Binaire, hexadécimal et décimal

Tags Misc

Binaire, hexadécimal et décimal

Qu'est-ce qu'un bit ?

En **binaire**, un **bit** c'est soit le chiffre **0** soit le chiffre **1**.

La **valeur maximum** d'un bit **c'est 1.**

C'est le même principe qu'un **booléen**. Vous le savez, un booléen c'est un type de valeur qui ne peut être que deux choses : soit **false**, soit **true**.

Vous voyez donc que le bit et le booléen peuvent chacun n'avoir que deux valeurs possibles.

En réalité, le bit qui est lu par votre ordinateur est peut être un 0, peut-être un 1, mais ce chiffre, dans tous les cas, n'est **qu'une idée**. C'est un chiffre et **c'est abstrait**.

Il n'y a pas de 0 ni de 1 qui circule dans votre ordinateur. On verra ce qui circule dans votre ordinateur plus tard, mais l'essentiel, c'est que pour l'instant, vous compreniez que **le bit n'est qu'un chiffre** qui est soit zéro, soit un, et qu'il n'existe pas vraiment.

Le bit, c'est surtout **un chiffre**, et les humains utilisent les chiffres principalement pour une chose : **faire des calculs mathématiques**. Ce sont des calculs abstraits.

Par contre, ce qui est **réel et très concret**, dans votre ordinateur, c'est **les transistors**.

Un **transistor**, pour faire *très simple*, c'est juste un petit bout de métal qui va laisser passer ou non l'électricité.

Et le code le plus basique, dans un ordinateur, pour dire à un transistor de faire passer de l'électricité ou non, c'est du code rédigé dans un **langage de programmation** qu'on appelle le **langage machine**. **C'est littéralement du code écrit avec uniquement des 0 et des 1**. Totalement low level.

Il n'y a pas plus bas niveau comme langage de programmation que le langage machine. Il n'y a aucune abstraction dedans.

Pourquoi est-ce que je parle d'abstraction ? Et qu'est-ce que c'est ? Un langage de haut niveau, avec un **haut niveau d'abstraction**, c'est par exemple le langage avec lequel vous avez commencé à apprendre l'**algorithmique** : le **JavaScript**.

Il y a **un haut niveau d'abstraction** dans ce langage, ce qui permet de donner à l'ordinateur physique et concret des instructions écrites non pas avec des 0 et des 1 mais plutôt avec des mots anglais, comme l'échantillon de code ci-dessous, écrit avec un type de variable ("**const**" pour une valeur constante), avec des noms de variables ("**name"**, "**age"**, "**greetingMessage")** et avec une fonction ("**console.log( )"**). Tout ça, c'est du code écrit avec des mots, qui eux-mêmes sont écrits avec les lettres de notre alphabet.

L'abstraction est très élevée. On écrit les idées, et la machine exécute les actions précises.

```
const name = "Philippin";
const age = 32;
const greetingMessage = `Hello ! My name is ${name} and I am ${age} years old.`;
```

```
console.log(greetingMessage);
// logs into the console : "Hello ! My name is Philippin and I am 32 years old."
```

> Petite anecdote : le nom de notre groupe, ***Hopper 2024***, est inspiré de l'informaticienne **Grace Hopper**. Hopper a notamment participé à la création du **Fortran** et du **Cobol**, les deux premiers langages de programmation avec un niveau d'abstraction élevé. Ce niveau d'abstraction élevé est rendu possible grâce à ce qu'on appelle un **compiler**. Le compiler va transformer le texte abstrait (le code) rédigé avec des lettres (**de a à z**) en langage machine (**fait de 0 et de 1**). Le langage machine pourra être lu directement par le matériel physique de l'ordinateur, et plus précisément par le **processeur** (**CPU**) qui est chargé d'exécuter les instructions et donc de dire à quel moment tel transistor va passer de "chargé électriquement" à "non-chargé électriquement", ou inversement.

Un bit, c'est compter en base 2.

Donc vous l'avez compris, le bit c'est un chiffre qui vaut soit 0, soit 1, et on utilise ce chiffre pour compter et faire des calculs mathématiques.

En binaire, les ordinateurs comptent en base 2. Qu'est-ce que cela signifie ? Cela signifie qu'il y a 2 possibilités maximum dans un seul chiffre binaire.

Vous savez qu'un nombre, dans votre système décimal (quand vous comptez le nombre de pommes qui restent après que Mathilde en ait mangé une), c'est une combinaison de plusieurs chiffres.

> Si Mathilde a 15 pommes, et qu'elle en mange 1, il ne lui reste plus que 14 pommes.
> Parce que 15 - 1 = 14

Un nombre, c'est une combinaison de chiffres.

Ici vous voyez le nombre **15**, c'est une combinaison en système décimal de **deux chiffres** : le 1 et le 5.

Donc maintenant le but, c'est que vous imaginiez un nombre binaire composé de plusieurs 0 et de 1.

Par exemple : **1101 1110**

Vous voyez que j'ai choisi ici de prendre un nombre binaire formé de **8 bits** :

> **1101 1110**

Je n'en ai pas pris **8** au hasard. Une combinaison de **8 bits**, cela s'appelle cela un **byte** (ou "**octet**" en français).

Et qu'est-ce que l'ordinateur va vouloir faire avec des nombres comme ce byte ? Il va vouloir faire des calculs avec.

C'est comme ça que l'ordinateur peut décider s'il entreprend telle action ou non. En faisant des calculs simples en base 2.

Mais avant de pouvoir ajouter deux nombres, ou les soustraire, il faut pouvoir compter dans un premier temps.

Donc maintenant il faut que vous compreniez comment vous pouvez compter un byte. Compter un byte, c'est un peu comme lire un byte.

Pour cela, vous devrez respecter la même idée en binaire (qui est en base 2), en décimal (qui est en base 10) et en hexadécimal (qui est en base 16).

Quelle est cette idée ?

L'idée présente dans tous les systèmes de numération

Vous allez prendre le nombre, et vous allez déjà supposer plusieurs choses par rapport à ce nombre.

1. Premièrement, vous allez partir du principe que le nombre est comme une Array, c'est une collection de chiffres. Donc votre nombre est une Array.

2. Deuxièmement, vous allez supposer que le nombre d'éléments de cette Array est représenté par la lettre N. Donc N est votre nombre d'éléments dans l'Array.
3. Troisièmement, votre Array a bien sûr un indice pour chaque élément qu'elle comprend. Vous allez partir de l'indice zéro dans cette Array. C'est comme en JavaScript. Et vous finirez à l'indice N - 1.

Vous avez vu qu'en JavaScript, pour obtenir le troisième élément de cette array, vous deviez aller chercher l'indice 2 :

```
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const thirdElement = arr[2];
console.log(thirdElement);
// logs into the console : "2".
```

Donc votre nombre binaire est une Array, de taille N éléments, et le premier élément se situe à l'indice 0.

Cela va vous aider à lire le byte, car vous allez devoir toujours lire de droite à gauche.

Donc quel était encore votre nombre binaire ? C'était celui ci :

Indice	7	6	5	4	3	2	1
Valeur binaire	1	1	0	1	1	1	1

Et vous allez lire de droite à gauche.

Vous voyez que la valeur binaire de votre nombre est dans la rangée du dessous du tableau.
Vous voyez aussi l'indice de votre nombre dans la rangée du dessus du tableau.

Quel est votre premier chiffre, votre première valeur binaire en partant de la droite ? C'est un 0.

Donc ce que vous allez faire, c'est multiplier votre valeur binaire (0) par la base binaire (la base 2) à la puissance (une puissance, un exposant) correspondant à l'indice de votre valeur binaire. L'indice ici c'est 0.

Donc vous faites ceci :

$$0 \times 2^0 = 0$$

(En sachant que n'importe quel nombre exposant zéro donne toujours 1 comme résultat).

Bravo ! Vous avez lu votre premier chiffre binaire, celui qui est tout à droite. Et il vaut 0.

Maintenant, vous pouvez continuer en allant toujours plus vers la gauche.

Le deuxième chiffre binaire est un 1, et son indice est 1.

Vous allez donc le lire ainsi :

$$1 \times 2^1 = 2$$

(En sachant que n'importe quel nombre exposant un donne toujours ce nombre lui-même comme résultat).

Votre deuxième chiffre binaire, à l'indice 1, vaut donc 2.

Ensuite le troisième chiffre binaire :

$$1 \times 2^2 = 4$$

Votre troisième chiffre vaut 4.

Et ainsi de suite.

Vous avez compris, alors je vous affiche la liste de la lecture de votre byte, de droite à gauche, de haut en bas :

$$0 \times 2^0 = 0$$

> $1 \times 2^1 = 2$
>
> $1 \times 2^2 = 4$
>
> $1 \times 2^3 = 8$
>
> $1 \times 2^4 = 16$
>
> $0 \times 2^5 = 0$
>
> $1 \times 2^6 = 64$
>
> $1 \times 2^7 = 128$

Ainsi, votre byte 1101 1110 peut être lu ainsi :

Indice	7	6	5	4	3	2	1
Valeur binaire	1	1	0	1	1	1	1
Valeur décimale	128	64	0	16	8	4	2

C'est ainsi que vous pouvez lire un nombre binaire de droite à gauche.

Et maintenant, vous pouvez additionner chaque valeur décimale de votre byte, et vous obtenez le résultat décimal de **222.**

(Sachant que 128 + 64 + 0 + 16 + 8 + 4 + 2 + 0 = 222).

Un nombre décimal, c'est compter en base 10

Donc vous avez vu comment compter en base 2, et comment traduire dans le système décimal qui est un système en base 10.

Si vous avez tout suivi jusqu'ici, vous comprendrez que 222 en décimal, c'est un peu la même idée.

Vous allez compter de droite à gauche.

Vous allez partir d'une Array de taille N, et cette Array commence à l'indice zéro.

Sauf qu'au lieu de multiplier seulement des bits qui valent 0 ou qui valent 1 par la base binaire 2 à la puissance d'indice **x** (donc 1×2^x), vous allez multiplier nos chiffres décimaux (de 0 à 9) par la base décimale 10 à la puissance d'indice **x** (donc 2×10^x).

Vous pouvez déjà voir à quoi cela va ressembler avec votre nombre décimal 222.

222 c'est quoi ? 222 c'est :

Indice	2	1	0
Valeur décimale	2	2	2

> $2 \times 10^0 = 2$
>
> $2 \times 10^1 = 20$
>
> $2 \times 10^2 = 200$

Ainsi, si vous décomposez tout, vous voyez que votre 222 est en réalité 200 + 20 + 2.

Une valeur hexadécimale, c'est compter en base 16

Les hexadécimales c'est toujours la même idée.

Vous savez que mot hexa signifie 6, et décimal signifie 10.

Donc base hexadécimale, base 16.

Les chiffres n'iront pas de 0 à 1 comme en binaire, ni de 0 à 9 comme en décimal.

En hexadécimal, les valeurs iront de 0 à 9, mais à partir de l'indice 10, vous allez représenter chaque valeur par une lettre (car on n'a pas d'autres chiffres qui vont au-delà de 9).

La lettre A représentera donc la valeur hexadécimale 10 (à l'indice 10), la lettre B représentera la valeur hexadécimale 11 (à l'indice 11), etc.

Voici le tableau représentant toutes les valeurs hexadécimales :

Valeur hexadécimale	0	1	2	3	4	5	6
Valeur décimale	0	1	2	3	4	5	6

Vous savez qu'on représente notamment les couleurs avec des valeurs hexadécimales.

En décimal, les couleurs peuvent aller de RGB(0,0,0) à RGB(255, 255, 255).

En hexadécimal, les couleurs peuvent aller de #000000 à #FFFFFF.

Pour représenter le nombre décimal 222 en hexadécimal, le plus simple, c'est de convertir 222 en binaire et ensuite le binaire en hexadécimal très rapidement.

L'utilité de l'hexadécimal, c'est de pouvoir représenter le binaire de manière plus concise.

Donc mettons qu'on reparte de notre byte, le 1101 1110.

Comment représenter cela en hexadécimal ?

C'est simple : vous pouvez regrouper chaque fois un demi byte, donc à savoir 4 bits, par deux valeurs hexadécimales.

Donc si vous avez bien suivi jusqu'ici, vous avez compris qu'une valeur hexadécimale représente 2 bits. Vous pourrez représenter chaque paire de bits par une seule valeur hexadécimale.

Regardez comment vous pouvez représenter votre fameux byte en hexadécimal :

Indice	7	6	5	4	3	2	1
Valeur binaire	1	1	0	1	1	1	1

Indice	1	0
Valeur hexadécimale	D	E
Valeur décimale	$13 \times 16^1 = 208$	$14 \times 16^0 = 14$

Votre conversion part donc d'un point de départ, une valeur binaire (**1101 1110**), pour passer par une étape à la gare de l'hexadécimal (**DE)** afin de finalement s'arrêter au terminus décimal ($208 + 14 = 222$).

Valeurs maximum d'un type de données

Donc 1 byte comprend 8 bits.

Si vous prenez deux bytes, c'est comme Mathilde qui prendrait deux pommes, une dans chaque main... 2 bytes comprennent logiquement 16 bits, il y a $2 \times 8 = 16$ bits.

Avec la même logique élémentaire, 4 bytes comprennent 32 bits.

Et là vous retrouvez quelque chose que vous connaissez déjà, pour ceux qui ont déjà fait du C# par exemple. 4 bytes, c'est 32 bits, et c'est le nombre de bits utilisés pour pouvoir représenter un int32 en C#, ou plus communément appelé "***int***".

Les mots clés “**int**” et “**Int32**” représentent exactement le même type de données. C’est un nombre entier, signé (donc qui peut représenter aussi des nombres en dessous du zéro) et qui peut être compris **entre -2_147_483_648 < 0 < 2_147_483_647.**

```
int age = 32;
Int32 age = 32;
// Les deux lignes du dessus sont exactement les mêmes, après compilation.
```

Ça, c’est un Int32 signé.

Il existe aussi des Int32 non signés : les **uint**. Comme ils ne peuvent pas aller en dessous de 0, ils peuvent être compris **entre 0 et 4_294_967_295.**

Car ils ont de toute façon le même nombre de valeurs possibles, signé ou non signé. L’idée, c’est que le int non signé ne peut pas aller en dessous de zéro.

```
uint unsignedInteger = uint.MaxValue;
UInt32 unsignedInteger = 4_294_967_295;
// Les deux lignes du dessus sont exactement les mêmes, après compilation.
Console.WriteLine(unsignedInteger);
// logs into the console : "4294967295".
```

Philémon Émile Olivier Philippin - 24 septembre 2024.