A ajouter : setw, [syntaxe alternative : flags, setf, unsetf, precision, width]. Cf std::ios_base

Les nombres à virgule flottante

Vous avez vu dans les chapitres précédents comment écrire et manipuler les nombres entiers. Et vous avez vu que le C++ faisait la distinction entre les nombres entiers (nombres “sans virgule”) et les nombres réels (nombre “avec virgule”). En particulier, lors d'une division, le résultat est totalement différent selon le type de nombre.

main.cpp

```
#include <iostream>

int main() {
    std::cout << "11   / 4   = " << 11   / 4   << std::endl;
// division entière
    std::cout << "11.0 / 4.0 = " << 11.0 / 4.0 << std::endl;
// division réelle
}
```

affiche :

```
11   / 4   = 2
11.0 / 4.0 = 2.75
```

Pour faire des calculs mathématiques, les nombres entiers ne seront généralement pas suffisants. Par exemple, si vous souhaitez calculer la circonférence d'un cercle à partir de son diamètre, il faudra multiplier celui-ci par le nombre Pi, qui vaut environ 3,14159265358979323846264338327950288... Ces types de nombres et de calculs ne peuvent pas être réalisés avec des nombres entiers, il faudra donc utiliser des nombres conçus spécialement dans ce but : les nombres à virgule flottante (*floating-point numbers*).

Les nombres réels et représentations binaires
En toute rigueur, un “nombre réel” est un nombre qui peut s'écrire avec une infinité de chiffres après la virgule. Ce type de nombre est un concept purement mathématique, il n'est pas possible de manipuler un nombre réel avec un ordinateur (du fait qu'un ordinateur a une mémoire limitée).

Les ordinateurs manipulent en fait un concept proche des nombres réels : les nombres à virgule flottante, qui ont un nombre limité de chiffres après la virgule. Ces nombres à virgule flottante sont manipulées en mémoire dans une représentation binaire avec un nombre de bits fixés (en general, 32 bits pour le type `float` et 64 bits pour le type `double`, mais cela peut changer selon le systeme.

Une valeur réelle dans le code sera donc convertie dans une représentation binaire en mémoire. Et l'affichage de cette valeur avec `std::cout` dépendra des parametres choisis pour représenter les nombres réelles. Il peut donc y avoir des différences entre la valeur entrée dans le code, la valeur dans la mémoire de l'ordinateur (et les calculs) et la valeur affichée.

Il est assez classique d'utiliser le terme “nombre réel” en informatique, par abus de langage.

Écrire des nombres réels

Les nombres à virgule flottante sont donc simplement des nombres qui s'écrivent avec des chiffres après la virgule, comme par exemple Pi.

Un rappel important : en français, les nombres réels sont écrits avec une virgule (*comma* en anglais). Le C++ est basé sur l'anglais, il utilise donc le point comme séparateur décimal. La virgule ayant un sens spécifique en C++, vous n'aurez peut-être pas de message si vous faites l'erreur. Faites donc attention sur ce point.

À partir de là, l'écriture de nombres réels est relativement simple, par exemple pour écrire quelques constantes de physique :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Pi = " << 3.1415926535 << std::endl;
    std::cout << "Nombre d'or = " << 1.6180339887 << std::
endl;
    std::cout << "Vitesse de la lumière = " << 299792.458 << "
km/s" << std::endl;
}
```

affiche :

```
Pi = 3.14159
Nombre d'or = 1.61803
Vitesse de la lumière = 299792 km/s
```

Comme pour les nombres entiers, il est possible d'utiliser le guillemet droit simple ' pour faciliter la lecture des nombres contenant beaucoup de chiffres. Par convention, on sépare généralement les nombres en blocs de 3 chiffres :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Pi = " << 3.141'592'653'5 << std::endl;
    std::cout << "Nombre d'or = " << 1.618'033'988'7 <<
std::endl;
    std::cout << "Vitesse de la lumière = " << 299'792.458
<< " km/s" << std::endl;
}
```

Lorsque la partie entière (la partie à gauche du point) ou la partie décimale (à droite du point) d'un nombre ne contient que des zéros, il est possible de ne pas les écrire. Par exemple, “0.123” est équivalent à ”.123” et “123.0” est équivalent à “123.”. Mais faites attention de ne pas oublier le point, sinon cela correspondra à un nombre entier :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0.123 << " = " << .123 << std::endl;
    std::cout << 123.0 << " = " << 123. << std::endl;
    std::cout << (123. / 5.) << " != " << (123 / 5) << std::
endl;
}
```

ce qui affiche :

```
0.123 = 0.123
123 = 123
24.6 != 24
```

Cependant, il faut se rappeler qu'un code sera plus souvent lu qu'écrit et il n'y a aucun intérêt à faire l'économie d'un caractère. Il est donc préférable d'écrire quand même les chiffres zéro pour la lisibilité. C'est ce que l'on fera systématiquement dans ce cours.

Notation scientifique

Pour Pi ou la vitesse de la lumière *c*, cela ne pose pas de problème de les écrire comme présentés au-dessus. Par contre, si l'on souhaite écrire par exemple la constante de perméabilité magnétique du vide *μ0*, il faudra écrire :

```
std::cout << "Perméabilité magnétique du vide = " <<
    0.000'001'256'637'061'4 << "  kg.m/A²/s²" << std::endl;
```

Ce qui commence à faire beaucoup de zéros. Si on veut écrire la constante de Planck, c'est encore plus compliqué : il faut écrire 34 zéros après le séparateur décimal. On ne va pas le faire, cela devient trop compliqué. Heureusement, le C++ prend en charge la notation scientifique des nombres réels.

La notation scientifique consiste à écrire un nombre sous la forme :

$$ mantisse \times 10 ^ { exposant } $$

La mantisse (*mantissa* ou *significand* en anglais) est un nombre réel positif ou négatif et l'exposant (*exponent* en anglais) est un nombre entier positif ou négatif.

Pour obtenir la notation scientifique d'un nombre, il faut multiplier ou diviser ce nombre plusieurs fois par 10, jusqu'à obtenir un nombre supérieur ou à égal à 1 et strictement inférieur à 10. Le résultat est la mantisse, le nombre de fois que l'on a multiplié ou divisé par 10 est l'exposant.

Par exemple, si on prend le nombre 0.000123. On peut écrire :

```
0.000123 * 10 = 0.00123
0.00123  * 10 = 0.0123
0.0123   * 10 = 0.123
0.123    * 10 = 1.23
```

Il faut donc multiplier 4 fois 0.000123 par 10 pour obtenir 1.23. Le nombre 0.000123 peut donc s'écrire :

$$ 1.23 \times 10 ^ { -4 } $$

Pour écrire un nombre en C++ en utilisant cette notation, il faut ajouter l'exposant après le caractère “e” ou “E” dans l'écriture d'un nombre. Par exemple :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Perméabilité magnétique du vide = " <<
        1.256'637'061'4e-6 << "  kg.m/A²/s²" << std::endl;
    std::cout << "Constante de Planck = " << 6.626'069
'57e-34 << " kg.m²/s" << std::endl;
}
```

affiche :

```
Perméabilité magnétique du vide = 1.25664e-06  kg.m/A²/s²
Constante de Planck = 6.62607e-34 kg.m²/s
```

Un dernier point sur la notation scientifique. Si vous écrivez :

main.cpp
```cpp
#include <iostream>

int main() {
    std::cout << 123456789.0e10 << std::endl;
    std::cout << 0.0000000123456789e-10 << std::endl;
}
```

cela affichera :

```
1.23457e+18
1.23457e-18
```

On peut remarquer deux points sur le résultat affiché :

- la fin des chiffres (le 8 et le 9) n'est pas affichée. On verra cette problématique dans la partie suivante, sur la mise en forme de l'affichage des nombres réels.
- les exposants utilisés ont été modifiés automatiquement lors de l'affichage.

Mettre en forme la sortie

Lorsque vous afficher une valeur avec `std::cout`, cela convertit la représentation binaire interne de cette valeur (compréhensible uniquement par un ordinateur) en quelque chose de compréhensible par les humains. La façon dont `std::cout` affiche les valeurs peut être paramétrée pour personnaliser l'affichage.

C'est un point fondamental en informatique : la façon dont les données sont affichées ne correspond généralement pas à

comment elles sont manipulées en interne (représentation binaire). C'est valable pour les nombres, mais également pour n'importe quel autre type de données.
La valeur exacte que vous écrivez dans le code ne sera pas forcément conservée dans la représentation en mémoire. Et avec une même valeur en mémoire, vous pouvez afficher des valeurs différentes, en fonction de parametres d'affichage choisi.

La première chose que vous allez pouvoir modifier est l'utilisation de la notation scientifique ou non. La directive `std::scientific` force l'affichage en notation scientifique, `std::fixed` force l'affichage sans notation scientifique et `std::defaultfloat` affiche en utilisant la notation par défaut.

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Pi" << std::endl;
    std::cout << " Notation scientifique : " << std::
scientific << 3.141'592'653'5 << std::endl;
    std::cout << " Notation fixe :         " << std::fixed
<< 3.141'592'653'5 << std::endl;
    std::cout << " Notation par défaut :   " << std::
defaultfloat << 3.141'592'653'5 << std::endl;
    std::cout << std::endl;

    std::cout << "Constante de Planck" << std::endl;
    std::cout << " Notation scientifique : " <<
std::scientific << 6.626'069'57e-34 << std::endl;
    std::cout << " Notation fixe :         " << std::fixed
<< 6.626'069'57e-34 << std::endl;
    std::cout << " Notation par défaut :   " <<
std::defaultfloat << 6.626'069'57e-34 << std::endl;
}
```

affiche :

```
Pi
 Notation scientifique : 3.141593e+00
```

```
 Notation fixe :          3.141593
 Notation par défaut :    3.14159

Constante de Planck
 Notation scientifique : 6.626070e-34
 Notation fixe :          0.000000
 Notation par défaut :    6.62607e-34
```

Notez bien que ces directives ne s'appliquent qu'aux nombres réels. Si vous écrivez un nombre entier, celui-ci ne sera pas affiché en notation scientifique :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::scientific << 3.0 << std::endl;
    std::cout << std::scientific << 3 << std::endl;
}
```

affiche :

```
3.000000e+00
3
```

N'oubliez jamais ce point fondamental : le C++ est un langage de programmation à typage fort, les types ont une importance particulière.
Même si `3`, `3.0`, `“3”` et '3' représentent la même chose pour vous (le chiffre 3), ces valeurs ont des types différents (respectivement un nombre entier, un nombre réel, une chaîne de caractères et un caractère) et s'utilisent différemment en C++.

La directive `std::showpos` permet d'afficher le signe plus devant les nombres positifs et la directive `std::noshowpos` permet de ne pas l'afficher.

La directive `std::showpoint` permet d'afficher le séparateur décimal et

ajoute le nombre de zéros nécessaires après le séparateur décimal pour atteindre le nombre de chiffres fixés par la directive `std::setprecision` décrite plus bas. La directive `std::noshowpoint` permet de ne pas afficher le séparateur décimal pour les nombres dont la partie décimale est nulle.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::showpos << 123.456 << std::endl;
    std::cout << std::showpos << 0.123 << std::endl;
    std::cout << std::showpos << 123.0 << std::endl;
    std::cout << std::noshowpos << 123.456 << std::endl;
    std::cout << std::endl;

    std::cout << std::showpoint << 123.0 << std::endl;
    std::cout << std::noshowpoint << 123.0 << std::endl;
}
```

affiche :

```
+123.456
+0.123
+123
123.456

123.000
123
```

Pour terminer, la directive `std::setprecision` permet de définir le nombre de chiffres significatifs à afficher (donc sans compter les zéros au début et à la fin des nombres réels). Cette directive prend un nombre entier en paramètre, correspondant au nombre de chiffres à afficher.

Cette directive est disponible dans le fichier d'en-tête `iomanip`, il faut donc l'inclure au début du programme avec la directive de préprocesseur `#include` :

main.cpp

```cpp
#include <iostream>
#include <iomanip>

int main() {
    std::cout << "Pi (défaut) = " << 3.141'592'653'5 <<
std::endl;
    std::cout << "Pi (précision 0) = " <<
std::setprecision(0) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 1) = " << std::setprecision(
1) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 2) = " <<
std::setprecision(2) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 5) = " << std::setprecision(
5) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 10) = " <<
std::setprecision(10) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 30) = " << std::setprecision
(30) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 200) = " <<
std::setprecision(200) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (valeur invalide) = " << std::
setprecision(-1) << 3.141'592'653'5 << std::endl;
}
```

affiche :

```
Pi (défaut) = 3.14159
Pi (précision 0) = 3
Pi (précision 1) = 3
Pi (précision 2) = 3.1
Pi (précision 5) = 3.1416
Pi (précision 10) = 3.141592654
Pi (précision 30) = 3.14159265350000005412312020781
Pi (précision 200) =
3.14159265350000005412312020780518651008605957031250
Pi (valeur invalide) = 3.14159
```

Vous pouvez remarquer que si vous définissez une précision supérieure au nombre de chiffres que vous avez écrit dans votre littérale (par exemple 30 dans le code précédent), alors `std::cout` affichera des chiffres incorrects à la fin de votre nombre.

Si vous définissez une valeur très grande (200 dans le code précédent), alors le nombre de chiffres affichés sera limité à 50 maximum (le nombre maximal de chiffres peut varier selon le compilateur et le système).

Pour terminer, si vous définissez une valeur négative, alors `std::cout` affiche de nouveau les nombres en utilisant la précision par défaut (6 chiffres dans l'exemple précédent).

Opérateurs arithmétiques

Arithmétique de base

Comme pour les nombres entiers, il est possible de réaliser des calculs arithmétiques avec les nombres réels : addition `+`, soustraction `-`, multiplication `*` et division `/` (n'oubliez pas la différence entre division entière et réelle).

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Addition : " << 12.34 + 56.78 << std::endl;
    std::cout << "Soustraction : " << 12.34 - 56.78 << std::
endl;
    std::cout << "Multiplication: " << 12.34 * 56.78 << std
::endl;
    std::cout << "Division : " << 12.34 / 56.78 << std::endl;
}
```

affiche :

```
Addition : 69.12
Soustraction : -44.44
Multiplication: 700.665
Division : 0.21733
```

Le modulo réel

Dans le cas des nombres entiers, vous avez vu que le reste d'une division (le modulo) pouvait se calculer avec l'opérateur `%`. Le reste d'une division entière peut s'exprimer avec l'équation suivante :

$$ \text{dividende} = \text{diviseur} \times \text{quotient} + \text{reste} \qquad (\text{avec} ~ 0 \leq \text{reste} < \text{quotient}) $$

Dans le cas d'une division réelle, si toutes les variables de cette équation sont des nombres réelles, il est toujours possible de trouver une valeur pour le quotient tel que le reste est nul. L'equation devient alors simplement :

$$ \text{dividende}= \text{diviseur}\times \text{quotient} $$

C'est le résultat retourné en C++ par l'opérateur `/` lors que les variables sont des réels. Et c'est également pour cette raison que le modulo n'a généralement pas de sens dans le cas des divisions réelles, et que l'opérateur `%` n'est pas définie dans ce cas.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << (1234.0 % 56.0) << std::endl;
}
```

Affiche une erreur :

```
main.cpp:4:26: error: invalid operands to binary expression
('double' and 'double')
    std::cout << (1234.0 % 56.0) << std::endl;
                  ~~~~~~ ^ ~~~~
```

Cependant, il est possible de modifier l'equation precedente pour définir une sorte de modulo pour la division reelle. Pour cela, on va conserver l'equation precedente, mais modifier les conditions.

$$ \text{dividende} = \text{diviseur} \times \text{quotient} + \text{reste} \qquad $$

avec :

- le dividende, le diviseur et le reste sont des nombres réels ;
- le quotient est un nombre entier ;
- la valeur absolue du reste est le plus petit possible (et donc toujours plus petit que le diviseur).

Pour réaliser ce calcul, vous pouvez utiliser l'une des fonctions C++ suivantes, définies dans le fichier d'en-tête `<cmath>` :

- `std::remainder`, qui ne garantie pas que le reste est de même signe que le dividende ;
- `std::fmod`, qui garantie que le reste est de même signe que le dividende.

main.cpp

```cpp
#include <iostream>
#include <cmath>

int main() {
    const auto dividende { 5.1 };
    const auto diviseur  { 2.0 };
    std::cout << std::fmod     ( dividende, diviseur) << std
::endl;
    std::cout << std::fmod     (-dividende, diviseur) << std
::endl;
    std::cout << std::remainder( dividende, diviseur) << std
::endl;
    std::cout << std::remainder(-dividende, diviseur) << std
::endl;
}
```

affiche :

```
1.1
-1.1
-0.9
```

```
0.9
```

Vous pouvez verifier les differents resultats pour etre sur que les valeurs affichées sont bien les plus petites possibles, selon les contraintes de chaque fonction. Par exemple, pour le premier `std::fmod`, vous pouvez voir qu'il est possible d'écrire l'équation précédente avec les valeurs données :

$$ 5.1 = 2.0 \times 2 + 1.1 $$

- Avec un quotient qui vaut 3, alors le reste vaut -0.9, ce qui est plus petit que 1.1 (en valeur absolue), mais le signe du reste n'est pas le même que le dividende. De même, toutes valeurs entières supérieures a 3 pour le quotient n'est pas valide, pour la même raison.
- Avec un quotient qui vaut 1, alors le reste vaut 2.1, ce qui n'est pas la plus petite valeur possible et le reste est supérieur au dividende. Et de même pour toutes valeurs entières inférieures à 1.
- La seule valeur acceptable pour le quotient est donc 2 et le reste vaut 1.1.

N'hésitez pas à faire la vérification pour les autres lignes de code.

Les comparaisons

Pour comparer deux nombres réels, vous pouvez utiliser les opérateurs de comparaison présentés dans le chapitre sur les entiers : est égal `==`, est différent `!=`, est supérieur `>`, est supérieur ou égal `>=`, est inférieur `<`, est inférieur ou égal `<=`.

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::boolalpha;
```

```
    std::cout << "Est égale : " << (12.34 == 56.78) << std::
endl;
    std::cout << "Est différent : " << (12.34 != 56.78) <<
std::endl;
    std::cout << "Est supérieur : " << (12.34 > 56.78) <<
std::endl;
    std::cout << "Est supérieur ou égal : " << (12.34 >=
56.78) << std::endl;
    std::cout << "Est inférieur : " << (12.34 < 56.78) <<
std::endl;
    std::cout << "Est inférieur ou égal : " << (12.34 <=
56.78) << std::endl;
}
```

affiche :

```
Est égale : false
Est différent : true
Est supérieur : false
Est supérieur ou égal : false
Est inférieur : true
Est inférieur ou égal : true
```

Egalité de nombres réels et epsilon

Encore une fois, il faut insister sur un point très important : ce qui est affiché ne correspond pas forcement à ce qu'il y a en mémoire. Par exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 3.0000000000001 << std::endl;
    std::cout << 3.0000000000000001 << std::endl;
}
```

affiche :

```
3
```

```
3
```

Si vous utilisez l'opérateur d'égalité sur ces nombres, vous aurez peut-être des surprises :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 3 << " est égal à " << 3.0000000000001 << "
? "
              << std::boolalpha << (3 == 3.0000000000001) <<
std::endl;
    std::cout << 3 << " est égal à " << 3.0000000000000001
<< " ? "
              << std::boolalpha << (3 == 3.0000000000000001)
<< std::endl;
}
```

affiche :

```
3 est égal à 3 ? false
3 est égal à 3 ? true
```

Les nombres 3, 3.0000000000001 et 3.0000000000000001 sont affichés de la même façon (ce qui est normal, si vous vous souvenez du rôle de `std::setprecision`). Par contre, le résultat du test d'égalité change aussi, ce qui est plus surprenant.

Dans le premier cas, 3.0000000000001 est effectivement enregistré en mémoire comme différent de 3. Le test d'égalité échoue puisque l'ordinateur sait que ces nombres sont différents (même s'il affiche le même nombre). Dans le second cas, le nombre dépasse les capacités de l'ordinateur (plus précisément la façon dont les nombres sont enregistrés en mémoire). En effet, le nombre est arrondi en mémoire à 3 donc le test d'égalité n'échoue pas.

Il est facile de voir ici que les nombres entrés sont différents. Mais imaginez que vous réalisez des calculs scientifiques complexes. Cela pourrait générer des nombres qui semblent identiques (à l'affichage),

mais qui sont en fait différents.

Le problème posé par la comparaison entre deux nombres réels est d'ailleurs indiquée par les compilateurs. Voici par exemple les avertissements produits par Clang :

```
prog.cc:5:39: warning: comparing floating point with == or
!= is unsafe [-Wfloat-equal]
              << std::boolalpha << (3 == 3.0000000000001) <<
std::endl;
                                     ~ ^  ~~~~~~~~~~~~~~~
prog.cc:7:39: warning: comparing floating point with == or
!= is unsafe [-Wfloat-equal]
              << std::boolalpha << (3 == 3.0000000000000001)
<< std::endl;
                                     ~ ^  ~~~~~~~~~~~~~~~~~
```

Pour éviter cela, on ne compare pas directement l'égalité de deux nombres réels en général. On va considérer que deux nombres sont égaux s'ils sont suffisamment proches, compte tenu d'une éventuelle erreur de précision. Dit autrement, il y a égalité si la valeur absolue de la différence entre les deux nombres est inférieure à une certaine erreur de précision maximale appelée *epsilon* (nous reviendrons sur la valeur absolue plus tard).

```
std::abs(nombre1 - nombre2) < epsilon
```

Epsilon sera choisie en fonction du type de calculs que vous réalisez, de la précision des nombres, de l'ordinateur, etc. Bref, elle va dépendre du contexte.

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << 3 << " est égal à " << 3.0000000000001 << "
? "
              << std::boolalpha <<  (std::abs(3 -
3.0000000000001) < 0.0001) << std::endl;
```

```
    std::cout << 3 << " est égal à " << 3.0000000000000001
<< " ? "
              << std::boolalpha <<  (std::abs(3 -
3.0000000000000001) < 0.0001) << std::endl;
}
```

affiche :

```
3 est égal à 3 ? true
3 est égal à 3 ? true
```

Infini et pas-un-nombre

Vous avez vu dans le chapitre sur les entiers que la division par zéro produisait un *undefined behavior* (comportement indéterminé en français). Essayons le même code en utilisant des nombres réels :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "1.0 / 0.0 = " << (1.0 / 0.0) << std::endl;
}
```

affiche :

```
1.0 / 0.0 = inf
```

Contrairement à la division sur les nombres entiers, la division sur les nombres réels ne produit pas un comportement indéterminé. Le résultat affiché “inf” signifie *infinity* (infini en français), ce qui a un sens au niveau mathématique.

Si on modifie le code pour calculer la division de 0 par 0 :

main.cpp

```
#include <iostream>

int main() {
```

```
    std::cout << "0.0 / 0.0 = " << (0.0 / 0.0) << std::endl;
}
```

on obtient :

```
0.0 / 0.0 = nan
```

Le résultat affiché “nan” signifie *not-a-number* (pas-un-nombre en français). Ce résultat est obtenu lorsqu'un calcul n'a pas de sens mathématique. Mais du point de vue du C++, cela n'est pas une erreur (au sens d'erreur de calcul ou d'exécution).

Ces deux valeurs particulières “inf” et “nan” sont parfaitement définies en C++. Pour les écrire directement vous pouvez utiliser les macros `INFINITY` et `NAN` définies dans le fichier d'en-tête `cmath` :

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Infini : " << INFINITY << std::endl;
    std::cout << "Pas-un-nombre : " << NAN << std::endl;
}
```

affiche :

```
Infini : inf
Pas-un-nombre : nan
```

Cependant, dans la majorité des cas, vous aurez plus souvent besoin de vérifier qu'un calcul ne produit pas une de ces valeurs. Pour cela, vous pouvez utiliser une des fonctions suivantes :

- `std::isinf()` pour tester si une valeur est infinie ;
- `std::isnan()` pour tester si une valeur est pas-un-nombre ;
- `std::isfinite()` pour tester si une valeur est finie, c'est-à-dire qu'elle n'est ni infinie, ni pas-un-nombre.

Ces fonctions sont également définies dans le fichier d'en-tête `cmath`, il

faut donc l'inclure dans votre code.

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << std::boolalpha;
    std::cout << "isinf(1.0 / 1.0) = " << std::isinf(1.0 /
1.0) << std::endl;
    std::cout << "isinf(0.0 / 1.0) = " << std::isinf(0.0 /
1.0) << std::endl;
    std::cout << "isinf(1.0 / 0.0) = " << std::isinf(1.0 /
0.0) << std::endl;
    std::cout << "isinf(0.0 / 0.0) = " << std::isinf(0.0 /
0.0) << std::endl;

    std::cout << std::endl;
    std::cout << "isnan(1.0 / 1.0) = " << std::isnan(1.0 /
1.0) << std::endl;
    std::cout << "isnan(0.0 / 1.0) = " << std::isnan(0.0 /
1.0) << std::endl;
    std::cout << "isnan(1.0 / 0.0) = " << std::isnan(1.0 /
0.0) << std::endl;
    std::cout << "isnan(0.0 / 0.0) = " << std::isnan(0.0 /
0.0) << std::endl;

    std::cout << std::endl;
    std::cout << "isfinite(1.0 / 1.0) = " << std::isfinite(
1.0 / 1.0) << std::endl;
    std::cout << "isfinite(0.0 / 1.0) = " << std::isfinite(
0.0 / 1.0) << std::endl;
    std::cout << "isfinite(1.0 / 0.0) = " << std::isfinite(
1.0 / 0.0) << std::endl;
    std::cout << "isfinite(0.0 / 0.0) = " << std::isfinite(
0.0 / 0.0) << std::endl;
}
```

affiche :

```
isinf(1.0 / 1.0) = false
```

```
isinf(0.0 / 1.0) = false
isinf(1.0 / 0.0) = true
isinf(0.0 / 0.0) = false

isnan(1.0 / 1.0) = false
isnan(0.0 / 1.0) = false
isnan(1.0 / 0.0) = false
isnan(0.0 / 0.0) = true

isfinite(1.0 / 1.0) = true
isfinite(0.0 / 1.0) = true
isfinite(1.0 / 0.0) = false
isfinite(0.0 / 0.0) = false
```

Les fonctions mathématiques

Pour terminer avec l'utilisation de base des nombres réels, le C++ propose de nombreuses fonctions mathématiques usuelles, comme le calcul des puissances et des racines ou les fonctions trigonométriques. Nous n'allons pas toutes les voir dans ce chapitre, elles ne présentent pas de difficulté particulière d'utilisation. Ces fonctions sont définies dans le fichier d'en-tête `cmath`, vous trouverez la totalité des fonctions dans la documentation : Common mathematical functions.

Voici quelques exemples d'utilisation :

main.cpp

```cpp
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Fonctions basiques" << std::endl;
    std::cout << "valeur absolue : " << fabs(-1.0) << std::
endl;
    std::cout << "minimum : " << fmin(1.0, 2.0) << std::endl;
    std::cout << "maximum : " << fmax(1.0, 2.0) << std::endl;
    std::cout << std::endl;
    std::cout << "Fonctions exponentielles et logarithmiques"
<< std::endl;
```

```
    std::cout << "exponentielle naturelle : " << exp(1.0) <<
std::endl;
    std::cout << "logarithme naturel : " << log(1.0) << std
::endl;
    std::cout << "logarithme décimal : " << log10(1.0) <<
std::endl;
    std::cout << std::endl;
    std::cout << "Fonctions puissances" << std::endl;
    std::cout << "puissance : " << pow(2.0, 3.0) << std::
endl;
    std::cout << "racine carrée : " << sqrt(2.0) << std::
endl;
    std::cout << "racine cubique : " << cbrt(2.0) << std::
endl;
    std::cout << std::endl;
    std::cout << "Fonctions trigonométriques" << std::endl;
    std::cout << "sinus : " << sin(1.0) << std::endl;
    std::cout << "cosinus : " << cos(1.0) << std::endl;
    std::cout << "tangente : " << tan(1.0) << std::endl;
    std::cout << "arc sinus : " << asin(1.0) << std::endl;
    std::cout << "arc cosinus : " << acos(1.0) << std::endl;
    std::cout << "arc tangente : " << atan(1.0) << std::endl;
    std::cout << std::endl;
    std::cout << "Fonctions hyperboliques" << std::endl;
    std::cout << "sinus hyperbolique : " << sinh(1.0) << std
::endl;
    std::cout << "cosinus hyperbolique : " << cosh(1.0) <<
std::endl;
    std::cout << "tangente hyperbolique : " << tanh(1.0) <<
std::endl;
    std::cout << "argument sinus hyperbolique : " << asinh(
1.0) << std::endl;
    std::cout << "argument cosinus hyperbolique : " << acosh
(1.0) << std::endl;
    std::cout << "argument tangente hyperbolique : " <<
atanh(1.0) << std::endl;
    std::cout << std::endl;
    std::cout << "Fonctions partie entière" << std::endl;
    std::cout << "partie entière par défaut : " << floor(1.3)
<< std::endl;
    std::cout << "partie entière par excès : " << ceil(1.3)
<< std::endl;
```

```
    std::cout << "troncature : " << trunc(1.3) << std::endl;
    std::cout << "arrondi : " << round(1.3) << std::endl;
}
```

affiche :

```
Fonctions basiques
valeur absolue : 1
minimum : 1
maximum : 2

Fonctions exponentielles et logarithmiques
exponentielle naturelle : 2.71828
logarithme naturel : 0
logarithme décimal : 0

Fonctions puissances
puissance : 8
racine carrée : 1.41421
racine cubique : 1.25992

Fonctions trigonométriques
sinus : 0.841471
cosinus : 0.540302
tangente : 1.55741
arc sinus : 1.5708
arc cosinus : 0
arc tangente : 0.785398

Fonctions hyperboliques
sinus hyperbolique : 1.1752
cosinus hyperbolique : 1.54308
tangente hyperbolique : 0.761594
argument sinus hyperbolique : 0.881374
argument cosinus hyperbolique : 0
argument tangente hyperbolique : inf

Fonctions partie entière
partie entière par défaut : 1
partie entière par excès : 2
troncature : 1
```

```
arrondi : 1
```

Chapitre précédent	Sommaire principal	Chapitre suivant