# QCM - Types fondamentaux et encodage

Exercices - Chapitre 2 - Solutions [CSC-43043-EP] 2026

---

## Solutions

**Question 1.** Réponse : **C)**

*255*

Un `unsigned char` est encodé sur 8 bits et représente des valeurs non signées. Il peut donc stocker des valeurs de 0 à $2^8 - 1 = 255$. La valeur 256 nécessiterait 9 bits. La valeur 127 correspond au maximum d'un `signed char` (sur 7 bits de données), et 128 à sa valeur absolue minimale négative.

**Question 2.** Réponse : **B)**

*10*

Le nombre binaire `00001010` se décompose en puissances de 2 : $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 2 = 10$. Seuls les bits aux positions 1 et 3 sont à 1, ce qui donne 10 en décimal.

**Question 3.** Réponse : **B)**

*`00001101`*

Le nombre 13 en binaire s'obtient par décomposition en puissances de 2 : $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$. En plaçant des 1 aux positions 3, 2 et 0, on obtient `00001101`. On peut vérifier : $1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 13$.

**Question 4.** Réponse : **B)**

*26*

En hexadécimal, chaque chiffre représente 4 bits (un nibble). `0x1A` se décompose en `1 × 16 + A × 1`. La lettre `A` vaut 10 en décimal, donc `0x1A = 16 + 10 = 26`. Les littéraux hexadécimaux sont couramment utilisés en C++ pour les masques de bits et les adresses mémoire.

**Question 5.** Réponse : **C)**

*8 octets*

Le type `double` (double précision IEEE 754) occupe 8 octets (64 bits). Il se compose d'un bit de signe, 11 bits d'exposant et 52 bits de mantisse. Cela lui confère une précision d'environ 15-16 chiffres décimaux significatifs, contre 6-7 pour un `float` (4 octets).

**Question 6.** Réponse : **B)**

*`11111111`*

En complément à deux sur 8 bits, $-1$ est obtenu en partant de $+1$ (`00000001`), en inversant les bits (`11111110`), puis en ajoutant 1 (`11111111`). On peut vérifier : $11111111 + 00000001 = 100000000$, le bit de débordement est ignoré et on obtient bien 0. Ainsi $-1$ est représenté par tous les bits à 1.

**Question 7.** Réponse : **B)**

*`0b1000` (8)*

L'opérateur `&` (ET bit à bit) donne 1 uniquement lorsque les deux bits correspondants sont à 1. Pour `1100 & 1010` : position 3 → $1\&1 = 1$, position 2 → $1\&0 = 0$, position 1 → $0\&1 = 0$, position 0 → $0\&0 = 0$. Le résultat est `1000`, soit 8 en décimal.

**Question 8.** Réponse : **B)**

*`78 56 34 12`*

En Little Endian, l'octet de poids faible (least significant byte) est stocké en premier, à l'adresse mémoire la plus basse. Pour `0x12345678`, l'octet de poids faible est `0x78`, suivi de `0x56`, `0x34`, et enfin `0x12`. L'ordre en mémoire est donc `78 56 34 12`. C'est la convention utilisée par les processeurs Intel x86 et ARM par défaut.

**Question 9.** Réponse : **B)**

$-2^{31}$ *à* $2^{31} - 1$

En complément à deux sur 32 bits, le bit de poids fort est le bit de signe. Les valeurs représentables vont de $-2^{31}$ ($-2\,147\,483\,648$) à $2^{31} - 1$ ($2\,147\,483\,647$). L'asymétrie vient du fait que le zéro est compté du côté positif. La réponse D serait incorrecte car $-2^{31}$ est bien représentable.

**Question 10.** Réponse : **C)**

*Parce que `0.1` et `0.2` n'ont pas de représentation binaire exacte en IEEE 754*

Les nombres `0.1` et `0.2` ne possèdent pas de représentation binaire finie en base 2, tout comme 1/3 n'a pas de représentation décimale finie. IEEE 754 en stocke une approximation. Lorsqu'on additionne ces approximations, le résultat diffère très légèrement de l'approximation de `0.3`. C'est pourquoi on compare les flottants avec une tolérance (epsilon) plutôt qu'avec `==`.

**Question 11.** Réponse : **B)**

*Il est négatif et son exposant réel est 2*

Le premier bit est 1, ce qui signifie que le nombre est négatif. L'exposant encodé est `10000001` en binaire, soit $128 + 1 = 129$. L'exposant réel est obtenu en soustrayant le biais : $129 - 127 = 2$. Le nombre est donc négatif avec un exposant réel de 2, ce qui correspond à une multiplication par $2^2 = 4$.

**Question 12.** Réponse : **B)**

*`0xCD`*

Le décalage `w >> 16` déplace les bits 16-31 en position 0-15. Pour `0xABCD1234`, les bits 16-23 contiennent `0xCD` et les bits 24-31 contiennent `0xAB`. Après le décalage de 16, la valeur basse est `0xABCD`. Le masque `& 0xFF` ne garde que les 8 bits de poids faible, soit `0xCD`. Le mot original se décompose octet par octet : `AB | CD | 12 | 34`.

**Question 13.** Réponse : **B)**

*On obtient `10000000`, interprété comme -128 (dépassement de capacité)*

En complément à deux sur 8 bits, `01111111` est la plus grande valeur positive (+127). Ajouter 1 donne `10000000`, qui en complément à deux est interprété comme $-128$. C'est un dépassement de capacité (overflow) : on passe brutalement de la valeur maximale positive à la valeur minimale négative. Ce comportement est en réalité indéfini (undefined behavior) en C++ pour les entiers signés.