du type de la destination, mais aussi du type des opérandes.

```
char A=3;
int B=4;
float C=4;
float D,E;
char F;
D = A/C;
E = A/B;
F = A/C;
```

* Pour le calcul de *D*, *A* est converti en **float** (règle 2) et divisé par *C*. Le résultat (0.75) est affecté à *D* qui est aussi du type **float**. On obtient donc : *D=0.75*.
* Pour le calcul de *E*, *A* est converti en **int** (règle 1) et divisé par *B*. Le résultat de la division (type **int,** valeur 0) est converti en **float** (règle 4). On obtient donc : *E=0.000*
* Pour le calcul de *F*, *A* est converti en **float** (règle 2) et divisé par *C*. Le résultat (0.75) est retraduit en **char** (règle 4). On obtient donc : *F=0*

### e) Perte de précision :

Lorsque nous convertissons une valeur en un type qui n'est pas assez précis ou pas assez grand, la valeur est coupée sans arrondir et sans nous avertir ...

***Exemple :***

```
unsigned int A = 70000;
/* la valeur de A sera : 70000 mod 65536 = 4464 */
```

## 2) Les conversions de type forcées (casting) :

Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation à l'aide de la syntaxe :

***Casting (conversion de type forcée)*** :

```
(<Type>) <Expression>
```

***Exemple :***

Nous divisons deux variables du type entier. Pour avoir plus de précision, nous voulons avoir un résultat de type rationnel. Pour ce faire, nous convertissons l'une des deux opérandes en **float**. Automatiquement C convertira l'autre opérande en **float** et effectuera une division rationnelle :

```
char A=3;
int B=4;
float C;
C = (float)A/B;
```

La valeur de *A* est explicitement convertie en **float**. La valeur de *B* est automatiquement convertie en **float** (règle 2). Le résultat de la division (type rationnel, valeur 0.75) est affecté à *C*.

Résultat : *C=0.75*

***Attention ! :***

Les contenus de A et de B restent inchangés; seulement les valeurs utilisées dans les calculs sont converties !