

# **Cours 2**

Compter en binaire  
Variables et types

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Pour des raisons technologiques, l'ordinateur ne peut traiter ou manipuler qu'une information codée sous forme binaire:

- *le courant électrique passe ou ne passe pas;*
- *qu'on peut interpréter en 0 ou 1.*

Au sein de l'ordinateur, toutes les informations sont codées en binaire, c'est-à-dire une succession de 0 et de 1:

les nombres, les lettres, les images, la musique, les programmes, des adresses-mémoire...

# Binaire = base 2

En base 10, les nombres sont représentés avec les symboles

0 1 2 3 4 5 6 7 8 et 9

En base 2, les nombres sont représentés uniquement avec les symboles

0 et 1

**Décimal (base 10)**

**Binaire (base 2)**

0	→	0
1	→	1
2	→	10
3	→	11
4	→	100
5	→	101
6	→	110
7	→	111
8	→	1000
9	→	1001
10	→	1010
11	→	1011

$$\begin{array}{r} 2 \rightarrow 10 \\ + 1 \rightarrow + 1 \\ \hline = 3 \rightarrow = 11 \end{array}$$

$$\begin{array}{r} 3 \rightarrow 11 \\ + 1 \rightarrow + 1 \\ \hline = 4 \rightarrow = 100 \end{array}$$

# Conversion binaire $\rightarrow$ décimal

Un nombre écrit **239** en base 10 (= en décimal) vaut en base 10:

$$2 \times 10^2 + 3 \times 10^1 + 9 \times 10^0 =$$

$$2 \times 100 + 3 \times 10 + 9 \times 1 = 200 + 30 + 9 = 239 \text{ en base 10}$$

Un nombre écrit **101** en base 2 (= en binaire) vaut en base 10:

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$$

$$1 \times 4 + 0 \times 2 + 1 \times 1 = 4 + 0 + 1 = 5$$

# Conversion décimal → binaire

On peut retrouver les chiffres d'un nombre écrit en base 10 en le divisant par 10 autant de fois que nécessaire.

Les restes des divisions successives constituent l'écriture décimale de ce nombre:

$$239 / 10 = 23 \text{ reste } 9$$

$$23 / 10 = 2 \text{ reste } 3$$

$$2 / 10 = 0 \text{ reste } 2$$

On peut procéder de façon équivalente pour déterminer l'écriture binaire d'un nombre:

$$6 / 2 = 3 \text{ reste } 0$$

$$3 / 2 = 1 \text{ reste } 1$$

$$1 / 2 = 0 \text{ reste } 1$$

→ 6 s'écrit 110 en binaire.

# Multiplier et diviser par des puissances de 2

**Multiplier** par 10 un nombre en décimal revient à décaler ses chiffres vers la gauche et ajouter un 0 à droite:

$$239 * 10 = 2390$$

De même, on peut multiplier par 2 un nombre en binaire simplement en décalant ses chiffres vers la gauche, et en ajoutant un 0 à droite:

$$110 (= 6) * 2 = 1100 (=12)$$

(Si on considère la division entière:)

**Diviser** par 10 un nombre en décimal revient à décaler ses chiffres vers la droite:

$$239 / 10 = 23$$

le chiffre le plus à droite étant perdu.

De même, on peut diviser par 2 un nombre en binaire simplement en décalant ses chiffres vers la droite:

$$110 (= 6) / 2 = 11 (= 3)$$

Décaler un nombre binaire d'un cran vers la gauche revient à le multiplier par 2.

Décaler un nombre binaire de  $n$  crans vers la gauche revient à le multiplier par  $2^n$ .

Décaler un nombre binaire d'un cran vers la droite revient à le diviser par 2.

Décaler un nombre binaire de  $n$  crans vers la droite revient à le diviser par  $2^n$ .

# Limites

En informatique, un nombre est stocké en utilisant un nombre fixe de bits.  
Par exemple, les nombres entiers que nous utiliserons seront stockés sur 32 bits.

Ceci limite le nombre de valeurs que peuvent prendre les variables:

En utilisant 8 bits, on peut représenter

$2^8 = 256$  valeurs, de 00000000 à 11111111.

Et peut provoquer des erreurs de calculs:

si on additionne:

$$\begin{array}{r} 1000\ 0000\ (= 128) \\ +\ 1000\ 0000\ (= 128) \\ =\ 1\ 0000\ 0000\ (= 256) \end{array}$$

comme le résultat sera limité à 8 bits, on obtiendra 0 comme résultat de l'addition.

# Comment coder des nombres négatifs ?

Le *binaire complément à 2* permet de coder des nombres négatifs.

En utilisant 8 bits, on peut alors représenter les nombres de -128 à +127:

Conversion décimal → binaire complément à deux:

- si le nombre est positif, conversion identique;
- si le nombre est négatif:
  1. on convertit d'abord sa valeur absolue en binaire;
  2. on change les 0 en 1, les 1 en 0;
  3. on ajoute 1 au résultat.

Par exemple, pour représenter -6 sur 8 bits:

6 s'écrit            00000110

-6 s'écrit donc    11111001 + 1 = 11111010

Intérêt: l'addition s'effectue naturellement. Par exemple, additionnons 6 et -6:

```
    0000 0110 (= 6)
+   1111 1010 (= -6)
=  1 0000 0000 qui vaut 0 puisqu'on se restreint aux 8 premiers bits.
```

# Interprétation

Le nombre binaire

1111 1001

peut donc s'interpréter comme valant:

- soit -6 s'il s'agit d'un nombre binaire en complément à 2;
- soit 249 (= 128 + 64 + 32 + 16 + 8 + 1) s'il s'agit d'un nombre binaire "classique".

Nous verrons qu'il pourrait également s'interpréter comme, par exemple, un caractère.

# Codage de l'information

Avec des 0 et des 1, on peut coder des nombres.

Avec des nombres, on peut coder d'autres informations:

- par exemple la lettre A est codée (par convention) par le nombre 65, la lettre B par le nombre 66...
- de la même façon, on peut coder des *instructions de programme*, la couleur d'un pixel pour les images, du son, etc...

# Unités de mesure de la mémoire

- Un *bit* est une unité élémentaire de codage binaire, et vaut 0 ou 1;
- 8 bits = 1 octet (*byte* en anglais).

Avec 8 bits, on peut coder les valeurs

0000 0000, 0000 0001... jusqu'à 1111 1111

donc 256 valeurs différentes

- 1 Ko = 1024 ( $=2^{10}$ ) octets
- 1 Mo = 1 Méga octet =  $1024 \times 1024$  octets = 1 048 576 octets
- 1 Go = 1 Giga octet =  $1024^3$  octets = 1 073 741 824 octets
- 1 To = 1 Téra octet =  $1024^4$  octets
  
- la mémoire vive d'un ordinateur actuel est de l'ordre de quelques Go ;
- un disque dur actuel contient quelques centaines de Go, jusqu'à 1 ou 2 To;
- 1 minute de MP3  $\approx$  1 Mo.

# Relation entre binaire et compilateur

Dans un programme exécutable, les instructions sont elles aussi codées en binaire. C'est ce qu'on appelle le *langage machine*.

Heureusement, un programmeur n'a pas à écrire ses programmes en langage machine:

*On utilise l'ordinateur lui-même pour convertir un programme écrit en un langage "évolué" (par exemple le langage C) en un programme en langage machine.*

C'est ce que fait le compilateur.

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```

Programme en langage C:  
*Fichier texte compréhensible  
par un programmeur*

Compilateur

```
0001 1000 1001
0100 1010 ...
```

*Fichier compréhensible  
par l'ordinateur*

# A quoi ressemble un programme en langage machine ?

Supposons que l'on souhaite calculer:

$a * b + c$

a, b et c étant des variables.

Pour que l'ordinateur puisse effectuer ce calcul, il faut tout d'abord le décomposer en actions élémentaires:

1. multiplier a par b

2. ajouter c au résultat

Il faut ensuite convertir ces actions en binaire. En supposant que la multiplication soit codée par 0001, l'addition par 0100, et les variables a, b et c soient représentées par 1000, 1001 et 1010, le programme en langage machine serait:

0001 1000 1001

0100 1010

Cette explication est un peu simplifiée mais suffisamment réaliste pour faire comprendre le mécanisme du langage machine.

Supposez maintenant que vous souhaitiez calculer la valeur de :  $(a + b) / (c * d) + 2 * e...$

# Les variables

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char ** argv)
```

```
{
```

```
    float x;
    float x_carre;
```

```
    x = 4;
    x_carre = x * x;
```

```
    cout << "La variable x contient " << x << "." << endl;
```

```
    cout << "Le carre de " << x << " est " << x_carre << "." << endl;
```

```
    return 0;
```

```
}
```

Déclarations de variables:  
Doivent être situées au début du bloc

Affectations

Affichage

# Déclarations de variables

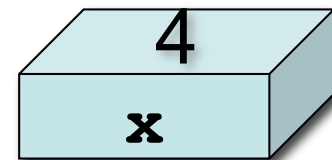
Les lignes:

```
float x;  
float x_carre;
```

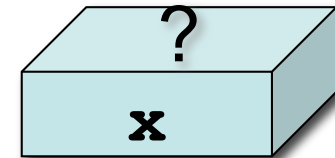
sont des **déclarations de variables**.

Une déclaration de variable permet de créer des **variables** du langage C avec un **nom** choisi par le programmeur.

**Attention:** une variable en informatique est très différente d'une variable en mathématiques. Intuitivement, on peut se représenter une variable informatique comme une « boîte » qui contient une valeur:



La déclaration ne fait que créer la variable, elle ne lui affecte aucune valeur. **Après une déclaration, on ne sait pas ce que contient la variable: il faut toujours initialiser une variable avant d'utiliser sa valeur.**



# Noms de variables

## Règles pour nommer les variables:

- Les noms des variables doivent être formés d'une suite de caractères choisis parmi les lettres et les chiffres;
- Les accents ne sont pas autorisés;
- Le premier caractère est nécessairement une lettre;
- Le caractère souligné \_ (*underscore*) est autorisé et considéré comme une lettre;
- le nom ne doit pas être un mot-clé réservé par le langage C (voir fin du recueil);
- Les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Les noms `ligne` et `Ligne` désignent deux variables différentes.

## Exemples de noms valides:

`x_au_carre`    `Total`    `sousTotal98`

## Exemples de noms invalides:

`x_au_carré`            Contient un accent;  
`x-carre`            Contient le symbole – (moins);  
`1element`            Commence par un chiffre.

# Types de variables

Une variable stocke un certain **type** d'information en mémoire.

Dans la déclaration de la variable `x` de l'exemple:

```
float x;
```

- `x` est le nom de la variable;
- `float` est le type de la variable, c'est-à-dire que `x` est une variable destinée à contenir des nombres **flottants** (des approximations de nombres réels).

Une fois défini, le **type** de la variable ne peut plus changer. Par contre, la **valeur** stockée dans la variable peut être modifiée.

Nous verrons dans la suite d'autres types:

- `double` : pour des approximations plus précises de nombres réels;
- `int` : pour les entiers;
- `unsigned int`: pour les entiers positifs;
- `char`: pour les caractères (A..Z etc...);
- ...

# Affectations

Les lignes:

```
x = 4;
```

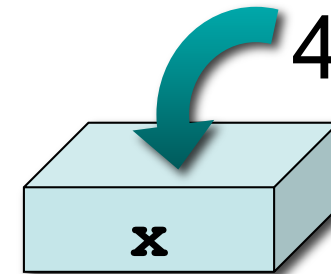
```
x_carre = x * x;
```

sont des **affectations**.

**Attention:** une affectation *ressemble* à une équation mathématique, mais est très différente.

Une affectation est une instruction permettant de mettre une valeur dans une variable.

```
x = 4;
```



L'exécution d'une affectation se décompose en deux temps :

- l'expression à droite du signe = est évaluée
- le résultat est stocké dans la variable à gauche du signe =
- **L'ancienne valeur est perdue.**

Dans le cas de notre exemple `x_carre = x * x;`

- l'étoile \* représente la multiplication;
- `x * x` est évaluée avec la valeur de `x` au moment de l'exécution.
- le résultat est stocké dans la variable `x_carre`.

# Affectations

De façon plus générale, une affectation suit le schéma:

*nom\_de\_variable = expression;*

Exemples d'expression:

- 4
- $x + 1$
- $x * (x + 1)$

une expression calcule une valeur, qui doit être de même type que la variable.

# Précautions (1)

Le mécanisme de l'instruction d'affectation est simple. Attention néanmoins de ne pas la confondre avec une égalité mathématique, même si les deux utilisent le signe égal =

- Les deux instructions:

$$a = b;$$

$$b = a;$$

ne sont pas identiques. La première place dans la variable  $a$  la valeur de la variable  $b$ , tandis que la seconde place dans  $b$  la valeur de  $a$ .

- En mathématiques:

$$b = a + 1$$

signifie que tout au long des calculs,  $a$  et  $b$  vérifieront toujours cette relation. Autrement dit, quel que soit  $a$ ,  $b$  sera toujours égal à  $a+1$

En informatique, si on écrit:

$$a = 5;$$

$$b = a + 1;$$

$$a = 2;$$

la deuxième instruction donne à  $b$  la valeur de  $a+1$ . En revanche, la troisième donne à  $a$  la valeur 2 sans que la valeur de  $b$  ne soit changée.

# Précautions (2)

- L'instruction:

```
a = a + 1;
```

signifie : calculer l'expression de  $a + 1$  et ranger le résultat dans  $a$ . Cela revient à augmenter de 1 la valeur de  $a$ .

(A noter que ce type d'affectation où une variable apparaît de chaque côté du signe = permet de résoudre de nombreux problèmes et sera souvent utilisé par la suite).

- En C et C++, comme dans beaucoup d'autres langages:

```
a + 5 = b;
```

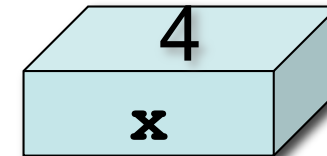
**n'a pas de sens**. On ne peut attribuer de valeur à une expression, mais seulement à une variable.

# Pour écrire à l'écran la valeur d'une variable

```
cout << "La variable x contient " << x << "." << endl;
```

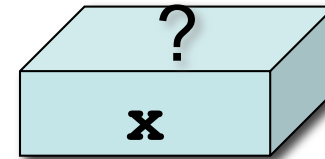
affiche:

```
La variable x contient 4.
```



# Déroulement du programme pas-à-pas

```
float x;  
float x_carre;  
x = 4;  
x_carre = x * x;
```



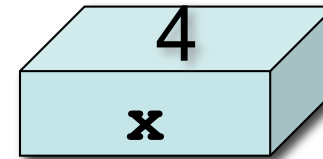
```
cout << "La variable x contient " << x << "." << endl;  
cout << "Le carre de " << x << " est " << x_carre << "." << endl;
```

Ce qui s'affiche dans la fenêtre Terminal:



# Déroulement du programme pas-à-pas

```
float x;  
float x_carre;  
  
x = 4;  
x_carre = x * x;
```



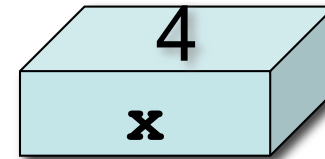
```
cout << "La variable x contient " << x << "." << endl;  
cout << "Le carre de " << x << " est " << x_carre << "." << endl;
```


Ce qui s'affiche dans la fenêtre Terminal:



# Déroulement du programme pas-à-pas

```
float x;  
float x_carre;  
  
x = 4;  
x_carre = x * x;
```



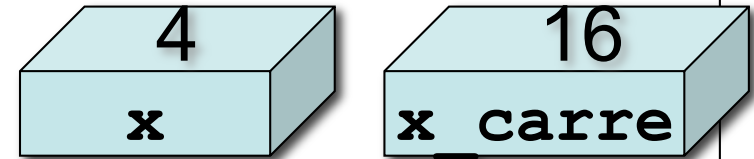
 `cout << "La variable x contient " << x << "." << endl;`  
`cout << "Le carre de " << x << " est " << x_carre << "." << endl;`

Ce qui s'affiche dans la fenêtre Terminal:



# Déroulement du programme pas-à-pas

```
float x;  
float x_carre;  
  
x = 4;  
x_carre = x * x;
```



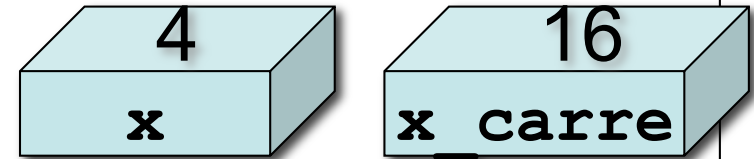
```
cout << "La variable x contient " << x << "." << endl;  
cout << "Le carre de " << x << " est " << x_carre << "." << endl;
```

Ce qui s'affiche dans la fenêtre Terminal:

```
La variable x contient 4.  
|
```

# Déroulement du programme pas-à-pas

```
float x;  
float x_carre;  
  
x = 4;  
x_carre = x * x;
```



```
cout << "La variable x contient " << x << "." << endl;  
cout << "Le carre de " << x << " est " << x_carre << "." << endl;
```

Ce qui s'affiche dans la fenêtre Terminal:

```
La variable x contient 4.  
Le carre de 4 est 16.  
|
```

# Qu'affichent les 3 premiers programmes ?

## PROGRAMME 1:

```
float a, b;  
  
a = 2;  
b = 1;  
b = a * (a + 1) * (b + 2);  
  
cout << a << ", " << b << endl;
```

## PROGRAMME 2:

```
float a, b;  
  
a = 5;  
b = a + 3;  
a = 1;  
  
cout << a << ", " << b << endl;
```

## PROGRAMME 3:

```
float a, b;  
  
a = 1;  
b = 2;  
a = b;  
b = a;  
  
cout << a << ", " << b << endl;
```

Comment inverser la valeur de deux variables a et b ?

# Les types de base du langage C

# Notion de type d'une variable

Les variables servent à conserver différentes sortes d'informations: nombres entiers, nombres réels, caractères...

Chaque sorte d'information devra disposer d'un codage approprié et on parlera de *type* d'une variable.

En C, le programmeur doit préciser le type des variables: c'est l'utilité de la déclaration des variables.

# Déclaration de variables

Dans un programme C ou C++, on définit le type des variables à l'aide d'instructions de déclaration. Exemples:

```
float x;  
float valeur, res, x1;  
float x2 = 10;  
int n = 1, p;
```

(int: nombres entiers; float: approximation de nombres réels)

On peut donc:

- déclarer plusieurs variables simultanément;
- initialiser la variable au moment de la déclaration.

Les instructions de déclaration doivent précéder les instructions exécutables.

# Types de variables contenant des nombres entiers

- `int`: entiers relatifs codés sur 32 bits
- `short`: entiers relatifs codés sur 16 bits
- `unsigned int`: entiers positifs codés sur 32 bits
- `unsigned short`: entiers positifs codés sur 16 bits
  
- Une variable de type `short` peut donc prendre  $2^{16} = 65536$  valeurs possibles:
- une variable de type `unsigned short` peut prendre les valeurs de 0 à 65535
- une variable de type `short` peut prendre les valeurs de -32768 à +32767
- Une variable de type `int` peut prendre les valeurs de  $-2^{31}$  à  $+2^{31}-1$  ( $\approx$  de -2 milliards à 2 milliards).

→ Un type en informatique est limité, contrairement en mathématiques.

En pratique, pour les exercices, on utilisera principalement le type `int` pour les variables contenant des nombres entiers.

# Les opérateurs relatifs au type `int`

# Opérateurs relatifs au type `int`

*On dispose des 4 opérateurs usuels:*

- + pour l'addition;
- pour la soustraction;
- \* pour la multiplication;
- / pour la division.

**Attention**, il s'agit de la division entière:

`1 / 2` donne `0` et non pas `0.5`

# Opérateurs relatifs au type `int`

*De plus, il existe:*

- un opérateur modulo, noté `%`, qui renvoie le reste de la division entière:  
`11 % 4` vaut 3 (la division de 11 par 4 a pour reste 3).
- un opérateur opposé, noté également `-`. Cet opérateur ne porte que sur un seul terme, par exemple: `-n`
- deux opérateurs d'incrément et de décrémentation, c'est-à-dire consistant à ajouter ou à soustraire 1 à une variable.  
Ils sont notés respectivement `++` et `--`

Ainsi, l'instruction:

```
i++;
```

est équivalente à :

```
i = i + 1;
```

Ces deux opérateurs (`++` et `--`) sont souvent utilisés avec l'instruction `for`, que nous verrons par la suite.

# Opérateurs relatifs au type `int`

- les opérateurs `+=` , `-=` , `*=` , `/=`

Par exemple:

```
a += 5;
```

est équivalent à

```
a = a + 5;
```

```
b *= a;
```

est équivalent à

```
b = b * a;
```

# Lire la valeur d'une variable

Supposons qu'on ait déclaré la variable `n` de type `int`:

```
int n;
```

L'instruction:

```
cin >> n;
```

lit la valeur que tape l'utilisateur au clavier et stocke cette valeur dans la variable `n`.

# Exemple

```
#include <iostream.h>
using namespace std;

int main(int argc, char ** argv)
{
    int n;

    cout << "Donnez un nombre entier:" << endl;
    cin >> n;

    cout << "Vous avez entre: " << n << endl;

    cout << "Au revoir" << endl;

    return 0;
}
```

**Les types `float` et `double`**  
**Leurs opérateurs**  
**Les fonctions mathématiques**

# Les types `float` et `double`

Les types `float` et `double` permettent de représenter, de manière approchée, des nombres réels.

- Le type `float` est codé en utilisant 32 bits
- Le type `double` est codé en utilisant 64 bits.

→ le type `double` est plus précis mais occupe plus de place en mémoire.

*Comment coder des nombres réels en binaire ?* En les décomposant sous la forme:

*mantisse*  $10^{\text{exposant}}$

Par exemple:  $2.65 = 265 \cdot 10^{-2}$  la mantisse vaut 265, l'exposant 2.

La *mantisse* et l'*exposant* sont des nombres entiers qui peuvent alors être codés directement en binaire.

# Opérateurs relatifs aux types `float` et `double`

Mêmes opérateurs que pour le type `int` (`+` `-` `*` `/` `+=` `--` `*=` `/=`),  
MAIS:

- les opérateurs `%`, `<<` et `>>` n'existent pas;
- la division `/` est la division entre nombres décimaux:

```
float x = 1.5, y = 2.0;
```

```
float r = x / y;
```

→ `r` contient 0.75

# Expressions mixtes

Il est possible d'utiliser des variables de type `int` et des variables de type `float` ou `double` dans une même expression. On parle alors d'expression mixte.

Par exemple:

```
int n;  
float x, y;
```

```
n = 2;  
x = 12.5;  
y = x / n;
```

→ *y* vaut *6.25* : Il s'agit ici d'une division entre valeurs réelles.

**En cas d'opération entre un `int` et un `float`, la valeur entière est d'abord convertie en `float`, puis l'opération est effectuée entre `floats` et renvoie un `float`.**

# Affectation d'une valeur réelle à une variable entière

Quand on affecte une valeur réelle (`float` ou `double`) à une variable de type `int`, la partie fractionnaire est perdue.

Exemple:

```
float x = 1.5;  
int n = 3 * x; // !! provoque un warning  
           → n vaut 4 (commentaire)
```

**MAIS** le compilateur g++ génère un *warning* :

**warning: converting to 'int' from 'float'**

pour signaler qu'il y a potentiellement un problème, puisqu'on perd la partie fractionnaire.

Pour préciser au compilateur qu'il sait ce qu'il fait, le programmeur **DOIT** avoir une conversion de type `float` vers `int`:

```
n = int(3 * x);
```

# Conversions de type (*cast*)

- Le **cast implicite** est la conversion effectuée automatiquement par le compilateur quand nécessaire.

Le compilateur signale les *casts* implicites qui peuvent poser problème par un *warning* (notamment quand on affecte une valeur flottante à un entier):

```
float x = 1.5;
int n = 3 * x; // !! provoque un warning
```

L'inverse:

```
int n = 5;
float x = 3 * n;
```

ne provoque pas de *warning* avec g++, car il n'y a pas de perte de précision contrairement à avant.

- Un **cast explicite** est une conversion ajoutée par le programmeur, notamment pour éviter le *warning* précédent, ou pour effectuer une division flottante entre entiers:

```
float x = 1.5;
int n = int(3 * x); // pas de warning
```

```
float x = float(1) / 2; // x contient 0.5
```

# Types des constantes

Les nombres qui apparaissent dans les expressions sont appelées des constantes.

Les constantes entières sont traitées comme des `int`.

Les constantes réelles sont traitées comme des `float`.

Par exemple :

```
int n = 5;
```

```
float x = n / 2;
```

→ *x vaut 2*. `n` et `2` sont de type `int`, la division est donc ici la division entière.

```
int n = 2;
```

```
float x = 12.5 / n;
```

→ *x vaut 6.25*

`12.5` est de type `float`, la division est ici la division entre valeurs réelles.

Une constante finissant par un point (`.`), sans valeur flottante après le point, est traitée comme un `float`:

```
x = 12. / n;
```

→ *x vaut 2.4*

# Retour sur le piège de la division entière

Soit les déclarations:

```
int n = 1;  
float x;
```

Supposons qu'on veuille calculer la division *flottante* de `n` de type `int` par 2. Il suffit que le dénominateur ou le numérateur soient de type `float`. Il y a plusieurs possibilités:

```
x = float(n) / 2;
```

ou

```
x = n / 2.0;
```

ou

```
x = n / 2.;
```

ou

```
x = float(n) / 2.0;
```

**MAIS PAS:**

```
x = float(n / 2);
```

`n / 2` vaut 0, la conversion arrive trop tard !

# Exercice

Soit les instructions suivantes:

```
int n, p, q;  
float x, y;
```

```
n = 15;  
p = 10;  
x = 2.5;
```

Que contiennent les variables `q` ou `y` après les instructions suivantes:

```
y = x + n % p;  
q = x + n / p;  
y = (x + n) / p;  
y = (n + 1) / p;  
y = (n + 1.) / p;  
q = (n + 1.) / p;  
y = (float(n) + 1) / p;
```

Où faut-il ajouter des conversions de type pour éviter les *warnings*?

# Fonctions mathématiques

La bibliothèque standard du C++ fournit les fonctions mathématiques usuelles. Par exemple:

```
#include <iostream>
using namespace std;
#include <cmath>
```

**Il faut inclure `cmath`**

```
int main(int argc, char ** argv)
{
    float angle;
    float s;

    angle = 10 * 3.14159 / 180;
    s = sin(angle);
    // ...
}
```

→ `sin(angle)` calcule le sinus de la valeur contenue dans la variable `angle`.

# Fonctions mathématiques

Autres exemples d'expressions contenant des fonctions mathématiques:

```
y = 2 * cos(a) * sin(a);
```

```
b_1 = sqrt(delta);  
      (sqrt est la fonction racine carrée)
```

```
angle = atan(l / w);  
       (atan est la fonction arc tangente).
```

# Fonctions mathématiques

- `sin`
- `cos`
- `tan`
- `asin` sinus inverse ou arc sinus
- `acos`
- `atan`
- `atan2` `atan2(y, x)` fournit la valeur de l'arc-tangente de  $y / x$
- `sinh` sinus hyperbolique ou sh
- `cosh`
- `tanh`
- `exp`
- `log` logarithme népérien ou  $\ln$
- `log10` logarithme à base 10 ou  $\log$
- `pow` `pow(x, y)` fournit la valeur de  $x^y$
- `sqrt` racine carrée
- `ceil` `ceil(x)` renvoie le plus petit entier qui ne soit pas inférieur à  $x$ : `ceil(2.6) = 3`
- `floor` `floor(x)` renvoie le plus grand entier qui ne soit pas supérieur à  $x$ : `floor(2.6) = 2`
- `fabs` valeur absolue

# Séances d'exercices

Les exercices repérés avec une étoile (\*) peuvent être sautés si vous vous sentez à l'aise;

Les exercices repérés avec une † sont facultatifs, et peuvent être gardés pour les révisions.

# Séances d'exercices

Les exercices repérés avec une étoile (\*) peuvent être sautés si vous vous sentez à l'aise;

Les exercices repérés avec une † sont facultatifs, et peuvent être gardés pour les révisions.

**Créez vos programmes dans le répertoire `programmation1`, pas dans le *home directory* directement:**

```
mkdir programmation1  
cd programmation1
```

**Compilation:**

```
g++ -Wall -o nom_executable nom_programme_c++.cpp
```

**exemple:**

```
g++ -Wall -o hello helloworld.cpp
```

**Pour exécuter le programme compilé pour cet exemple:**

```
./hello
```

# Compléments

# Pour écrire à l'écran en C

L'instruction:

```
printf("La variable x contient %f.\n", x);
```

ressemble à l'instruction que nous avons déjà vue:

```
printf("Bonjour\n");
```

mais ici la fonction `printf` reçoit deux arguments.

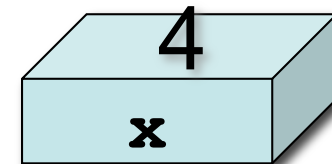
"La variable x contient %f.\n" n'est pas affiché directement, c'est un **format** qui guide l'affichage.

La notation `%f` est un code de format qui sera remplacé par une valeur flottante (*f*). Cette valeur doit se trouver dans la liste des arguments donnés à `printf`. Il s'agit de la valeur contenue par la variable `x`.

```
printf("La variable x contient %f.\n", x);
```

affiche:

La variable x contient 4.



# Lire la valeur d'une variable entière en C

Supposons qu'on ait déclaré la variable `n` de type `int`:

```
int n;
```

L'instruction:

```
scanf ("%d", &n) ;
```

lit la valeur que tape l'utilisateur au clavier et stocke cette valeur dans la variable `n`.

Comme pour `printf`, le premier argument est une chaîne de caractères qui donne le format de la lecture. Cette chaîne de caractères ne peut contenir que des formats, pas de messages.

**Attention:** notez le `&` devant le `n` dans le cas du `scanf` (nous verrons plus tard pourquoi le `&` est nécessaire).

# Afficher la valeur d'une variable entière en C

Rappel: la fonction `printf` permet d'afficher des chaînes de caractères à l'écran:

```
printf("Bonjour\n");
```

Supposons que la variable `n` soit déclarée et initialisée comme suit:

```
int n = 2;
```

l'instruction:

```
printf("%d", n);
```

affichera à l'écran:

```
2
```

Entre les parenthèses qui suivent la fonction `printf`, on trouve deux éléments:

- `"%d"` est un *format* d'affichage, place entre guillemets (`"`)
- le nom de la variable qu'on souhaite afficher, ici `n`.

La valeur de la variable `n` remplace le `%d` lors de l'exécution de la fonction `printf`.

On peut également faire:

```
printf("La variable n contient %d.\n", n);
```

affiche:

```
La variable n contient 2.
```

# Erreur à éviter avec `scanf` (en C)

Si vous oubliez le `&` dans la fonction `scanf`:

```
scanf("%d",n); // !!!
```

au lieu de:

```
scanf("%d",&n);
```

le compilateur ne signale *pas* d'erreur si on n'utilise pas l'option `-Wall`. En revanche, le programme ne fonctionnera pas correctement, et il risque fort de "planter" en affichant le message:

```
Segmentation fault (core dumped)
```

Utilisé avec `-Wall`, le compilateur affiche un *warning*:

```
In function `main':
```

```
7: warning: format argument is not a pointer (arg 2)
```

# Comment afficher ou lire au clavier une variable `float` ou `double` (en C)

Comme pour les variables de type `int` en remplaçant le `%d` par:

- `%f` pour les variables de type `float`;
- `%lf` pour les variables de type `double`.

```
float x = 5;
```

```
double y;
```

```
printf(" x vaut %f\n", x);
```

```
printf("Entrez la valeur de y:\n");
```

```
scanf("%lf", &y);
```

# Casts explicites en C

```
float x = 1.5;  
int n = (int)(3 * x); // pas de warning  
  
float x = (float)1 / 2; // x contient 0.5
```

# Opérateurs << et >>

L'opérateur << permet de décaler les bits vers la gauche, donc d'opérer des multiplications par des puissances de 2:

```
a = 3 << 2;
```

→ a contient 12;

>> : même chose, vers la droite:

```
a = 3 >> 1;
```

→ a contient 1.

# Priorités entre opérateurs

Lorsque plusieurs opérateurs apparaissent dans une expression, il est nécessaire de savoir dans quel ordre ils doivent être appliqués.

$$5 + 3 * a * (b + 2)$$

Les règles de priorité sont celles habituelles:

1. l'opérateur unaire – (opposé) a la priorité la plus élevée;
2. ensuite, à un même niveau: \* , / et %
3. sur un dernier niveau: + et – (la soustraction).

En cas de priorités identiques, les calculs s'effectuent de gauche à droite.

Les parenthèses ( et ) permettent de forcer les priorités.