4. Variables, constantes, types, opérateurs

1 Variables, constantes, types

1.1 Variables et constantes

Les variables et les constantes peuvent être considérées comme des registres de mémoire pour lesquels l'utilisateur choisit un nom. Ce nom doit être constitué de caractères dits « alphanumériques » c'est à dire les vingt-six lettres de l'alphabet, les dix chiffres et le caractère souligné. Les majuscules sont autorisées et sont distinguées des minuscules : `x` et `X` ne représentent pas la même variable. Un nom ne peut pas commencer par un chiffre et peut comporter au plus trente et un caractères.
Chaque variable ou constante a un type, défini par une déclaration. Dans ce cours on se limite aux trois types principaux :

- entier, déclaré par `int x`
- réel, déclaré par `double x`
- caractère, déclaré par `char x`

Sur les PC utilisés en TD le type `int` est écrit sur 4 octets, le type `double` sur 8, le type `char` sur 1.
Pour indiquer qu'un nom désigne une constante et non pas une variable, on ajoute le mot clé `const` dans la déclaration de `x` et on lui attribue sa valeur [1] : `const int x = 2`

Remarque

Il faut savoir qu'il existe aussi des entiers sur 1 octet, sur 2 octets, avec ou sans signe, des réels sur 4 octets (type `float`), même si nous ne nous servirons pas de ces types.

1.2 Constantes explicites

Ce sont des nombres ou des caractères.

un nombre entier se note par exemple : `123`
un nombre réel de type `double` : `1.23` ou `0.123e1` ou `12.3e-1` etc. pour le même nombre réel 1.23
un caractère : `'a'` (entre « simples quotes ») [2]

Pour les réels :

`1.6e-19` est donc la notation informatique de $1.6 \cdot 10^{-19}$
on met un point même s'il s'agit d'un nombre n'ayant que des chiffres décimaux nuls, pour bien distinguer entre entiers et réels. On écrit le réel 123 : `123.`

2 Attribution de valeur : le signe =

L'instruction `y = x` a un sens différent de celui qu'a en mathématiques la proposition $y = x$. Elle signifie qu'on met dans le registre de mémoire situé à gauche du signe = (désigné ici par `y`) le contenu qui se trouve dans le registre de mémoire situé à droite du signe = (ici désigné par `x`) à l'instant où l'instruction est exécutée. Par exemple :

`x = 1` `x` vaut 1
`y = x` `x` vaut 1 `y` vaut 1
`x = 2` `x` vaut 2 `y` vaut 1

On peut noter aussi sur l'exemple suivant qu'en C le signe = n'a pas non plus exactement la même signification que dans un langage comme Maple :

1. Le compilateur C ignore des espaces supplémentaires : pour lui `x=2` et `x = 2` sont équivalents. Dans ce poly on préfère mettre des espaces autour des signes = etc. pour améliorer la lisibilité des programmes.
2. On verra dans la suite que les « chaînes de caractères » se notent entre « doubles quotes » : `"abc"`

	C	Maple

`x=1`
`y=x`
`x=2`
	`x` vaut 1
	`x` et `y` valent 1
	`x` vaut 2, `y` vaut 1
		idem
		idem
		idem

`y=x`
`x=1`
`x=2`
	`x` et `y` ont une même valeur bien déterminée, mais inconnue
	`x` vaut 1, `y` garde sa valeur inconnue
	`x` vaut 2, `y` garde sa valeur inconnue
		`x` et `y` n'ont pas de valeur bien déterminée et sont égaux
		`x` vaut `1`, `y` vaut `1`
		`x` vaut `2`, `y` vaut `2`

Dans le second tableau on suppose qu'aucune valeur n'a été attribuée à `x` précédemment.
En C comme en Maple la règle est que `y` vaut ce que vaut `x` au moment où on écrit `y=x`. La différence est qu'en Maple une variable peut très bien ne pas avoir de valeur bien définie (comme en mathématiques) alors qu'en C une variable vaut toujours le contenu de la mémoire qu'elle désigne et donc a nécessairement une valeur bien particulière même si celle-ci n'a aucun sens.

Remarque

D'après ce qui précède une constante ne pourra pas figurer à gauche d'un signe = hormis dans sa déclaration.

3 Opérateurs

De façon générale, un opérateur est une règle qui fait correspondre à n éléments $x_1, x_2, ..., x_n$ donnés un élément y :

$$x_1, x_2, ..., x_n \longrightarrow y$$

Si $n = 1$ l'opérateur est dit « unaire »
Si $n = 2$ il est dit « binaire »

Exemple d'opérateur unaire :

$$x \longrightarrow y = -x$$

Exemples d'opérateurs binaires :

$$\begin{array}{lcll} x_1, x_2 & \longrightarrow & y = x_1 x_2 & \text{multiplication} \\ x_1, x_2 & \longrightarrow & y = 0 \text{ si } x_1 \neq x_2 & \text{comparaison} \\ & & y = 1 \text{ si } x_1 = x_2 & \end{array}$$

3.1 Opérateurs arithmétiques, règles de priorité

Les opérateurs arithmétiques sont présentés dans le tableau suivant :

	Signification mathématique	
Opérateur	Entiers	Réels
+	addition	addition
-	soustraction	soustraction
*	multiplication	multiplication
/	partie entière de la division	division
%	reste de la division	non défini

Exemple

2/3 vaut 0
2./3. vaut 0.333333
2%3 vaut 2

Dans une suite d'opérations (formule) il y a des priorités :
- les calculs effectués en premier sont ceux placés entre parenthèses, en commençant par les parenthèses les plus internes.
- ensuite sont effectués : * ou / ou % puis + ou -

* / et % ont la même priorité
\+ et - ont la même priorité

A priorité égale les opérations sont effectuées de la gauche vers la droite.

Exemple

$\frac{a+b}{c+d}$ sera écrit : `(a+b)/(c+d)`

alors que :

`a+b/c+d` donne : $a+\frac{b}{c}+d$

`(a+b)/c+d` donne : $\frac{a+b}{c}+d$

et :

`a+b/(c+d)` donne : $a+\frac{b}{c+d}$

Par ailleurs :

$\frac{a}{bc}$ peut s'écrire indifféremment : `a/(b*c)` ou `a/b/c`

Exercice : exemple d'utilisation de l'opérateur %

On convient que :

Lundi=1, Mardi=2, ..., Dimanche=7

On veut déterminer quel jour j de la semaine est le $n^{\text{ème}}$ jour de l'année, sachant que le premier Janvier est le jour p de la semaine. On fournit donc une valeur de n comprise entre 1 et 365 (ou 366) et une valeur de p comprise entre 1 et 7 et on cherche la valeur de j, comprise entre 1 et 7. Ecrire une expression composée de `n` et `p` et dont le résultat est `j`.

Réponse : `j=(n+p-2)%7+1`

Si l'on veut utiliser des opérateurs un peu moins élémentaires il est nécessaire d'utiliser la bibliothèque mathématique du C en incluant la directive :

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

Pour les entiers on dispose alors de la valeur absolue de n avec `abs(n)`.
Pour les réels les fonctions les plus courantes sont présentées dans le tableau ci-dessous :

Notation mathématique	x^y	$\sqrt{x}$	$\exp x$	$\ln x$	$\log x$	$\lvert x\rvert$	$\sin x$	$\cos x$	$\tan x$
Notation C	`pow(x,y)`	`sqrt(x)`	`exp(x)`	`log(x)`	`log10(x)`	`fabs(x)`	`sin(x)`	`cos(x)`	`tan(x)`

Toutes ces fonctions ont des valeurs et des arguments de type `double`.

Remarques

Les opérations non définies telles que division par 0, racine carrée d'un nombre négatif, élévation d'un nombre négatif à une puissance réelle, etc., donnent lieu à un diagnostic en cours d'exécution.

Si une opération aboutit à un dépassement de capacité :
— pour les entiers il n'y a pas de diagnostic et le résultat est complètement aberrant
— pour les réels il y a un diagnostic mais le calcul se poursuit et le résultat peut être complètement aberrant.

L'instruction `a += b` est équivalente à `a = a+b`, l'instruction `i++` est équivalente à `i = i+1`.
Il existe de même `a -= b`, `a *= b`, `a /= b`, `a %= b`[3].

3.2 Mélange des types entier et réel

3.2.1 Conversion de type dans une affectation

Lors d'une affectation :

```
y = x;
```

la valeur de `x` est convertie dans le type de `y` avant d'être mise dans `y`. Supposons que `x` soit un `double` : si `y` est un `double` le résultat final sera bien un `double` mais si `y` est un `int` le résultat final sera un `int` et il y aura donc troncature à l'entier de valeur absolue inférieure ou égale.

3.2.2 Mélange de types dans une expression

On peut dans une expression mélanger les types entier et réel. Mais il faut l'éviter le plus possible.
Quand le calcul peut être effectué uniquement avec des entiers, il ne faut pas introduire inutilement de réels : les opérations sont plus rapides avec les entiers et ne sont pas affectées d'erreur de troncature (sauf cas de la division).
Quand le calcul comprend nécessairement à la fois des entiers et des réels, il faut tenir compte de la règle suivante :

lors d'une opération (+ - * /) entre un entier et un réel, l'entier est converti automatiquement en réel avant l'opération et le résultat est toujours réel (on dit que le réel l'emporte toujours).

Il faut aussi tenir compte de la conversion lors de l'affectation en examinant à quel type de variable ce résultat est affecté.

Exemple

3. Attention, il peut y avoir des cas à traiter avec précaution : par exemple `a`, `b` et `c` étant des entiers, `a *= b/c` est équivalent à `a = a*(b/c)`, ce qui donne un résultat différent de `a = a*b/c` si `b` n'est pas divisible par `c`.

```
#include<iostream>
using namespace std;
int main() {
  int i, j = 2, k = 3;
  double x, y = 2., z = 3.;
  i = j/k; cout << "Resultat : " << i << endl;
  x = y/z; cout << "Resultat : " << x << endl;
  x = j/z; cout << "Resultat : " << x << endl;
  x = y/k; cout << "Resultat : " << x << endl;
  i = j/z; cout << "Resultat : " << i << endl;
  i = y/k; cout << "Resultat : " << i << endl;
  x = j/k; cout << "Resultat : " << x << endl;
  i = y/z; cout << "Resultat : " << i << endl;
  return 0;
}
```

donne :

Resultat : 0
Resultat : 0.666667
Resultat : 0.666667
Resultat : 0.666667
Resultat : 0
Resultat : 0
Resultat : 0
Resultat : 0

Les fonctions mathématiques indiquées ci-dessus [4], réalisent la conversion automatique du type des arguments vers le type `double`. Ainsi :

```
...
int i; double y;
...
... = sqrt(i);
... = sqrt(y);
...
```

`i` est converti en `double` avant le calcul de la racine carrée, `y` est pris tel quel, et le résultat est `double`.

Remarque (à sauter en première lecture)

En mathématiques, x^y n'est défini pour x négatif que si y est un nombre rationnel de la forme p/q avec p et q entiers premiers entre eux et q impair. Si x est négatif x^y est donc défini pour y entier (positif ou négatif) et pour $y = 2/3$ par exemple, mais pas pour $y = 3/2$. En C les arguments de `pow(x,y)` sont toujours réels [5] au sens informatique et on peut se demander ce qu'il advient lorsque x est négatif. Si y est entier au sens mathématique, c'est à dire vaut 1.000..., 2.000..., (ou -1.000..., -2.000...) etc. la fonction `pow` est capable de calculer `pow(x,y)`, mais dans les autres cas non. Elle ne peut reconnaître que y est un rationnel du type précédent, ce qui n'est pas étonnant. Elle refuse de calculer $(-1.2)^{3/2}$, ce qui est normal puisque cette quantité n'est pas définie, mais elle refuse aussi de calculer $(-1.2)^{2/3}$ qui est pourtant bien définie. Ceci est illustré par le programme suivant :

```
#include<iostream>
#include <math.h>
using namespace std;
int main() {
  int i, j;
  double x, y, z;
  /* pow(-4.5,2./3.) ne peut etre calcule : */
  x = 4.5; y = 2./3.; z = 3./2.;
```

4. comme les autres fonctions qui seront étudiées ultérieurement
5. éventuellement après une conversion

```
  cout << endl;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y)
       << "   pow(" << x << "," << z << ")=" << pow(x,z) << endl;
  x = -4.5;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y)
       << "   pow(" << x << "," << z << ")=" << pow(x,z) << endl << endl;
  /* mais pow(-4.5,3.) est correctement calcule : */
  x = 4.5; y = 3.;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y) << endl;
  x = -4.5;
  cout << "pow(" << x << "," << y << ")=" << pow(x,y) << endl << endl;
  /* et la conversion entier-reel se fait bien : */
  i = 7; j = 3;
  cout << "pow(" << i << "," << j << ")=" << pow(i,j) << endl;
  i = -7;
  cout << "pow(" << i << "," << j << ")=" << pow(i,j) << endl << endl;
  return 0;
}
```

qui donne le résultat :

pow(4.5,0.666667)=2.72568 pow(4.5,1.5)=9.54594
pow(-4.5,0.666667)=nan pow(-4.5,1.5)=nan
pow(4.5,3)=91.125
pow(-4.5,3)=-91.125
pow(7,3)=343
pow(-7,3)=-343

3.2.3 Opérateur de conversion de type (...)

Soit deux variables :

```
int i; double x;
```

l'expression `(double)i` représente la valeur de `i` transformée en `double`
l'expression `(int)x` représente la valeur de `x` transformée en `int`

Ces expressions ne sont pas des variables, elles ne peuvent pas figurer à gauche d'un signe `=`.
De façon plus générale l'expression `(t)y` représente la valeur de `y` transformée dans le type `t`.
`(t)` s'appelle l'opérateur de conversion de type vers le type `t` (opérateur « cast » en anglais).
Ainsi :

```
...
int i, j;
i = 2; j = 3;
cout << i/j << " " << (double)i/j << endl;
...
```

donne : *0 0.666667*

3.3 Exemple

Formule de Planck :

$$u(\nu) = \frac{8\pi h\nu^3}{c^3} \frac{1}{e^{\frac{h\nu}{kT}} - 1}$$

peut par exemple s'écrire :

```
#include<iostream>
#include<iomanip>
#include<math.h>
using namespace std;
int main() {
  const double h = 6.62618e-34, k = 1.38066e-23, c = 2.99792e8;
  double b, nu, t, u;
  nu = 1.e15; t = 5.e3;
  b = h*nu/k/t;
  u = 8.*M_PI*h*pow(nu/c,3)/(exp(b)-1.);
  cout << "Resultat : " << u << endl;
  return 0;
}
```

`M_PI` est une constante définie dans `math.h` qui donne la valeur de π.
Plus il y a de niveaux de parenthèses, plus, en général, la formule est difficile à lire. Il faut donc éviter de mettre des parenthèses inutiles.
Cependant dans quelques cas une parenthèse inutile rend au contraire la formule plus facile à lire.
Il faut aussi utiliser des variables intermédiaires, surtout si des sous-formules identiques se répètent dans la formule. Il ne faut pas non plus pulvériser la formule en multipliant les variables intermédiaires, car la formule s'étend alors en hauteur au lieu de s'étendre en largeur. Il n'y a pas de règle stricte, c'est une question d'appréciation personnelle du programmeur, l'objectif étant toujours la lisibilité, pour gagner du temps de lecture et diminuer le risque d'erreur.

3.4 Opérateurs de comparaison

Ce sont `<` `<=` `>` `>=` `==` `!=`, la signification des quatre premiers est explicite, les deux derniers désignant respectivement « égal à » et « différent de » . L'expression `x > y` a la valeur entière 1 si elle est vraie et 0 si elle est fausse, et de même pour les autres opérateurs de comparaison.

Exemple

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y;
  x = 1.; y = 2.;
  i = x>y; cout << "i=" << i << endl;
  i = x<y; cout << "i=" << i << endl;
  return 0;
}
```

Affiche à l'écran le résultat :

i=0
i=1

Remarque

Pour les caractères la comparaison teste l'ordre alphabétique (ne fonctionne pas pour les chaînes de caractères).

Remarque

Les tests entre réels souffrent de l'imprécision attachée à la représentation des réels. Par exemple, sur les PC utilisés en TD, le programme suivant :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y;
  x = 0.5; y = 0.25;
  i = (x*x == y); cout << i << endl;
```

```
  x = 0.3; y = 0.09;
  i = (x*x == y); cout << i << endl;
  return 0;
}
```

donne comme résultat :

1
0

Il ne faut pas oublier cela quand on fait des comparaisons entre réels.

3.5 Opérateurs logiques

Ce sont `&&`, `||` et `!`, correspondant respectivement à « et » , « ou » et « non » .
Exemple

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, y, z;
  x = 1.; y = 2.; z = 3.;
  i = x<y && y<z; cout << "i=" << i << endl;
  i = x>y || y==z; cout << "i=" << i << endl;
  i = !(x>y || y>z); cout << "i=" << i << endl;
  return 0;
}
```

Donne :

i=1
i=0
i=1

3.6 Priorités des opérateurs arithmétiques, de comparaison et logiques

Dans une expression qui comprend simultanément des opérateurs arithmétiques, de comparaison et logiques, l'ordre de priorité décroissante est : arithmétique, comparaison, logique.
Entre les opérateurs de comparaison `<`, `<=`, `>`, `>=` ont tous la même priorité, et `==` et `!=` la priorité immédiatement inférieure.
`&&` est prioritaire sur `||`.
Les termes entre parenthèses sont évalués en premier en commençant par les parenthèses les plus internes et à priorité égale les opérations sont effectuées dans l'ordre de l'écriture.

Exemple

`n1, n2, n3` étant par exemple des entiers :

`n1+n2 > n3` l'addition est effectuée avant la comparaison

`n1+n2 > n3 || n1-n2 < n3` addition et soustraction sont effectuées d'abord puis les deux comparaisons, puis le ou.
Il peut être utile de mettre des parenthèses même si elles ne sont pas nécessaires, pour rendre l'expression plus lisible :
`((n1+n2) > n3) || ((n1-n2) < n3)`

Liste des opérateurs mentionnés jusqu'ici, par ordre de priorité décroissante d'une ligne à l'autre :

Opérateurs
`()`
`!`
`* / %`
`+ -`

```
<   <=   >   >=
   ==   !=
     &&
     ||
```

Exercice : réduction au musée

Un individu peut obtenir une réduction s'il remplit une des conditions suivantes :
— il a moins de 18 ans
— il a moins de 25 ans et fait des études
— il a moins de 25 ans et ses deux parents ne travaillent pas
On définit les variables de type `int` suivantes :

`age` qui vaut l'âge de l'individu
`etu` qui vaut 1 s'il fait des études, 0 sinon
`pt` qui vaut 1 si le père travaille, 0 sinon
`mt` qui vaut 1 si la mère travaille, 0 sinon

Ecrire une expression composée de ces quatre variables qui vaut 1 s'il a droit à la réduction, 0 sinon.

Réponse : `(((mt¡ && pt¡) || etu) && age<25) || age<18`

Quelles parenthèses peut-on supprimer dans la réponse précédente ?

Réponse : les plus externes et les plus internes