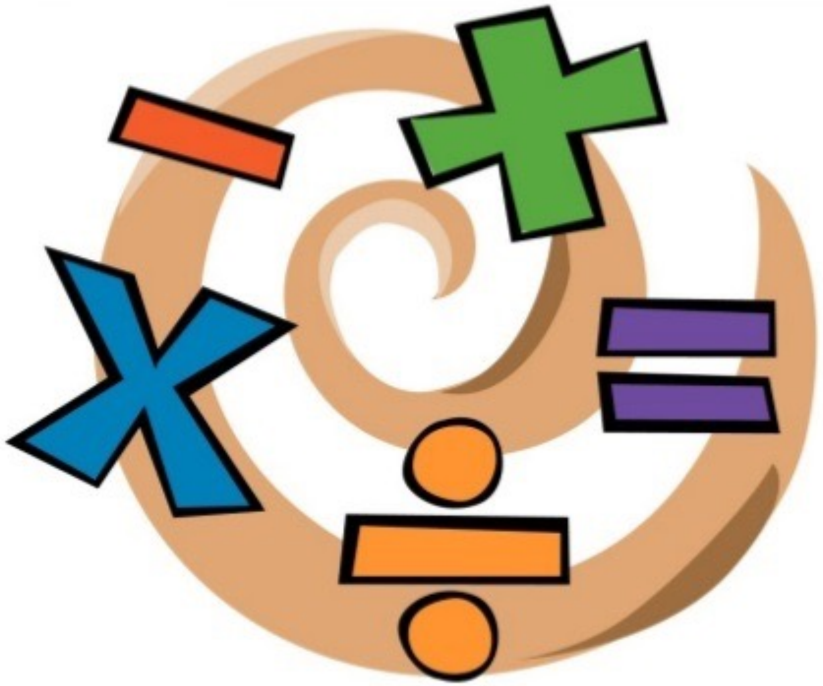
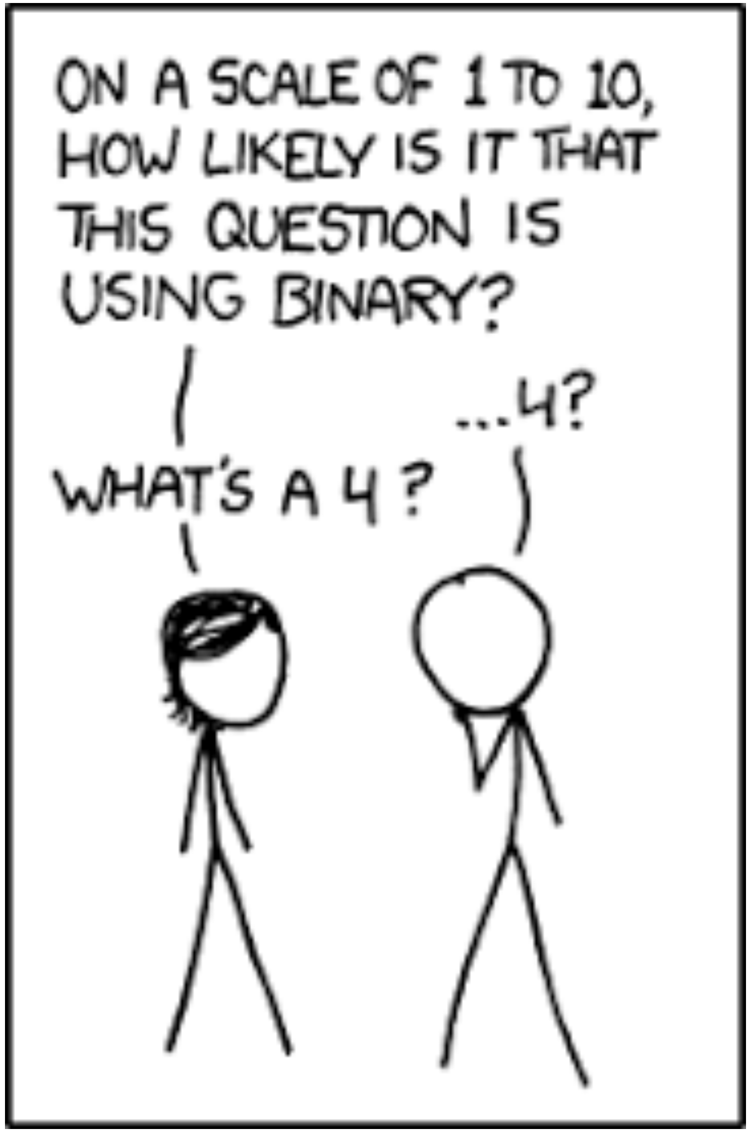


Types arithmétiques et conversions

Entiers

- Entiers signés
  - Représentation en mémoire
  - `numeric_limits`
  - Dépassement
- Entiers non signés
- Alias de type et entiers de taille fixe
- Entiers littéraux
- Entrées / sorties entières



https://xkcd.com/953/

Entiers signés

- Stockés en mémoire sous forme binaire
  - **Bit** = chiffre binaire : 0 ou 1
  - **Octet** = groupe de 8 bits
  - **Byte** = plus petit groupe de bits adressable. Typiquement un octet, mais pas toujours. Le nombre exact est défini dans la constante `CHAR_BIT` provenant de `<climits>`.
- Le **nombre de bits utilisés** détermine le nombre d'entiers représentables
  - Avec *n* bits - $\frac{n}{8}$ octets - on peut représenter jusque 2^n entiers différents
  - Les entiers positifs de 0 à $2^{n-1} - 1$ sont codés en base 2. Leur premier bit est toujours 0
  - Pour les entiers négatifs, cela dépend de leur représentation, qui n'est pas spécifiée par le standard C++. Leur premier bit est toujours 1.

Exemple : entiers positifs sur 16 bits

Entier	Binaire	Hexadécimal
1	0000000000000001	0001
2	0000000000000010	0002
3	0000000000000011	0003
16	0000000000010000	0010
127	0000000001111111	007F
255	0000000011111111	00FF
256	0000000100000000	0100
32767	0111111111111111	7FFF

Complément à 2

- Même si ce n'est pas obligatoire, tous les systèmes actuels utilisent le complément à 2 pour représenter les entiers négatifs
  - Coder la valeur absolue en base 2
  - Inverser tous les bits (0 devient 1, 1 devient 0)
  - Ajouter 1
- En complément à 2 avec des entiers sur n bits, on peut coder les entiers négatifs de -2^{n-1} à -1

Note : les autres représentations diffèrent peu. Par exemple, une représentation signe + amplitude peut coder les entiers de $-(2^{n-1}-1)$ *à* $2^{n-1}-1$*. L'entier 0 utilise alors 2 représentations binaires*

Complément à 2 : entiers négatifs sur 16 bits

Entier	Binaire	Hexadécimal
-1	1111111111111111	FFFF
-2	1111111111111110	FFFE
-3	1111111111111101	FFFD
-4	1111111111111100	FFFC
-16	1111111111110000	FFF0
-255	1111111100000001	FF01
-256	1111111100000000	FF00
-32768	1000000000000000	8000

Types entiers signés en C++

- Le C++ propose 4 tailles d'entiers signés

  ```
  signed short int
  signed int
  signed long int
  signed long long int
  ```

- Les mots clés `signed` et `int` sont optionnels. L'ordre des éléments du type n'est pas fixé. `long long`, `long int long`, et `long long signed` par exemple sont des noms de type valables, synonymes de `signed long long int`

- La nombre de bits utilisés par ces types n'est pas garanti. Les seules garanties sont
  - `sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
  - `short` et `int` utilisent au moins 16 bits, `long` au moins 32, et `long long` au moins 64

- On peut également utiliser le type caractère `signed char` comme un entier sur 8 bits. Le mot clé `signed` n'est pas optionnel dans ce cas

sizeof

- L'opérateur `sizeof` retourne le nombre de **bytes** utilisés par le type ou l'expression en paramètre

- Syntaxe :

```
sizeof(type)
sizeof expression
```

- Utilisation :

```
cout << "s char    : " << sizeof(signed char) << endl;
cout << "short     : " << sizeof(short)       << endl;
cout << "int       : " << sizeof(int)         << endl;
cout << "long      : " << sizeof(long)        << endl;
cout << "long long : " << sizeof(long long)   << endl;
short int a = 42;
cout << "a         : " << sizeof a            << endl;
```

```
s char    : 1
short     : 2
int       : 4
long      : 8
long long : 8
a         : 2
```

Nombre de bits des types entiers

- En combinant `sizeof` et `CHAR_BIT`, on peut déterminer la taille en bits des types entiers

```
cout << "short     : " << sizeof(short) * CHAR_BIT     << endl;
cout << "int       : " << sizeof(int) * CHAR_BIT       << endl;
cout << "long      : " << sizeof(long) * CHAR_BIT      << endl;
cout << "long long : " << sizeof(long long) * CHAR_BIT << endl;
```

- Le résultat peut varier selon l'architecture cible. Par exemple, pour des systèmes 64 bits.

```
short     : 16
int       : 32
long      : 32
long long : 64
```

```
short     : 16
int       : 32
long      : 64
long long : 64
```

Modèles de données

4 modèles de données sont / ont été largement utilisés. Ils sont caractérisés par 3 chiffres : les nombres de bytes utilisés par les types `int` / `long` / pointeur

- **LP32** ou **2/4/4** - `int` 16-bit, `long` et pointeurs 32-bit : Win16 API
- **ILP32** ou **4/4/4** - `int`, `long`, et pointeurs 32-bit : Win32 API, Unix, Linux, macOS sur systèmes 32 bits
- **LLP64** ou **4/4/8** - `int` et `long` 32-bit, pointeurs 64-bit : Windows API pour systèmes x64 ou ARM64
- **LP64** ou **4/8/8** - `int` 32-bit, `long` et pointeurs 64-bit : Unix, Linux, macOS sur systèmes 64 bits

D’autres modèles existent mais sont très rares. Par exemple, **ILP64** ou **8/8/8** - `int`, `long`, et pointeurs 64-bit : UNICOS (super-ordinateurs Cray)

numeric_limits

- La manière la plus propre de connaitre les propriétés d’un type numérique est d’utiliser les méthodes de `std::numeric_limits` , provenant de la librairie `<limits>`

- Syntaxe :

```
std::numeric_limits<type>::lowest()     // plus petite valeur représentable
std::numeric_limits<type>::max()        // plus grande valeur représentable
std::numeric_limits<type>::digits       // nombre de bits hors bit de signe
std::numeric_limits<type>::is_signed    // vrai pour les types signés
std::numeric_limits<type>::is_integer   // vrai pour les types entiers
```

numeric_limits : utilisation

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

int main() {
   cout << " ** signed short int ** " << endl;
   cout << "lowest : " << std::numeric_limits<short>::lowest
   cout << "max    : " << std::numeric_limits<short>::max()
   cout << "digits : " << std::numeric_limits<short>::digits

   cout << "\n ** signed int ** " << endl;
   cout << "lowest : " << std::numeric_limits<int>::lowest() << endl;
   cout << "max    : " << std::numeric_limits<int>::max() << endl;
   cout << "digits : " << std::numeric_limits<int>::digits << endl;

   cout << "\n ** signed long long int ** " << endl;
   cout << "lowest : " << std::numeric_limits<long long>::lowest() << endl;
   cout << "max    : " << std::numeric_limits<long long>::max() << endl;
   cout << "digits : " << std::numeric_limits<long long>::digits << endl;
}
```

```
 ** signed short int **
lowest : -32768
max    : 32767
digits : 15

 ** signed int **
lowest : -2147483648
max    : 2147483647
digits : 31

 ** signed long long int **
lowest : -9223372036854775808
max    : 9223372036854775807
digits : 63
```

Dépassement

- Le résultat d'un calcul effectué avec un type donné peut ne pas être représentable dans ce type. On parle de dépassement
- Exemple avec `std::numeric_limits<int>::max()` qui vaut `2'147'483'647` :

```
int a = 2'000'000'000;
int b = 1'000'000'000;
int c = a + b;
cout << a << " + " << b << " = "
     << c << endl;
```

```
2000000000 + 1000000000 = -1294967296
```

- Selon le standard C++, un dépassement sur un calcul en type entier signé donne un résultat **non défini**, et donc inutilisable

Comportement indéfini ?

- En pratique, en complément à 2, il est très probable que

```
cout << numeric_limits<int>::max() + 1;
```

affiche

```
-2147483648
```

- Attention, on ne peut pas s'y fier ! Par exemple, le code suivant

```
int x = numeric_limits<int>::max();
cout << boolalpha << ( x < x + 1 ) << endl;
```

  - compilé en mode debug, affiche `false`
  - compilé en mode release, affiche `true`

- En effet, `x < x + 1` est vrai pour tout x sauf en cas de dépassement qui - n'étant pas défini pour x entier signé - peut être ignoré par l'optimisation

Prévenir un dépassement

- On peut vérifier si un calcul va dépasser avant de l'effectuer
- L'addition déborde si
  - Les 2 opérandes sont positives et le résultat est > `numeric_limits<int>::max()`
  - Les 2 opérandes sont négatives et le résultat est < `numeric_limits<int>::lowest()`
- Le code suivant vérifie ces deux possibilités

```
bool sum_would_overflow(int lhs, int rhs) {
   return (lhs >= 0) ?
          numeric_limits<int>::max() - lhs < rhs :
          rhs < numeric_limits<int>::lowest() - lhs;
}
```

- Soustraction et multiplication peuvent également déborder

Vol 501 d'Ariane 5

- Vol inaugural, 4 juin 1996
- Même système de guidage inertiel qu'Ariane 4
- Accélérations 5 fois plus fortes qu'Ariane 4
- Dépassement dans le calcul de la position à partir des accélérations
- L'ordinateur de bord ordonne un virage serré pour corriger la trajectoire
- L'accélération latérale arrache un des boosters latéraux
- La destruction automatique est engagée ☹

Entiers non signés

- A chacun des types d'entier signé correspond un type d'entier non signé de même taille en remplaçant le mot clé `signed` par `unsigned`

```
unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int
```

- Le mot clé `int` est optionnel. L'ordre des éléments du type n'est pas fixé. `long unsigned int long` est un nom de type valable, synonyme de `unsigned long long int`
- Un type `unsigned` de n bits peut représenter les entiers positifs de 0 à $2^n - 1$
- En mémoire, les bits correspondent à la représentation de l'entier en base 2

Limites typiques des entiers non signés

```
cout << "unsigned char      ( " << std::numeric_limits<unsigned char>::digits << " bits ) :  "
     << +std::numeric_limits<unsigned char>::lowest() << " -> "
     << +std::numeric_limits<unsigned char>::max() << endl;

cout << "unsigned short     ( " << std::numeric_limits<unsigned short>::digits << " bits ) : "
     << std::numeric_limits<unsigned short>::lowest() << " -> "
     << std::numeric_limits<unsigned short>::max() << endl;

cout << "unsigned int       ( " << std::numeric_limits<unsigned>::digits << " bits ) : "
     << std::numeric_limits<unsigned>::lowest() << " -> "
     << std::numeric_limits<unsigned>::max() << endl;

cout << "unsigned long long ( " << std::numeric_limits<unsigned long long>::digits << " bits ) : "
     << std::numeric_limits<unsigned long long>::lowest() << " -> "
     << std::numeric_limits<unsigned long long>::max() << endl;
```

```
unsigned char      ( 8 bits ) :  0 -> 255
unsigned short     ( 16 bits ) : 0 -> 65535
unsigned int       ( 32 bits ) : 0 -> 4294967295
unsigned long long ( 64 bits ) : 0 -> 18446744073709551615
```

Arithmétique modulo 2^n

- Il n'y a **pas de dépassement** dans les calculs effectués en `unsigned`
- Les opérateurs +, -, et * sont définis « modulo 2^n »

```
unsigned a32 = 2'000'000’000, b32 = 3'000'000'000;
unsigned long long a64 = a32, b64 = b32;
cout << a32 << " + " << b32 << " = " << a32 + b32
                             << " = " << a64 + b64 << " mod 4294967296";
```

```
2000000000 + 3000000000 = 705032704 = 5000000000 mod 4294967296
```

- En pratique, aucun calcul de modulo n'est effectué par le processeur. Il se contente d'ignorer les bits de report au-delà du n[ième] bit lors des calculs.
- L'`unsigned` le plus grand et 0 sont des entiers consécutifs

```
unsigned zero = 0; cout << zero - 1 << endl;
cout << numeric_limits<unsigned>::max() + 1 << endl;
```

```
4294967295
0
```

typedef

- Le spécificateur `typedef` permet de définir des **alias de type**, i.e. de nouveaux noms de types synonymes de types existants

- Syntaxe :

  `typedef` *`type_existant nouveau_type`*`;`

- Exemple :

```
typedef int Entier;
Entier entier = 42;

typedef double Reel;
Reel pi = 3.141592;
```

using

- Le spécificateur `using` permet une syntaxe alternative à `typedef`
- Syntaxe :

```
using nouveau_type = type_existant;
```

- Exemple :

```
using Entier = int;
Entier entier = 42;

using Reel = double;
Reel pi = 3.141592;
```

- Cette syntaxe est préférable en C++ moderne, `typedef` étant un reliquat du langage C

Entiers signés de taille fixe de <cstdint>

- Le header `<cstdint>` fournit les alias de type suivants

`int8_t`
`int16_t`
`int32_t`
`int64_t`
	• Entier signé dont la **taille est exactement** 8, 16, 32, 64 bits.
	• Utilisent le complément à 2 pour les entiers négatifs
	• Potentiellement non définis si aucun type parmi `signed char`, `short`, `int`, `long` et `long long` ne respecte ces contraintes

`int_fast8_t`
`int_fast16_t`
`int_fast32_t`
`int_fast64_t`
	• Entier signé dont la taille est au moins 8, 16, 32, 64 bits.
	• Le type le **plus rapide** à l'exécution respectant cette contrainte

`int_least8_t`
`int_least16_t`
`int_least32_t`
`int_least64_t`
	• Entier signé dont la taille est au moins 8, 16, 32, 64 bits.
	• Le type le **plus petit** respectant cette contrainte

`intmax_t`	• Entier signé dont la taille est **la plus grande** possible
`intptr_t`	• Entier signé dont la taille permet de **stocker un pointeur** • Potentiellement non défini si aucun type existant ne respecte cette contrainte

Entiers non signés de taille fixe

- Le header `<cstdint>` fournit les alias de type suivants pour les types non signés

`uint8_t`
`uint16_t`
`uint32_t`
`uint64_t`
	• Entier non signé dont la **taille est exactement** 8, 16, 32, 64 bits.
	• Potentiellement non définis si aucun type parmi `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` et `unsigned long long` ne respecte ces contraintes

`uint_fast8_t`
`uint_fast16_t`
`uint_fast32_t`
`uint_fast64_t`
	• Entier non signé dont la taille est au moins 8, 16, 32, 64 bits.
	• Le type le **plus rapide** à l'exécution respectant cette contrainte

`uint_least8_t`
`uint_least16_t`
`uint_least32_t`
`uint_least64_t`
	• Entier non signé dont la taille est au moins 8, 16, 32, 64 bits.
	• Le type le **plus petit** respectant cette contrainte

`uintmax_t`	• Entier non signé dont la taille est **la plus grande** possible
`uintptr_t`	• Entier non signé dont la taille permet de **stocker un pointeur** • Potentiellement non défini si aucun type existant ne respecte cette contrainte

`std::size_t`

- Type de retour de l'opérateur **`sizeof`**
- Alias d'un type **entier non signé** suffisamment grand pour **stocker la taille** de tout objet en C++ y compris des tableaux
- Défini dans de nombreux headers : `<cstdef>, <cstdio>, <cstdlib>, …`
- Peut-être synonyme de `unsigned int` (e.g. win32), `unsigned long` (e.g. macOS) ou `unsigned long long` (e.g. win64)
- Type des **indices** pour accéder aux éléments d'un `std::array`, d'un `std::vector`, ou aux caractères d'une `std::string`

Entiers littéraux

- Les entiers littéraux permettent d'écrire une valeur entière dans le code. Ils sont constitués, sans caractère blanc intercalaire, de
  - Un **préfixe** optionnel spécifiant la base (2, 8, 10 ou 16)
  - Une suite de **chiffres** dans cette base (0 ou 1 en binaire, 0 à 7 en octal, 0 à 9 en décimal, 0 à F en hexadécimal, casse au choix pour les chiffres A à F)
  - Un **suffixe** optionnel spécifiant le type (`signed / unsigned` et `int / long / long long`)
  - On peut intercaler le caractère `'` entre les chiffres. Il ne change pas la valeur mais peut aider à rendre la valeur numérique plus lisible
- Exemples :

```
int un_million = 1'000'000;
long vingt_six = 032L;
unsigned long long quarante_deux = 0x00'00'00'2Aull;
```

Préfixe de base

- Les entiers littéraux peuvent être précédés d'un préfixe qui spécifie la base

`0b` ou `0B`	Binaire (2)
`0`	Octal (8)
Pas de préfixe	Décimal (10)
`0x` ou `0X`	Hexadécimal (16)

- Exemple :

```
cout << 0b10 << ' ' << 010 << ' ' << 10 << ' ' << 0x10 << endl;

cout << 0b101010 << ' ' << 052 << ' ' << 42 << ' ' << 0x2A ' ' << 0x2a ;
```

```
2 8 10 16
42 42 42 42 42
```

Suffixes de type

- Les entiers littéraux peuvent être suivis d'un suffixe qui spécifie le type

Pas de suffixe	`signed int`
`L`	`signed long int`
`LL`	`signed long long int`
`U`	`unsigned int`
`UL`	`unsigned long int`
`ULL`	`unsigned long long int`

- Minuscule ou majuscule à choix
- Ordre quelconque entre U et L / LL. Ainsi, `ull`, `uLL`, `llu`, `LLu`, `Ull`, `ULL`, `llU`, et `LLU` sont des suffixes valides pour `unsigned long long`

Note : C++23 introduit le suffixe `UZ` *pour le type* `size_t`

Autres suffixes de type

Note : Les suffixes de type sont également utilisés dans la STL. Par exemple, la librairie `<chrono>` *définit les suffixes* `ns, us, ms, s, min`*, et* `h`*.*

Ils spécifient non seulement le type mais aussi comment interpréter la valeur numérique qui les précèdent

```
chrono::duration seconde = 1'000'000'000ns;
seconde = 1'000'000us;
seconde = 1'000ms;
chrono::duration minute = 60s;
chrono::duration heure = 60min;
chrono::duration jour = 24h;
```

Type d’un entier littéral

- Le suffixe de type (ou son absence) ne spécifie que la taille minimale du type de l’entier.
- Si l’entier littéral est trop grand par rapport à ce type, le plus petit type capable de le contenir sera sélectionné.
- Exemples (en supposant que le type `long` utilise 64 bits) :

```
auto a = 1'000'000'000;       // int
auto b = 5'000'000'000;       // long
auto c = 0x123456789ABCDEF0;  // long
auto d = 0xFEDCBA9876543210;  // unsigned long

// auto e = 10'000'000'000'000'000'000'000;
// ne compile pas, la valeur littérale étant trop
// grande être représentable dans tout type entier.
```

Affichage des entiers

- `<iomanip>` fournit divers **modificateurs de flux** dédiés à l'affichage d'entiers
  - **`oct`, `dec`, `hex`** spécifient la base octale, décimale ou hexadécimale
  - **`setbase(n)`** avec `n = 8, 10`, ou `16` fait de même. Toute autre valeur de n rétablit l'affichage par défaut (décimal)
  - **`showbase`** ajoute `0 / 0x` devant les chiffres lors de l'affichage en octal / hexadécimal. **`noshowbase`** (par défaut) l'annule.
  - **`uppercase`** affiche les chiffres hexadécimaux `A à F` en majuscule. **`nouppercase`** (par défaut) les affiche en minuscule
  - **`showpos`** ajoute le signe + devant les entiers positifs affichés en décimal. **`noshowpos`** (par défaut) l'annule
- Les modificateurs de flux spécifiant la base (`oct`, `dec`, `hex`, et `setbase(n)`) fonctionnent également pour la **lecture** d'entiers

Affichage en décimal

- Affichage par défaut, ou via les modificateurs **`dec`** ou **`setbase(n)`** avec n différent de 8 ou 16.
- **`showbase`**, **`noshowbase`**, **`uppercase`**, et **`nouppercase`** n'ont pas d'effet
- **`showpos`** ajoute le symbole + devant les entiers positifs, **`noshowpos`** (par défault) le supprime

```
cout << 42 << " " << -42 << endl;
cout << showpos;
cout << 42 << " " << -42 << endl;
cout << noshowpos;
cout << 42 << " " << -42 << endl;
cout << hex << showbase << 42 << " ";
cout << dec << -42 << endl;
```

```
42 -42
+42 -42
42 -42
0x2a -42
```

Affichage en octal

- Affichage via les modificateurs **`oct`** ou **`setbase(8)`**
- **`showbase`** ajoute le symbole `0` devant les chiffres , **`noshowbase`** (par défaut) le supprime
- Pour les entiers négatifs, c'est leur motif binaire en mémoire qui est affiché, pas leur valeur signée.
- **`showpos`**, **`noshowpos`**, **`uppercase`** et **`nouppercase`** n'ont pas d'effet

```
cout << 42 << " " << -42 << endl;
cout << oct;
cout << 42 << " " << -42 << endl;
cout << showbase;
cout << 42 << " " << -42 << endl;
cout << noshowbase;
cout << 42 << " " << -42 << endl;
```

```
42 -42
52 37777777726
052 037777777726
52 37777777726
```

Affichage en hexadécimal

- Affichage via les modificateurs **`hex`** ou **`setbase(16)`**
- **`showbase`** ajoute les symboles `0x` devant les chiffres, **`noshowbase`** (par défaut) les supprime
- **`uppercase`** et **`nouppercase`** changent la casse des chiffres `A` à `F`
- Pour les entiers négatifs, c'est leur motif binaire en mémoire qui est affiché, pas leur valeur signée
- **`showpos`**, et **`noshowpos`** n'ont pas d'effet

```
cout << 42 << " " << -42 << endl;
cout << hex;
cout << 42 << " " << -42 << endl;
cout << showbase;
cout << 42 << " " << -42 << endl;
cout << uppercase;
cout << 42 << " " << -42 << endl;
cout << noshowbase << nouppercase;
cout << 42 << " " << -42 << endl;
```

```
42 -42
2a ffffffd6
0x2a 0xffffffd6
0X2A 0XFFFFFFD6
2a ffffffd6
```

Lecture avec `setbase`

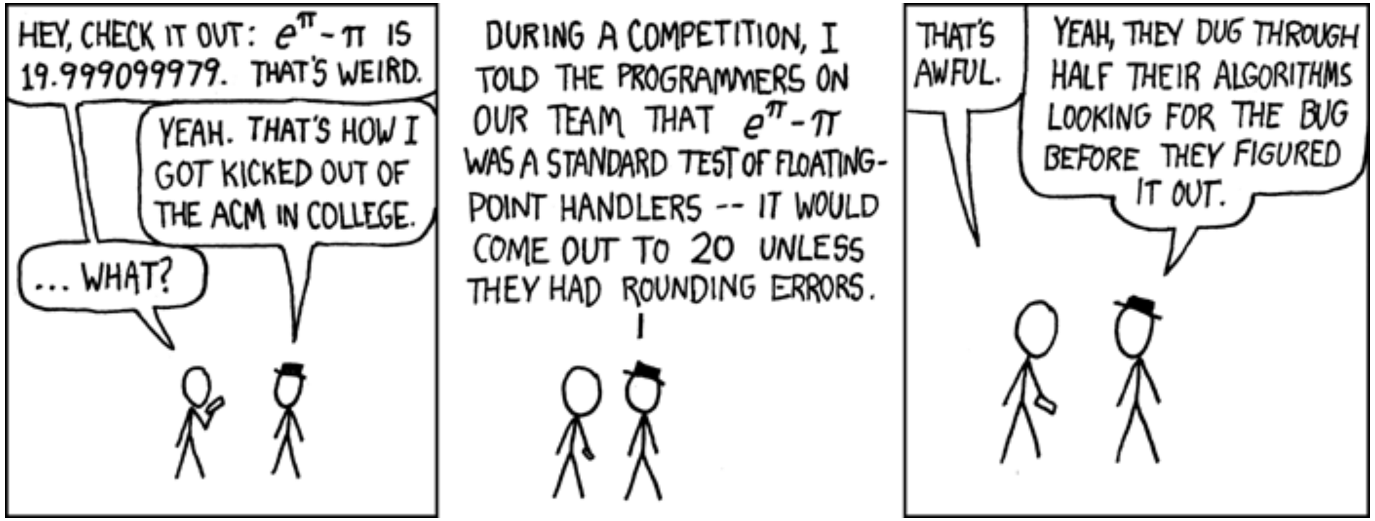
- **`setbase(n)`**, **`oct`**, **`hex`**, et **`dec`** affectent la manière dont le flux d'entrée est interprété lors de la lecture d'une variable entière
  - **`dec`** ou **`setbase(10)`** : par défaut, entrée décimale
  - **`oct`** ou **`setbase(8)`** : entrée octale
  - **`hex`** ou **`setbase(16)`** : entrée hexadécimale
- Dans ces bases, le symbole – est interprété comme un nombre négatif, le symbole + (optionnel) comme positif. Le préfixe de cette base est le seul reconnu. La lecture s'arrête au premier caractère qui n'est pas un chiffre de la base choisie.
- **`setbase(0)`** ou tout autre valeur différente de 8, 10 ou 16 : prise en compte des préfixes `0` ou `0x` ou de leur absence pour déterminer la base. Le nombre lu doit être suivi d'un caractère blanc

```
int e; cin >> setbase(n) >> e;
```

Valeur de n	Entrée utilisateur	Valeur de e
`10`	`-10a`	`-10`
	`+010a`	`10`
	`0x10`	`0`
`8`	`-10a`	`-8`
	`+010a`	`8`
	`0x10`	`0`
`16`	`-10a`	`-266`
	`+010a`	`266`
	`0x10`	`16`
`0`	`-10`	`-10`
	`+010`	`8`
	`0x10`	`16`
	`10a`	`0`

Réels

- Représentation en mémoire : IEEE 754, zéro, nombres dénormalisés, infini, NaN.
- Types réels en C++ : `float, double, long double`
- `std::numeric_limits : radix, digits, min(), denorm_min(), lowest(), max(), epsilon(), digits10, max_digits10`
- Entrées / sorties réelles : `fixed, scientific, defaultfloat, hexfloat, showpoint, setprecision(n)`
- Réels littéraux
- `<cmath>`



https://xkcd.com/217/

Types flottants

- Le concept de **virgule flottante** permet de représenter de **manière approchée** une large plage de nombres réels
- Il consiste à représenter le réel r sous la forme $r = (-1)^s \cdot m \cdot b^e$, avec
  - b la **base** entière, typiquement 2 ou 10
  - s le **signe** binaire $\in \{0,1\}$
  - e l'**exposant** entier
  - m la **mantisse** réelle, avec $1 \leq m < b$ dans sa forme normalisée pour qu'une seule paire (m, e) code le même r.
- Par exemple, le réel $r = 314{,}2$ peut être représenté comme
  - $r = +\,3{,}142 \cdot 10^2$ en base $b = 10$
  - $r = +\,1{,}22734375 \cdot 2^8$ en base $b = 2$

Représentation binaire du type flottant

- Pour stocker $r = (-1)^s \cdot m \cdot b^e$ sous forme binaire, on le code en l'approximant uniquement avec des **entiers positifs**
  - s est un entier sur 1 bit $\in \{0,1\}$
  - e via un entier positif E dont on soustrait un biais constant B

$$e = E - B \qquad\qquad E = e + B$$

  - m via un entier positif M avec p chiffres en base b, i.e. $0 \leq M < b^p$

$$m \approx \frac{M}{b^{p-1}} < b \qquad\qquad M = m \cdot b^{p-1}$$

- $(-1)^s \cdot M \cdot b^{E-B-p+1}$ est une **approximation** de la valeur de r. Avec p chiffres en base b pour coder la mantisse, il peut y avoir une **erreur relative** entre la valeur codée et le réel r de l'ordre de $\epsilon = \frac{1}{b^{p-1}}$

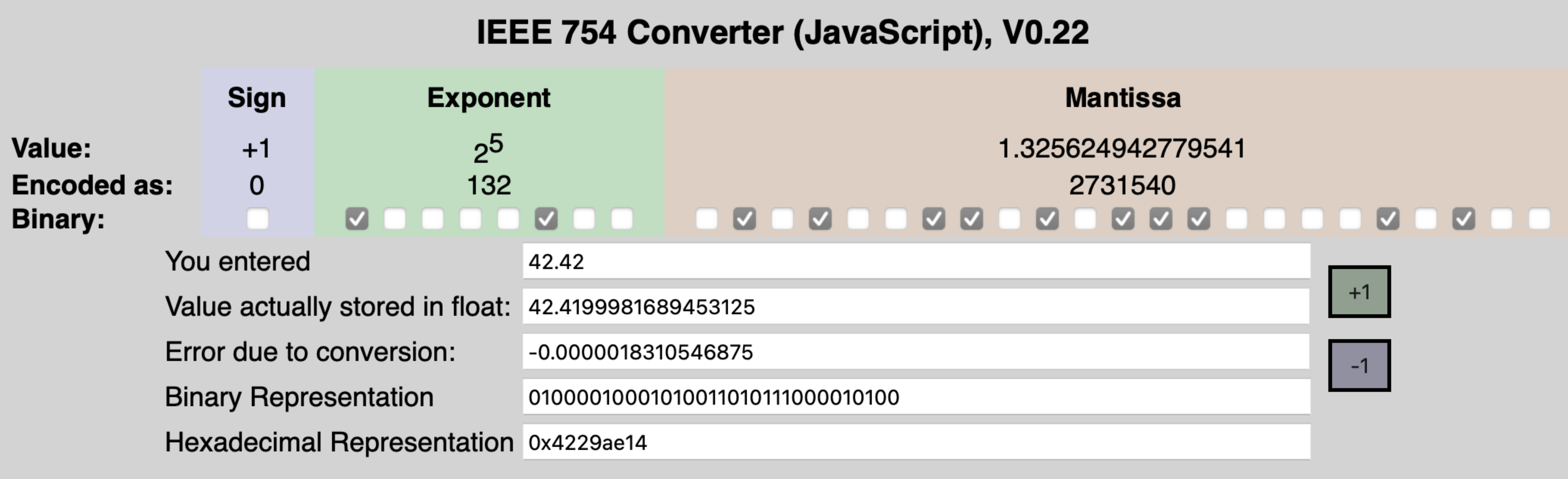
IEEE 754 : IEEE Standard for Floating-Point Arithmetic

- Etabli et mis à jour depuis 1985 par l'IEEE : Institute of Electrical and Electronics Engineers
- Définit les standards d'arithmétique en virgule flottante `binary16`, `binary32`, `binary64`, `binary128`, `binary256`, `decimal32`, `decimal64`, et `decimal128`
- Chacun de ces standards spécifie notamment
  - La base b : 2 ou 10
  - Le nombre de bits utilisés en tout : 16, 32, 64, 128, ou 256
  - La précision p de chiffres en base b utilisés par la mantisse entière M
  - La manière de coder les chiffres décimaux en bits pour les standards `decimal`…
  - Le biais B et donc la plage d'exposants e utilisables avec les bits restants

Exemple : r = 42.42 en IEEE 754 binary32

- Mantisse avec précision $p = 24$ bits codée sur 23 bits, le premier valant toujours 1 quand la mantisse est normalisée : $1 \leq m < 2$, donc $2^{23} \leq M < 2^{24}$ et on code $0 \leq M - 2^{23} < 2^{23}$
- Exposant codé sur 8 bits, moins le biais $B = 127$
- $42.42 \approx +1 \cdot 2^{132-127} \cdot (1 + \frac{2731540}{2^{23}})$

https://www.h-schmidt.net/FloatConverter/IEEE754.html



Zéro et les nombres dénormalisés

- La formule $r = (-1)^s \cdot m \cdot b^e$ ne permet pas trouver une mantisse $1 \leq m < b$ normalisée pour $r = 0$
- Par convention, pour le plus petit exposant e, i.e. pour $E = 0$, la mantisse est non normalisée, son premier bit ne vaut pas implicitement 1, et $r = (-1)^s \cdot M \cdot b^{1-B-p+1}$
- Le réel $r = 0$ est codé par $E = 0$ et $M = 0$. Le signe peut varier. +0 et -0 sont des nombres différents (en représentation binaire) mais égaux (en valeur)

```
double r = 0.;
cout << r << endl;

r *= -1.;
cout << r << endl;

cout << boolalpha << ( 0. == -0. ) << endl;
```

```
0
-0
true
```

∞ : infinité

- A l’opposé, le plus grand exposant e ($E_{max} = 2^q - 1$ s’il est codé sur q bits) et $M = 0$ est utilisé pour coder une valeur **infinie**, qui peut être positive ou négative selon le bit de signe.

- Divers calculs peuvent donner un résultat infini

```
cout << 1./0.   << endl;
cout << log(0.) << endl;
```

```
inf
-inf
```

- Les types C++ supportant cette notion d’infini ont

```
numeric_limits<type>::has_infinity == true
```

- On peut directement accéder à la valeur infinie via

```
cout << numeric_limits<type>::infinity();
```

```
inf
```

NaN : not a number

- Enfin, le plus grand exposant e et $M \neq 0$ codent « **not a number** », qui correspond au résultat de calculs impossibles ou indéterminés en nombres réels

```
cout << 0./0. << endl;
cout << sqrt(-1) << endl;
cout << 1./0. - 1./0. << endl;
```

```
nan
nan
nan
```

- Les types C++ supportant NaN ont

```
numeric_limits<type>::has_quiet_NaN == true
numeric_limits<type>::has_signaling_NaN == true
```

- Les **comparaisons** impliquant un NaN sont particulières, NaN n'étant pas ordonné par rapport aux autres réels
  - <, >, <=, >= et == retournent toujours `false` dès qu'au moins un opérande est NaN.
  - != retourne toujours `true`.

Les types réels en C++

- C++ supporte trois longueurs de nombres réels via les 3 types suivants

`float`	simple précision
`double`	double précision
`long double`	précision étendue

- `float` et `double` correspondent aux formats `IEEE-754 binary32` et `binary64` s'ils sont supportés par l'architecture du processeur

Type et nombre de bits		Signe	Mantisse	Exposant
`float`	32	1	23	8
`double`	64	1	52	11

- `long double` peut utiliser 64, 80 ou 128 bits selon la mise en œuvre

numeric_limits<T> radix / is_signed / digits

- Ces infos sont disponibles via `std::numeric_limits`
- `radix` retourne la base, toujours 2 pour les types C++ prédéfinis
- `digits` retourne la précision p de la mantisse, soit le nombre de bits utilisés pour la mantisse + 1 bit implicite en représentation normalisée

```
int total = sizeof(type) * CHAR_BIT;
int base = numeric_limits<type>::radix;
int s = (int) numeric_limits<type>::is_signed;
int p = numeric_limits<type>::digits;

cout << "Base : " << base
     << "\nBits : "      << total
     << "\nSigne : "     << s
     << "\nMantisse: "   << p-1
     << "\nExposant: "   << total - (p-1) - s;
     << "\nPrécision: " << p;
```

```
using type = float;
```

```
Base : 2
Bits : 32
Signe : 1
Mantisse: 23
Exposant: 8
Précision: 24
```

```
using type = double;
```

```
Base : 2
Bits : 64
Signe : 1
Mantisse: 52
Exposant: 11
Précision: 53
```

numeric_limits<T> min() / denorm_min()

- Le **plus petit** réel strictement positif représentable avec la formule **normalisée** utilise $M = 0$ et $\mathrm{E} = 1$, il vaut donc $r = 2^{1-B}$, soit 2^{-126} en `float` et 2^{-1022} en `double`

```
cout << "min: " << numeric_limits<type>::min() << endl;
```

float
```
min: 1.17549e-38
```

double
```
min: 2.22507e-308
```

- Le **plus petit** réel strictement positif représentable avec la formule **dénormalisée** utilise $M = 1$ et $E = 0$ et, il vaut donc $r = 2^{1-B-p+1}$, soit 2^{-149} en `float` et 2^{-1074} en `double`

```
cout << "denorm_min: " << numeric_limits<type>::denorm_min() << endl;
```

float
```
denorm_min: 1.4013e-45
```

double
```
denorm_min: 4.94066e-324
```

numeric_limits<T> lowest() / max()

- Le **plus grand** réel **fini** représentable utilise $m \approx 2$ et $e = E_{max} - 1 - B$, il vaut donc $r \approx 2^{E_{max}-B}$ et vaut un peu moins de 2^{128} en `float` et 2^{1024} en `double`

```
cout << "max: " << numeric_limits<type>::max() << endl;
```

`float`
```
max: 3.40282e+38
```

`double`
```
max: 1.79769e+308
```

- Le **plus petit** réel **fini** représentable utilise les mêmes m et e que le plus grand, mais le signe négatif. Il vaut donc un peu plus de -2^{128} en `float` et -2^{1024} en `double`

```
cout << "lowest: " << numeric_limits<type>::lowest() << endl;
```

`float`
```
lowest: -3.40282e+38
```

`double`
```
lowest: -1.79769e+308
```

numeric_limits<T> epsilon()

- Le nombre de bits utilisés par la mantisse impacte la **précision relative** avec laquelle on peut représenter les nombres / effectuer les calculs

- On caractérise cette précision par la différence entre le plus petit réel $r > 1$ représentable dans le type utilisé ($M = 1$ et $e = 0$), et le réel 1 ($M = 0$ et $e = 0$), ce qui vaut $\epsilon = 2^{-p+1}$, soit 2^{-23} en `float` et 2^{-52} en `double`

```
cout << "epsilon: " << numeric_limits<type>::epsilon() << endl;
```

float
```
epsilon: 1.19209e-07
```

double
```
epsilon: 2.22045e-16
```

- Cela signifie qu'il n'y a aucun `float` dont la valeur est strictement comprise entre 1 et 1.00000011920928955078125

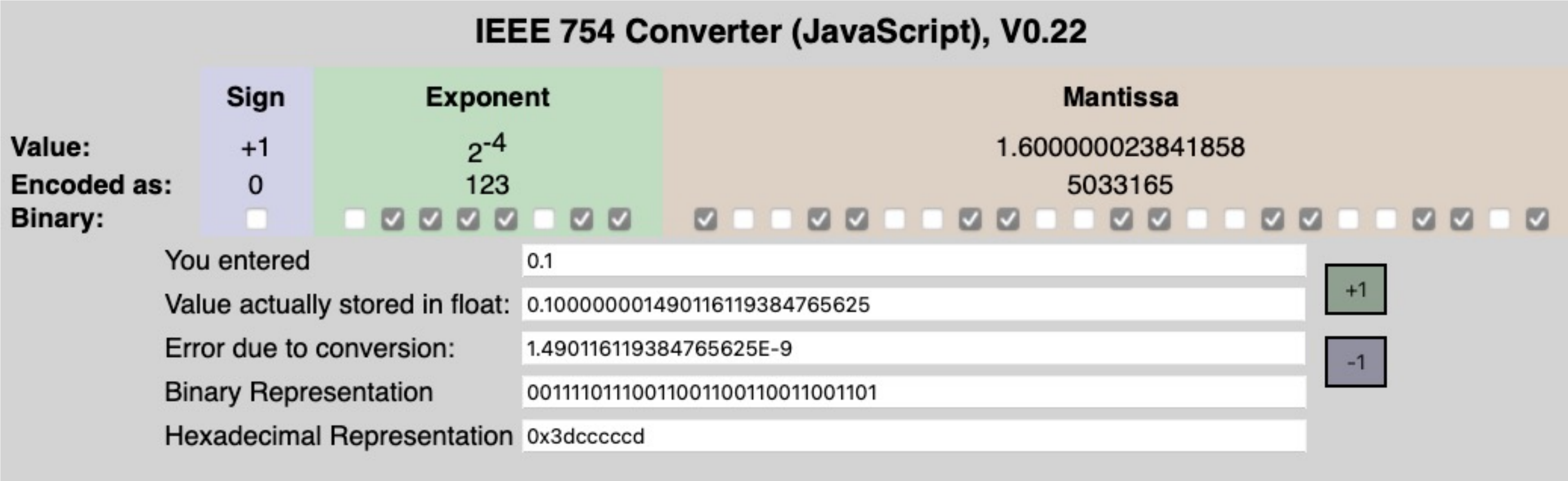
Egalité vs. « presque égalité »

```
bool almost_equal(double x, double y, int ulp) {

   // epsilon doit être mis à l'échelle en fonction de l'amplitude
   // des valeurs utilisées et multiplié par la précision souhaitée en ULP
   // (unités à la dernière place)

   return fabs(x - y) <= numeric_limits<double>::epsilon() * fabs(x + y) * ulp

   // sauf si le résultat est un nombre dénormalisé

          or std::fabs(x - y) < numeric_limits<double>::min();
}

int main() {
   double d1 = 1. / 5.;
   double d2 = 1. / std::sqrt(5.) / std::sqrt(5.);
   cout << fixed << setprecision(20) << "d1=" << d1 << "\nd2=" << d2 << '\n’;
   cout << ((d1 == d2) ? "d1 == d2" : "d1 != d2") << endl;
   cout << (almost_equal(d1, d2, 2) ? "d1 ~ d2\n" : "d1 !~ d2\n") << endl;
}
```

```
d1=0.20000000000000001110
d2=0.19999999999999998335
d1 != d2
d1 ~ d2
```

numeric_limits<T> digits10() / max_digits10()

- Les réels sont stockés et les calculs sont effectués en base 2
- Ils sont typiquement lus et affichés en base 10
- La conversion entre ces 2 bases n’est pas toujours sans erreur avec un nombre de chiffres limités. Par exemple, la représentation binaire du réel 0.1



numeric_limits<T> digits10() / max_digits10()

- `digits10()` donne le nombre maximal de chiffres décimaux significatifs pour lequel la conversion **texte** → **réel** → **texte** , i.e. la conversion décimal → binaire → décimal est toujours exacte.

```
cout << "digits10: " << numeric_limits<type>::digits10 << endl;
```

float	`digits10: 6`	double	`digits10: 15`

- `max_digits10()` donne le nombre minimum de chiffres décimaux significatifs nécessaires dans le texte intermédiaire pour garantir que la conversion **réel** → **texte** → **réel** soit exacte.

```
cout << "max_digits10: " << numeric_limits<type>::max_digits10 << endl;
```

float	`max_digits10: 9`	double	`max_digits10: 17`

Affichage d’un réel dans un flux

`<iomanip>` fournit divers modificateurs de flux dédiés à **l’affichage** de réels

- **`fixed`**, **`scientific`**, **`defaultfloat`**, **`hexfloat`** spécifient le type d’affichage
- **`setprecision(n)`** indique le nombre de chiffres à afficher. n vaut 6 par défaut. Le dernier chiffre est arrondi au plus proche.
- **`showpoint`** force l’affichage des chiffres après la virgule, même s’ils valent 0. **`noshowpoint`** (par défaut) l’annule.
- **`showpos`** / **`noshowpos`** (par défaut) fonctionnent comme pour les entiers
- **`uppercase`** / **`nouppercase`** (par défaut) affectent la casse du `E` en notation scientifique et les chiffres hexadécimaux en `hexfloat`

`cout.precision()` pour `cout` ou la méthode **`precision()`** du flux utilisé en donne la précision courante.

Affichage en `fixed`

- Affiche tous les chiffres avant la virgule
- Affiche `cout.precision()` chiffres après la virgule
- `showpoint` / `noshowpoint` et `uppercase` / `nouppercase` n'ont pas d'effet

```cpp
double pi = 3.14159265358979323846264338327950;
double one_billion_billions = 1e18;
cout << fixed << pi << " " << one_billion_billions << endl;
cout << setprecision(15) << pi << " " << one_billion_billions << endl;
cout << noshowpoint << pi << " " << one_billion_billions << endl;
```

```
3.141593 1000000000000000000.000000
3.141592653589793 1000000000000000000.000000000000000
3.141592653589793 1000000000000000000.000000000000000
```

Affichage en scientific

- Affiche un réel dans l'intervalle [1,10[en notation fixed
- suivi de e (ou E en uppercase)
- suivi de la puissance de 10 signée par laquelle il faut multiplier le réel précédent pour obtenir le nombre à afficher
- showpoint / noshowpoint n'ont pas d'effet

```
double pi = 3.14159265358979323846264338327950;
double billion = 1e9;
double billionth_of_pi = pi / 1e9;
cout << scientific;
cout << pi << " " << billion << " " << billionth_of_pi << endl;
cout << setprecision(3) <<  pi << " " << billion << " " << billionth_of_pi  << endl;
cout << uppercase <<  pi << " " << billion << " " << billionth_of_pi  << endl;
```

```
3.141593e+00 1.000000e+09 3.141593e-09
3.142e+00 1.000e+09 3.142e-09
3.142E+00 1.000E+09 3.142E-09
```

Affichage en `defaultfloat`

- Affiche comme `scientific` pour les nombres très grands ($r \geq 10^p$ pour une précision p) ou très petits (typiquement $r < 10^{-4}$)
- Affiche comme `fixed` pour les nombres intermédiaires, mais sans les 0 finaux après la virgule.
- La gestion de la précision diffère de `fixed` et `scientific`. Ici, c'est le nombre total de chiffres significatifs qui compte, y compris avant la virgule, mais pas les zéros initiaux

```
cout << setprecision(4);
for(double r = 3.141592e-6; r < 1e6; r *= 10)
   cout << r << endl;
```

```
3.142e-06
3.142e-05
0.0003142
0.003142
0.03142
0.3142
3.142
31.42
314.2
3142
3.142e+04
3.142e+05
```

showpoint / noshowpoint

- En mode `defaultfloat`, `showpoint` force à toujours afficher le point décimal, suivi éventuellement de zéros si nécessaire pour atteindre la précision d'affichage voulue

```
cout << setprecision(5) << left;
for (double r = 3.14e-6; r < 1e7; r *= 10)
   cout << showpoint << setw(12) << r
        << " " << noshowpoint << r << endl;
```

```
3.1400e-06   3.14e-06
3.1400e-05   3.14e-05
0.00031400   0.000314
0.0031400    0.00314
0.031400     0.0314
0.31400      0.314
3.1400       3.14
31.400       31.4
314.00       314
3140.0       3140
31400.       31400
3.1400e+05   3.14e+05
3.1400e+06   3.14e+06
```

setprecision(n)

- Précise le nombre de chiffres décimaux à afficher
  - après la virgule en `fixed` et `scientific`
  - avant et après la virgule en `defaultfloat`, sans compter les zéros initiaux, et en n'affichant pas les zéros finaux après la virgule si `noshowpoint`
- Le dernier chiffre est arrondi au plus proche
- Des chiffres supplémentaires peuvent apparaitre quand la précision demandée est plus grande que celle du type utilisé. Ils correspondent à la conversion en décimal de la représentation binaire.

```
float pi10 = 31.415926;
cout << cout.precision() << endl
     << scientific << pi10 << endl
     << fixed << pi10 << endl
     << defaultfloat << pi10 << endl
     << setprecision(4) << pi10 << endl
     << setprecision(30) << pi10 << endl
     << showpoint << pi10 << endl;
```

```
6
3.141593e+01
31.415926
31.4159
31.42
31.4159259796142578125
31.4159259796142578125000000000
```

hexfloat

- Affiche la représentation en mémoire du réel $r = S \cdot 2^{E-B} \cdot (1 + \frac{M}{2^m})$
  - signe – ou + (optionnel)
  - préfixe `0x`
  - La mantisse M sous la forme `1.abc` avec `abc` la représentation hexadécimale des m bits de mantisse, avec éventuellement des bits finaux à 0 ajoutés pour obtenir le plus petit multiple de `4` $\geq\ m$. (donc 1 pour `float`, 0 pour `double`)
  - la lettre p
  - L'exposant $E - B$ signé, affiché en décimal avec son signe toujours présent
- `showpos`, `showpoint` s'appliquent
- `setprecision` n'a pas d'influence

hexfloat

- zéro s’affiche `0x0p+0`
- les nombres dénormalisés s’affichent comme s’ils étaient normaux

```
cout << hexfloat << 0.f << " " << -0.f << endl
     << 1.f << " " << -1.f << endl
     << numeric_limits<float>::denorm_min() << endl
     << numeric_limits<float>::min() << endl
     << numeric_limits<float>::lowest() << endl
     << numeric_limits<float>::max() << endl
     << numeric_limits<float>::epsilon() << endl;

for (float r = 0; r < 10; ++r)
   cout << int(r) << " " << r << endl;
```

```
0x0p+0 -0x0p+0
0x1p+0 -0x1p+0
0x1p-149
0x1p-126
-0x1.fffffep+127
0x1.fffffep+127
0x1p-23
0 0x0p+0
1 0x1p+0
2 0x1p+1
3 0x1.8p+1
4 0x1p+2
5 0x1.4p+2
6 0x1.8p+2
7 0x1.cp+2
8 0x1p+3
9 0x1.2p+3
```

Lecture d'un réel depuis un flux

- Tous les formats d'affichage précédents peuvent être lus depuis un flux d'entrée
- Les modificateurs de flux n'ont pas d'influence
- La lecture depuis un format décimal implique une conversion en format binaire. Le nombre stocké n'est donc pas toujours exactement celui entré par l'utilisateur

```cpp
double a, b, c, d;
stringstream in("-1.0e2 1e-2 3.141592 0x1.5p5");
in >> a >> b >> c >> d;
cout << a << " " << b << " " << c << " " << d << endl;
cout << setprecision(80) << c << endl;
```

```
-100 0.01 3.14159 42
3.14159200000000016217427400988526642322540283203125
```

Réels littéraux

- Tous les formats d'affichage peuvent être utilisés pour des constantes littérales réelles.
- Le type par défaut est `double`.
- Les suffixes F ou f (resp. L ou l) spécifient une constante littérale de type `float` (resp. `long double`)
- Attention, il est nécessaire qu'au moins un élément permette de savoir qu'il ne s'agit pas d'un entier littéral : `.` / `e` / `0x…p`
- Un réel littéral qui déborde vaut l'infini

```
auto r1 = 2.;           // double
auto r2 = 2e+1;         // double
auto r3 = -0x1.p2f;     // float
auto r4 = 3.14L;        // long double
auto r5 = 0x2p2;        // double
auto r6 = 3.1E-4L;      // long double

auto e1 = 3;            // int
auto e2 = 314L;         // long int
auto e3 = 0x2f;         // int (47)
// auto nc1 = 314F ne compile pas

auto r7 = 1e600000;     // double (inf)
auto r8 = -1e600000;    // double (-inf)
```

Expressions mathématiques

- Comment écrire cette formule en C++ ?

$$b+\left(1+\frac{r}{100}\right)^{n}$$

- La partie entre parenthèses s'écrit simplement `( 1 + r / 100 )`
- Mais comment écrire « à la puissance n » ?
  - C++ ne propose pas d'opérateur puissance
  - La librairie `<cmath>` propose une fonction `pow(base, exposant)`

```
#include <cmath>
using namespace std;
 …
double resultat = b + pow(1 + r / 100, n);
```

#include <cmath>

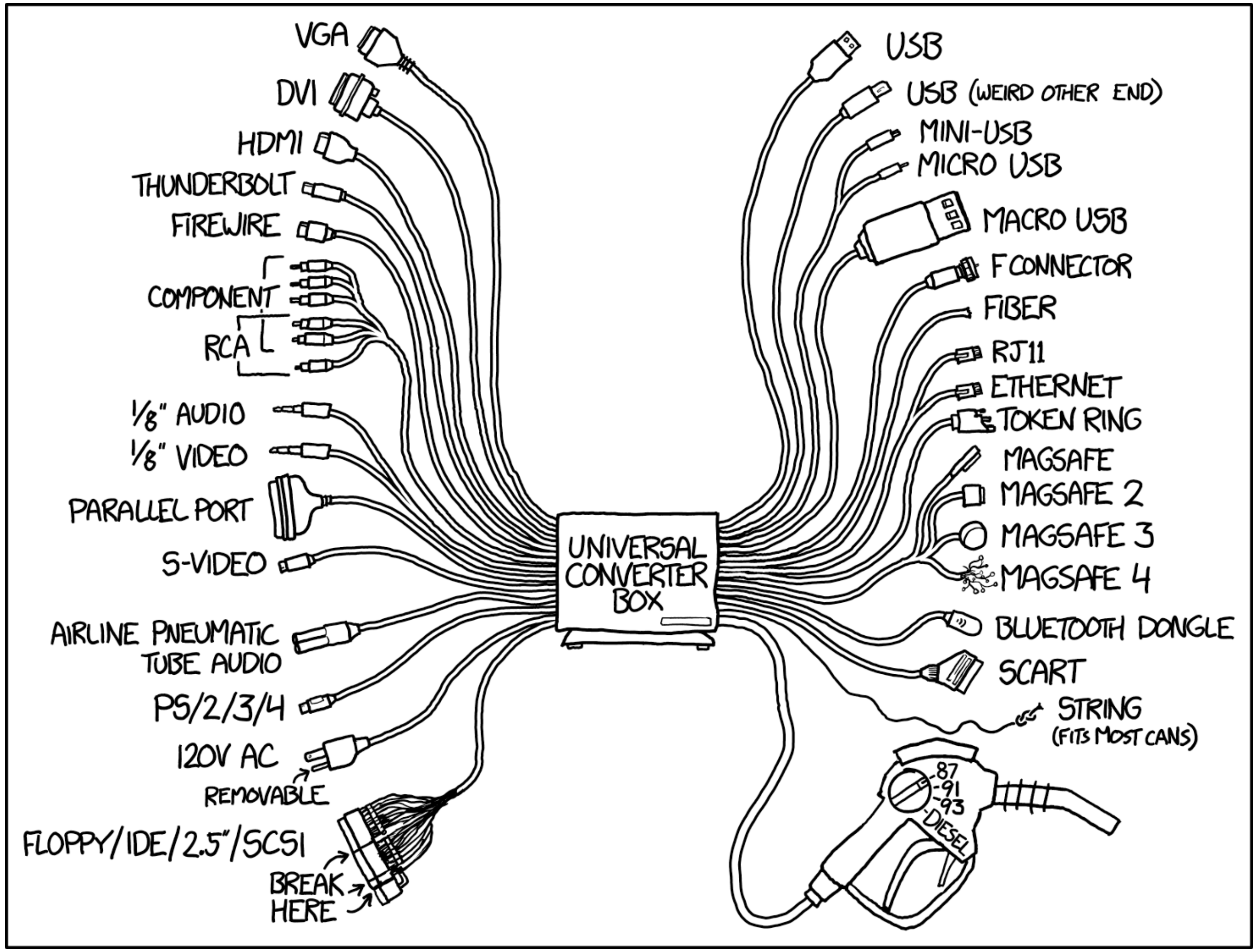
`<cmath>` fournit les fonctions

- Trigonométriques `sin, cos, tan, asin, acos,` ...
- Hyperboliques `cosh, sinh, tanh, asinh,` ...
- Exponentielles et logarithmiques `exp, log, …`
- Puissances `pow, sqrt, …`
- Valeur absolue `abs, fabs, …`
- Arrondi `round, ceil, floor, …`
- …

Voir http://www.cplusplus.com/reference/cmath/

Conversion entre types

- Syntaxe
- Types de conversions numériques
  - Promotion numérique entière
  - Conversions entier → entier, ? → réel, et réel → entier
  - Arrondis
- Conversions implicites



https://xkcd.com/1406/

Syntaxe de conversion

- Il y a 4 syntaxes qui permettent de convertir une expression d'un type à un autre. Par exemple, de `int` en `double`

```
int e = 42;
double d1 = double(e);                  // forme fonctionnelle
double d2 = (double)e;                  // operator de cast
double d3 = static_cast<double>(e);     // static_cast
double d4 = e;                          // conversion implicite
```

- La syntaxe fonctionnelle requiert un nom de type en un seul mot. Si nécessaire, on peut utiliser un alias de type

```
using ull = unsigned long long;        unsigned long long u = ull(e);
```

- Certains guides de style recommandent de ne pas dupliquer le nom de type en utilisant le mot clé `auto` pour déclarer la variable initialisée par conversion

```
auto d5 = (double)e;                    // d5 de type double
```

Types de conversions numériques

Avec 5 types entiers signés, 5 types non signés et 3 types réels, il y a 156 conversions possibles d'un type numérique à un autre. Pour simplifier, on les groupe en

- Promotions numériques vers `int`
- Conversions
  - Entier vers entier
  - Tout type numérique vers réel
  - Réel vers entier

Promotions numériques entières

- Conversion des types `char`, `signed char`, `unsigned char`, `signed short`, ou `unsigned short` vers `int`
  - exceptionnellement vers `unsigned int` si `int` n'est pas capable de stocker toutes les valeurs du type d'origine. Par exemple la promotion de `unsigned short` dans le modèle de donnée LP32 où `int` n'utilise que 16 bits.
- `bool` → `int` est aussi une promotion numérique. `false` vaut 0, `true` vaut 1
- Garantissent de **préserver la valeur**
- Fréquemment réalisées implicitement dans l'évaluation d'expressions arithmétiques, les opérateurs `+, -, *, /, %` n'étant pas définis pour des entiers plus courts que `int`
- Prioritaires sur les autres conversions pour l'appel de fonctions surchargées (c.f. chapitre 10)

Conversion entier → entier

- Toutes les conversions entières qui ne sont pas des promotions numériques
- La valeur est **préservée si elle est représentable** dans le type de destination
- Sinon, elle choisit la valeur dans l'intervalle représentable du type de destination sur n bits qui est **congrue modulo $\mathbf{2^n}$** avec la valeur de départ.

Note : Avant C++20, la valeur était indéfinie dans ce cas pour les types de destinations signés

```
for(int s : { 100, 200, 8100, 40000, -10 })
   cout << setw(5) << s << " : " << setw(4) << +(unsigned char) s << "(uc) "
        << setw(4) << +(signed char) s << "(sc) " << setw(5) << (unsigned short) s << "(us) "
        << setw(6) << (signed short) s << "(ss)" << setw(11) << (unsigned int) s << "(ui)\n";
```

```
  100 :  100(uc)  100(sc)   100(us)    100(ss)        100(ui)
  200 :  200(uc)  -56(sc)   200(us)    200(ss)        200(ui)
 8100 :  164(uc)  -92(sc)  8100(us)   8100(ss)       8100(ui)
40000 :   64(uc)   64(sc) 40000(us) -25536(ss)      40000(ui)
  -10 :  246(uc)  -10(sc) 65526(us)    -10(ss) 4294967286(ui)
```

Conversions entières en complément à 2

- Avec la représentation en complément à 2, la conversion …
  - **signed** ↔ **unsigned** ne change aucun bit
  - **long** → **court** tronque les bits de gauche
  - **court** → **long** ajoute des zéros (`unsigned`) ou bits de signe (`signed`) à gauche

```
for(int s : { 200, 40'000, 42'000'000, -10 })
  cout << setw(8) << setfill(' ') << dec << s << " : " << hex << setfill('0')
    << setw(8)  << (signed int) s       << "(si) " << setw(8)  << (unsigned int) s       << "(ui) "
    << setw(4)  << (signed short) s     << "(ss) " << setw(4)  << (unsigned short) s     << "(us) "
    << setw(16) << (signed long long) s << "(sl) " << setw(16) << (unsigned long long) s << "(ul)\n";
```

```
     200 : 000000c8(si) 000000c8(ui) 00c8(ss) 00c8(us) 00000000000000c8(sl) 00000000000000c8(ul)
   40000 : 00009c40(si) 00009c40(ui) 9c40(ss) 9c40(us) 0000000000009c40(sl) 0000000000009c40(ul)
42000000 : 0280de80(si) 0280de80(ui) de80(ss) de80(us) 000000000280de80(sl) 000000000280de80(ul)
     -10 : fffffff6(si) fffffff6(ui) fff6(ss) fff6(us) fffffffffffffff6(sl) fffffffffffffff6(ul)
```

Conversions vers des réels

- La conversion de `float` en `double` est une **promotion numérique**, qui ne modifie pas la valeur. *Note : Cela n'aura pas d'incidence avant le chapitre 10*

- Pour les autres conversions vers `float`, `double` et `long double`
  - Si la valeur peut être représentée **exactement**, elle ne change pas
  - Si la valeur est comprise **entre deux valeurs représentables**, le choix entre ces deux valeurs dépend de l'implémentation.
  - Sinon, le résultat est indéfini

```
float a = 0.5;          cout << setprecision(10) << a << endl;
float b = 3.1415926536; cout << setprecision(10) << b << endl;
float c = 1e41;         // c est indéfini, probablement +inf
```

```
0.5
3.141592741
```

Conversion réel vers entier

- Pour convertir un réel en entier, **la partie fractionnaire est tronquée**

```
int    a = 2.55;               // a =  2
int    b = -2.55;              // b = -2
double c = 2.55;
int    d = c + 0.5;            // troncature => 3
```

- Pour une valeur **non représentable** dans le type entier, le résultat est **indéfini**

```
unsigned char a = 300;         // conversion entier → entier : a = 44

unsigned char b = 300.;        // conversion réel → entier : b indéterminé
                               // b = 0   (Apple LLVM 8.1)
                               // b = 255 (Windows gcc)

unsigned char c = int(300.);   // conversion double → int → unsigned char
                               // c = 44
```

Gestion des arrondis

- La librairie `<cmath>` fournit des fonctions d'arrondi qui fournissent une **valeur entière** mais dans le **type réel** d'origine
  - `trunc` en tronquant après la virgule
  - `round` l'entier le plus proche, en s'éloignant de zéro pour .5
  - `floor` l'entier plus petit ou égal
  - `ceil` l'entier plus grand ou égal
- Les mêmes fonctions finissant par `f` ou `l` convertissent en `float` ou `long double`
  - `truncf, floorf, roundf, ceilf`
  - `truncl, floorl, roundl, ceill`

```
value  trunc  round  floor   ceil
---------------------------------
  2.3    2.0    2.0    2.0    3.0
  3.8    3.0    4.0    3.0    4.0
  5.5    5.0    6.0    5.0    6.0
 -2.3   -2.0   -2.0   -3.0   -2.0
 -3.8   -3.0   -4.0   -4.0   -3.0
 -5.5   -5.0   -6.0   -6.0   -5.0
```

Attention aux erreurs d'arrondi

- La conversion réel → entier pour donner des résultats surprenants

```
double d = 100 * 4.35;
cout << d << " ?= " << int(d) << endl;
```

```
435 ?= 434
```

- Le calcul en réel se fait avec une précision finie en sa représentation binaire. Le résultat n'est pas exactement 435

```
cout << setprecision(20) << d << endl;
```

```
434.99999999999994316
```

- On résout ce problème en spécifiant explicitement la méthode d'arrondi avant de convertir en entier

```
cout << int(round(d)) << endl;
```

```
435
```

Conversions implicites

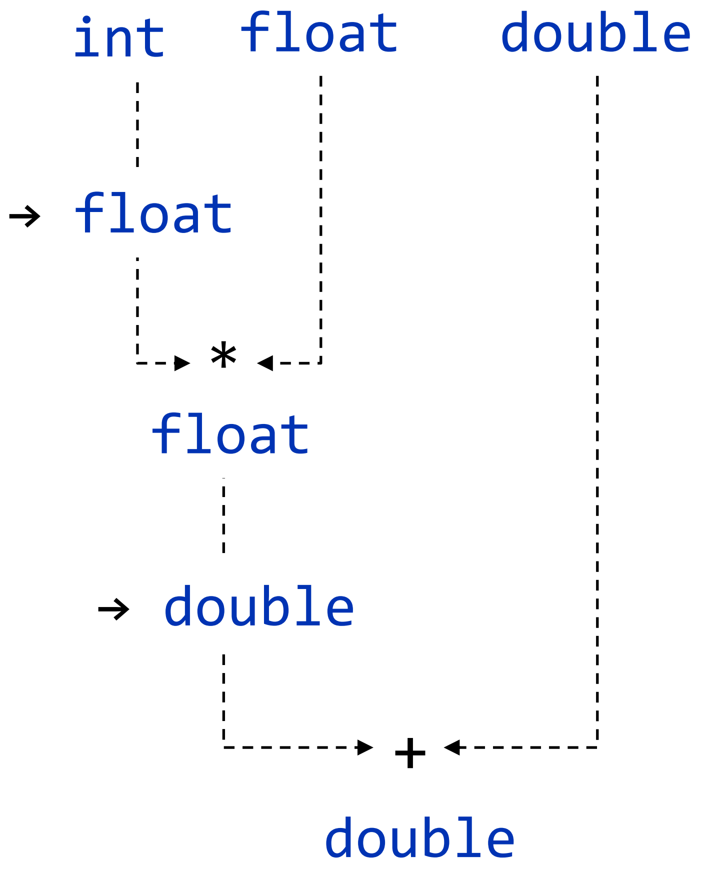
- Il est possible d'écrire des expressions arithmétiques en mélangeant les types. Par exemple,

```
auto x = 5 * 3.14F + 5.3e-2;
```

- Pourtant, les opérateurs arithmétiques binaires
  - ne sont définis que pour deux **opérandes de même type**
  - **retournent** une valeur du **même type** que leurs opérandes
- Les opérateurs (unaires ou binaires) ne sont **pas définis** pour les types entiers plus courts que `int` (`signed char`, `unsigned char`, `signed short` et `unsigned short`)
- Il y a donc **conversion implicite des opérandes** avant d'appliquer l'opérateur

Exemple

```
auto x = 5 * 3.14F + 5.3e-2;
```



Conversion arithmétique

- L'évaluation de l'expression se fait selon l'**ordre défini par les priorités** des opérateurs
  1. Unaire `(+,-)`
  2. Multiplicatif `(*,/,%)`
  3. Additif `(+,-)`

- **Pour les opérateurs unaires**, on applique si nécessaire la promotion numérique entière
  - Les expressions `+a` et `-a` sont de type `int` si `a` est de type `char` ou `short`, signé ou pas

```
unsigned char a = 65;
unsigned short b = 1;
unsigned int   c = 1;
cout << a << " " << +a << " " << -a << " " << -b << " " << -c << endl;
```

```
A 65 -65 -1 4294967295
```

Conversion arithmétique

Pour les opérateurs binaires

- On applique la **promotion numérique entière** aux types `char` ou `short`, signés ou pas
- Si un opérande est de type `long double`, on convertit l'autre en `long double`
- Sinon, si un opérande est de type `double`, on convertit l'autre en `double`
- Sinon, si un opérande est de type `float`, on convertit l'autre en `float`
- Sinon, la conversion est entre deux types entiers. Elle dépend
  - des signes (`signed` → `unsigned`)
  - des rangs (`int` → `long` → `long long`)

Conversion arithmétique entre entiers

- Quand le type d’un des opérandes est meilleur ou égal à l’autre du point de vue du **signe** et de la **longueur**, l’autre est converti dans ce type

```
int, long                          -> long
int, long long                     -> long long
int, unsigned int                  -> unsigned int
int, unsigned long                 -> unsigned long
int, unsigned long long            -> unsigned long long

long, long long                    -> long long
long, unsigned long                -> unsigned long
long, unsigned long long           -> unsigned long long

long long, unsigned long long      -> unsigned long long

unsigned int, unsigned long        -> unsigned long
unsigned int, unsigned long long   -> unsigned long long

unsigned long, unsigned long long -> unsigned long long
```

Conversion arithmétique entre entiers (2)

- Quand un type est meilleur que l'autre selon un critère et vice-versa pour l'autre, i.e. pour les 3 paires de types suivantes

```
unsigned int , long
unsigned int , long long
unsigned long, long long
```

- La conversion **dépend du modèle de donnée**, i.e. du nombre de bits utilisés pour représenter chaque type
  - Si l'opérande signé peut représenter toutes les valeurs de l'opérande non signé, l'opérande non signé est converti dans le type signé.
  - Sinon, les deux opérandes sont convertis dans la version non signée du type signé

Conversion arithmétique entre entiers (3)

- En pratique, si `int` et `long` utilisent 32 bits et `long long` 64 (LLP64 sur Windows API)

```
unsigned int , long      -> unsigned long
unsigned int , long long -> long long
unsigned long, long long -> long long
```

- Par contre, si `int` utilise 32 bits, `long` et `long long` 64 (LP64 sur Linux ou Mac OS X)

```
unsigned int , long      -> long
unsigned int , long long -> long long
unsigned long, long long -> unsigned long long
```

- Et donc, l'exécution du code suivant dépend de l'OS

```
unsigned int a = 1;  long b = 2;
unsigned long c = 1; long long d = 2;
cout << a - b << " " << c - d << endl;
```

```
-1 18446744073709551615
```

```
4294967295 -1
```

Attention aux comparaisons

- Les règles pour les opérateurs arithmétiques s'appliquent également aux opérateurs de comparaison
- Cela pose problème si l'on compare des entiers signés et non signés

```
signed int   a = -1;
unsigned int b =  1;

cout << boolalpha;

cout << (signed(-1)   < signed(1))   << endl;

cout << (unsigned(-1) < unsigned(1)) << endl;

cout << (a < b) << endl;
```

```
true

false

false
```

Avertissements du compilateur

- Le compilateur peut nous avertir de ces conversions implicites dégradantes (perte de précision / dépassement possible) avec l'option **`-Wconversion`**

```
int unEntier = 3.14;
```

```
warning: implicit conversion from
'double' to 'int' changes value from
3.14 to 3 [-Wliteral-conversion]
```

- Le compilateur peut nous en avertir de conversions implicites entre types signés et non signés avec l'option **`–Wsign-conversion`**

```
int a = 4;
unsigned int b = 5;
cout << a – b; // affiche 4294967295
```

```
warning: implicit conversion
changes signedness: 'int' to
'unsigned int' [-Wsign-conversion]
```

Au delà du C++, quel que soit le langage …

- Quels types sont disponibles pour traiter des entiers / des réels ?
- Quel est leur coût en mémoire ?
- Le langage dispose-t-il d'un type non signé ?
- Quelle est la plage des valeurs représentables par chaque type ?
- Quelle est la précision des calculs en réel ?
- Comment le dépassement est-il géré lors des calculs ?
- Quels opérateurs / fonctions mathématiques sont disponibles ?
- Comment convertir explicitement d'un type à l'autre ?
- Comment les conversions implicites sont-elles gérées dans les expressions hybrides (si elles sont autorisées) ?
- Comment les arrondis sont-ils gérés ?
- Comment formater ces nombres sous forme de texte ?
- Existe-t-il un type réel en base décimale pour les calculs financiers ?