Programmation Avancée

Cours 2 : La représentation binaire

Simon Forest

28 janvier 2021

Sommaire

Sommaire

Bit

Le « bit » est l'unité élementaire de stockage de l'information dans un ordinateur.

C'est une case mémoire qui peut contenir soit 0 (`false`) soit 1 (`true`), c'est-à-dire deux valeurs possibles.

Groupes

Pour stocker plus d'information, on peut considérer des « groupes » de bits.

Exemples :

- avec 2 bits, je peux manipuler $2 \times 2 = 2^2 = 4$ valeurs différentes
- avec 6 bits, je peux manipuler $2 \times \cdots \times 2 = 2^6 = 32$ valeurs différentes

Plus généralement, avec N bits, on peut stocker 2^N valeurs différentes.

Octet

Le bit, c'est une unité trop petite pour être manipulée directement efficacement.

À la place, on manie les **octets**, qui sont des groupes de 8 bits :

$$1 \text{ octet} = 8 \text{ bits} = 256 \text{ valeurs possibles}$$

Octet

Le bit, c'est une unité trop petite pour être manipulée directement efficacement.

À la place, on manie les **octets**, qui sont des groupes de 8 bits :

$$1 \text{ octet} = 8 \text{ bits} = 256 \text{ valeurs possibles}$$

Les types que l'on manipule en `C` sont des combinaisons d'octets.

`char`, `unsigned char`	1 octet
`short`, `unsigned short`	2 octets
`int`, `unsigned int`	4 octets
`long`, `unsigned long`	4 ou 8 octets suivant les machines

Octet

Le bit, c'est une unité trop petite pour être manipulée directement efficacement.

À la place, on manie les **octets**, qui sont des groupes de 8 bits :

$$1 \text{ octet} = 8 \text{ bits} = 256 \text{ valeurs possibles}$$

Les types que l'on manipule en `C` sont des combinaisons d'octets.

`char`, `unsigned char`	1 octet
`short`, `unsigned short`	2 octets
`int`, `unsigned int`	4 octets
`long`, `unsigned long`	4 ou 8 octets suivant les machines

La taille en octets des types est renvoyée par l'opération `sizeof` :

```
printf("%lu",sizeof(int)); // affiche '4'
```

Descriptions des bits

Décrire des groupes de bits par les valeurs des bits qu'ils contiennent n'est pas pratique :

> *Soit le groupe de* 4 *octets tel que le 3ème bit du 2ème octet vaut* 1, *le 2ème bit du premier octet vaut* 1 *et etc..*

On préfère à la place utiliser les nombres pour décrire une configuration d'octets.

Des bits aux nombres

Considérons à nouveau les bits.

Si 1 bit, alors la correspondance est simple :

$$\begin{array}{ccc} 0 & \mapsto & 0 \\ 1 & \mapsto & 1 \end{array}$$

Des bits aux nombres

Considérons à nouveau les bits.

Si 2 bits, alors on associe un nombre en triant les deux bits :

$$\begin{array}{rcl} 00 & \mapsto & 0 \\ 01 & \mapsto & 1 \\ 10 & \mapsto & 2 \\ 11 & \mapsto & 3 \end{array}$$

On remarque que

$$\begin{array}{rcr} 0x & \mapsto & x \\ 1x & \mapsto & 2+x \end{array}$$

Des bits aux nombres

Considérons à nouveau les bits.

Si 3 bits, alors on associe un nombre en triant les deux bits :

$$
\begin{array}{ccc}
000 & \mapsto & 0 \\
001 & \mapsto & 1 \\
010 & \mapsto & 2 \\
011 & \mapsto & 3 \\
100 & \mapsto & 4 \\
101 & \mapsto & 5 \\
110 & \mapsto & 6 \\
111 & \mapsto & 7
\end{array}
$$

On remarque que

$$
\begin{array}{ccc}
0yx & \mapsto & 2y + x \\
1yx & \mapsto & 4 + 2y + x
\end{array}
$$

Des bits aux nombres

Considérons à nouveau les bits.

Si n bits $x_{n-1}, x_{n-2}, \ldots, x_1, x_0$, alors

$$\boxed{x_{n-1}x_{n-2}\cdots x_1x_0 \qquad \mapsto \qquad x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0}$$

Exemple :

$$\begin{aligned}
11001 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 16 + 8 + 0 + 0 + 1 \\
&= 25
\end{aligned}$$

Des bits aux nombres

Considérons à nouveau les bits.

Si n bits $x_{n-1}, x_{n-2}, \ldots, x_1, x_0$, alors

$$\boxed{x_{n-1}x_{n-2}\cdots x_1x_0 \qquad \mapsto \qquad x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0}$$

Exemple :

$$\begin{array}{rcccccccccc}
11001 & = & 1 \times 2^4 & + & 1 \times 2^3 & + & 0 \times 2^2 & + & 0 \times 2^1 & + & 1 \times 2^0 \\
 & = & 16 & + & 8 & + & 0 & + & 0 & + & 1 \\
 & = & 25 & & & & & & & &
\end{array}$$

Si on a k octets, on a $n = 8k$ bits et la correspondance ci-dessus s'applique comme pour les séquences de bits.

Des nombres aux bits

Si maintenant on part d'un nombre $N \in \mathbb{N}$, comment avoir sa représentation en bits ?

Il faut pour cela calculer sa **décomposition binaire** de N.

Quelques exemples

On remarque déjà que

- nombre pair $\implies$ termine par 0,
- nombre impair $\implies$ termine par 1.

$$
\begin{array}{rcr}
2 & \mapsto & 10 \\
4 & \mapsto & 100 \\
8 & \mapsto & 1000 \\
12 & \mapsto & 1100
\end{array}
\qquad\qquad
\begin{array}{rcr}
3 & \mapsto & 11 \\
5 & \mapsto & 101 \\
11 & \mapsto & 1011 \\
15 & \mapsto & 1111
\end{array}
$$

Quelques exemples

On remarque aussi que la décomposition de N commence par la décomposition de $N/2$ (division entière).

$$
\begin{array}{rcr}
7 & \mapsto & 111 \\
12 & \mapsto & 1100 \\
16 & \mapsto & 10000 \\
31 & \mapsto & 11111
\end{array}
\qquad
\begin{array}{rl}
7/2 = 3 & (+\frac{1}{2}) \\
12/2 = 6 & \\
16/2 = 8 & \\
31/2 = 15 & (+\frac{1}{2})
\end{array}
\qquad
\begin{array}{rcr}
3 & \mapsto & 11 \\
6 & \mapsto & 110 \\
8 & \mapsto & 1000 \\
15 & \mapsto & 1111
\end{array}
$$

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version récursive :

```c
void afficher_decomposition(unsigned int N)
{
  if(N == 0)
    return;
  printf("%d",N % 2); // affiche "0" si N pair, "1" si N impair
  afficher_decomposition(N/2);
}
```

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version récursive :

```
void afficher_decomposition(unsigned int N)
{
  if(N == 0)
    return;
  printf("%d",N % 2); // affiche "0" si N pair, "1" si N impair
  afficher_decomposition(N/2);
}
```

Pour $N = 8$, affiche "0001" : mauvais ordre !

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version récursive (bon ordre) :

```
void afficher_decomposition(unsigned int N)
{
  if(N == 0)
    return;
  // printf("%d",N % 2);
  afficher_decomposition(N/2);
  printf("%d",N % 2); // affiche "0" si N pair, "1" si N impair
}
```

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version récursive (bon ordre) :

```
void afficher_decomposition(unsigned int N)
{
  if(N == 0)
    return;
  // printf("%d",N % 2);
  afficher_decomposition(N/2);
  printf("%d",N % 2); // affiche "0" si N pair, "1" si N impair
}
```

Pour $N = 8$, affiche "1000" : bon ordre.

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version itérative (mauvais ordre) :

```
void afficher_decomposition(unsigned int N)
{
  while(N > 0)
  {
    printf("%d",N % 2); // affiche "0" si N pair, "1" si N impair
    N /= 2; // équivalent à: N = N/2;
  }
}
```

Pour $N = 8$, affiche "0001" : mauvais ordre !

Algorithme

On déduit de ces propriétés un algorithme pour calculer la décomposition binaire de N.

Version itérative (bon ordre) :

```c
void afficher_decomposition(unsigned int N)
{
  int tab[32]; // 32 = sizeof(unsigned int) * 8
  int curseur = 0;
  while(N > 0){
    tab[curseur++] = N % 2;
    N /= 2; // équivalent à: N = N/2;
  }
  while(curseur > 0){
    printf("%d",tab[--curseur]);
  }
}
```

Incrémentation

Au fait, on est d'accord que, étant donnée une variable `int i`,

`i++` n'est pas la même chose que `++i` ?

Incrémentation

Au fait, on est d'accord que, étant donnée une variable `int i`,

`i++` n'est pas la même chose que `++i` ?

`i++` : on incrémente `i` et on renvoie l'**ancienne** valeur de `i`.

`++i` : on incrémente `i` et on renvoie la **nouvelle** valeur de `i`.

Incrémentation

Au fait, on est d'accord que, étant donnée une variable `int i`,

`i++` n'est pas la même chose que `++i` ?

Pareil pour `--` :

`i--` : on décrémente `i` et on renvoie l'**ancienne** valeur de `i`.

`--i` : on décrémente `i` et on renvoie la **nouvelle** valeur de `i`.

Incrémentation

Ainsi,

```
tab[curseur++] = N % 2;
```

est équivalent à

```
tab[curseur] = N % 2;
curseur=curseur+1;
```

mais pas à

```
curseur=curseur+1;
tab[curseur] = N % 2;
```

Incrémentation

Ainsi,

```
printf("%d",tab[--curseur]);
```

est équivalent à

```
curseur=curseur-1;
printf("%d",tab[curseur]);
```

mais pas à

```
printf("%d",tab[curseur]);
curseur=curseur-1;
```

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Déjà, comment on fait en base 10 ?

$$
\begin{array}{rr}
 & 932 \\
+ & 1285 \\
\hline
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Déjà, comment on fait en base 10 ?

$$\begin{array}{rr} & 932 \\ + & 1285 \\ \hline & 7 \end{array}$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Déjà, comment on fait en base 10 ?

$$
\begin{array}{rr}
 & 1 \\
 & 932 \\
+ & 1285 \\
\hline
 & 17
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Déjà, comment on fait en base 10 ?

$$
\begin{array}{rr}
 & 11 \\
 & 932 \\
+ & 1285 \\
\hline
 & 217
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Déjà, comment on fait en base 10 ?

$$\begin{array}{rr} & 11 \\ & 932 \\ + & 1285 \\ \hline & 2217 \end{array}$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$\begin{array}{rr} & 110 \\ + & 1011 \\ \hline \end{array}$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$\begin{array}{rr} & 110 \\ + & 1011 \\ \hline & 1 \end{array}$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$
\begin{array}{rr}
 & 1 \\
 & 110 \\
+ & 1011 \\
\hline
 & 01
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$
\begin{array}{rr}
 & 11 \\
 & 110 \\
+ & 1011 \\
\hline
 & 001
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$\begin{array}{rr} & 111 \\ & 110 \\ + & 1011 \\ \hline & 0001 \end{array}$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

En base 2, c'est pareil.

$$
\begin{array}{rr}
 & 111 \\
 & 110 \\
+ & 1011 \\
\hline
 & 10001
\end{array}
$$

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Attention, comme on a un nombre fixé de bits, on peut faire un **dépassement** lors d'une opération.

```
unsigned int x = 2147483648, y = 2147483648;
printf("%u",x+y); // affiche '0'
```

Addition

Comment on additionne deux nombres avec la représentation binaire ?

Attention, comme on a un nombre fixé de bits, on peut faire un **dépassement** lors d'une opération.

```
unsigned int x = 2147483648, y = 2147483648;
printf("%u",x+y); // affiche '0'
```

Avec k bits, quand on dépasse 2^k avec l'addition, le résultat est ramené entre 0 et $2^k - 1$ en retranchant 2^k.

Dans l'exemple au-dessus, on a `x+y` $= 4294967296 = 2^{32}$. Donc ce résultat est ramené à 0 en retranchant 2^{32} ($32 = 8 \times$ `sizeof(unsigned int)`).

Multiplication

L'algorithme classique de la multiplication fonctionne aussi, même si ce n'est pas le plus efficace (*c.f.* cours d'algorithmique).

$$
\begin{array}{rr}
 & 110 \\
\times & 1011 \\
\hline
 & 110 \\
+ & 110 \\
+ & 000 \\
+ & 110 \\
\hline
 & 1000010
\end{array}
$$

Les dépassements sont aussi gérés en ramenant le résultat entre 0 et $2^k - 1$ (k le nombre de bits du type) en retranchant des multiples de 2^k.

Bits et octets

Nombres négatifs

Notation hexadécimale

Boutisme

Opérations sur les bits

Nombres négatifs

Comment représenter les nombres négatifs avec la représentation binaire ?

Nombres négatifs

Comment représenter les nombres négatifs avec la représentation binaire ?

Première idée : réserver un bit (par exemple le premier), pour dire si le nombre est négatif ou pas.

$$
\begin{array}{lcrlcr}
\color{red}{0}0000000 & \mapsto & 0 & \color{red}{1}0000000 & \mapsto & 0 \\
\color{red}{0}0000101 & \mapsto & 5 & \color{red}{1}0000101 & \mapsto & -5 \\
\color{red}{0}0011110 & \mapsto & 30 & \color{red}{1}0011110 & \mapsto & -30 \\
\color{red}{0}1111111 & \mapsto & 127 & \color{red}{1}1111111 & \mapsto & -127
\end{array}
$$

Nombres négatifs

Comment représenter les nombres négatifs avec la représentation binaire ?

Première idée : réserver un bit (par exemple le premier), pour dire si le nombre est négatif ou pas.

$$
\begin{array}{lcr@{\qquad\qquad}lcr}
\color{red}{0}0000000 & \mapsto & 0 & \color{red}{1}0000000 & \mapsto & 0 \\
\color{red}{0}0000101 & \mapsto & 5 & \color{red}{1}0000101 & \mapsto & -5 \\
\color{red}{0}0011110 & \mapsto & 30 & \color{red}{1}0011110 & \mapsto & -30 \\
\color{red}{0}1111111 & \mapsto & 127 & \color{red}{1}1111111 & \mapsto & -127
\end{array}
$$

Problèmes :

- représentation sous-optimale : deux représentations pour 0
- complexifie le calcul des opérations arithmétiques
  Exemple : pour l'addition/multiplication, 4 cas différents suivant les bits de signe

En pratique, **on n'utilise donc pas cette méthode**.

Complément à 2

Une meilleure solution (mais moins intuitive) : utiliser un bit de signe comme **offset** pour passer dans les négatifs.

C'est le système du **complément à** 2.

Cas avec 8 bits

- les 7 derniers bits définissent une valeur positive N^+ (entre 0 et $2^7 - 1 = 127$),
- le premier bit (**de signe**) dit si l'on doit soustraire $2^7 = 128$ à N^+ pour passer dans les négatifs (entre -128 et -1)

Exemples :

$$\begin{array}{lcl} 00001001 & \rightarrow & 9 \\ \textcolor{red}{1}0001001 & \rightarrow & 9 - 128 = -119 \end{array}$$

Complément à 2

Une meilleure solution (mais moins intuitive) : utiliser un bit de signe comme **offset** pour passer dans les négatifs.

C'est le système du **complément à** 2.

Exemple avec k bits

- les $k-1$ derniers bits définissent une valeur positive N^+ (entre 0 et $2^{k-1}-1$),
- le premier bit (**de signe**) dit si l'on doit soustraire 2^{k-1} à N^+ pour passer dans les négatifs (entre -2^{k-1} et -1)

Exemples avec $k = 32$:

$$
\begin{array}{rcl}
0111\cdots111 & \rightarrow & 2147483647 \\
{\color{red}1}111\cdots111 & \rightarrow & 2147483647 - 2147483648 = -1
\end{array}
$$

Complément à 2

Une meilleure solution (mais moins intuitive) : utiliser un bit de signe comme **offset** pour passer dans les négatifs.

C'est le système du **complément à** 2.

Proposition

Les opérations d'addition et de multiplication sont les mêmes entre

- *les entiers non signés,*
- *les entiers signés par le complément à deux.*

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 1 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 11 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 1111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 01111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 101111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 0101111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{rr} & 10110010 \\ + & 01111101 \\ \hline & 00101111 \end{array}$$

Exemple

Calculer l'addition des deux nombres représentés en binaire par

$$10110010 \qquad \text{et} \qquad 01111101.$$

Question que l'on pourrait se poser :

mais est-ce que l'on regarde des entiers signés ou non signés ?

Réponse : on n'a pas besoin de savoir, car l'addition se fera de la même façon.

$$\begin{array}{cc} & 10110010 \\ + & 01111101 \\ \hline & 00101111 \end{array}$$

Pareil pour la multiplication : on n'a pas besoin de distinguer le cas signé du non-signé pour multiplier deux nombres.

Complément à 2

Comment obtenir la décomposition d'un entier négatif $-N$ à partir celle de N ?

On utilise pour cela l'**opération du complément à** 2 :

1. inverser tous les bits de la représentation de N,
2. ajouter 1 (en prenant en compte les retenues).

Complément à 2

Comment obtenir la décomposition d'un entier négatif $-N$ à partir celle de N ?

On utilise pour cela l'**opération du complément à** 2 :

1. inverser tous les bits de la représentation de N,
2. ajouter 1 (en prenant en compte les retenues).

Exemples en 8 bits :

représentation de -1 à partir de celle de 1 ?

- la représentation de 1 est 00000001
- on inverse tous les bits : 11111110
- on ajoute 1 : 11111111, qui est la représentation de -1.

À retenir : la représentation binaire de -1 ne contient que des bits à 1.

Complément à 2

Comment obtenir la décomposition d'un entier négatif $-N$ à partir celle de N ?

On utilise pour cela l'**opération du complément à** 2 :

1. inverser tous les bits de la représentation de N,
2. ajouter 1 (en prenant en compte les retenues).

Exemples en 8 bits :

représentation de -15 à partir de celle de 15 ?

- la représentation de 15 est 00001111
- on inverse tous les bits : 11110000
- on ajoute 1 : 11110001, qui est la représentation de -15.

Complément à 2

Comment obtenir la décomposition d'un entier négatif $-N$ à partir celle de N ?

On utilise pour cela l'**opération du complément à** 2 :

1. inverser tous les bits de la représentation de N,
2. ajouter 1 (en prenant en compte les retenues).

Exemples en 8 bits :

représentation de -96 à partir de celle de 96 ?

- la représentation de 96 est 01100000
- on inverse tous les bits : 10011111
- on ajoute 1 : 10100000, qui est la représentation de -96.

Complément à 2

Comment obtenir la décomposition d'un entier négatif $-N$ à partir celle de N ?

On utilise pour cela l'**opération du complément à** 2 :

1. inverser tous les bits de la représentation de N,
2. ajouter 1 (en prenant en compte les retenues).

Exemples en 8 bits :

représentation de -127 à partir de celle de 127 ?

- la représentation de 127 est 01111111
- on inverse tous les bits : 10000000
- on ajoute 1 : 10000001, qui est la représentation de -127.

Notation adéquate

La représentation binaire d'un nombre n'est pas pratique car souvent longue (≈ 3 fois plus long qu'en décimal).

Exemple : 99 est associé à 1100011.

Notation adéquate

La représentation binaire d'un nombre n'est pas pratique car souvent longue (≈ 3 fois plus long qu'en décimal).

Exemple : 99 est associé à 1100011.

Cependant, comme on l'a vu, obtenir la représentation binaire d'un nombre donné en base 10 n'est pas une action immédiate.

Exemple : quelle est la représentation binaire de 57 ? Il faut exécuter l'algorithme évoqué plus haut pour le savoir.

Notation adéquate

La représentation binaire d'un nombre n'est pas pratique car souvent longue (≈ 3 fois plus long qu'en décimal).

Exemple : 99 est associé à 1100011.

Cependant, comme on l'a vu, obtenir la représentation binaire d'un nombre donné en base 10 n'est pas une action immédiate.

Exemple : quelle est la représentation binaire de 57 ? Il faut exécuter l'algorithme évoqué plus haut pour le savoir.

Pour palier ces deux problèmes, on préfère utiliser la **notation hexadécimale**.

Notation hexadécimale

Nombre en base 10 : décomposition en chiffres pouvant prendre 10 valeurs (0,1,...,9).

Nombre en base 2 : décomposition en chiffres pouvant prendre 2 valeurs (0,1).

Nombre en base 16 : décomposition en chiffres pouvant prendre 16 valeurs :

0 1 2 3 4 4 6 7 8 9 A B C D E F

Comme $16 = 2^4$, un « chiffre » hexadécimal correspond à 4 bits.

$$
\begin{array}{llll}
0 \mapsto 0000 & 4 \mapsto 0100 & 8 \mapsto 1000 & \mathrm{C} \mapsto 1100 \\
1 \mapsto 0001 & 5 \mapsto 0101 & 9 \mapsto 1001 & \mathrm{D} \mapsto 1101 \\
2 \mapsto 0010 & 6 \mapsto 0110 & \mathrm{A} \mapsto 1010 & \mathrm{E} \mapsto 1110 \\
3 \mapsto 0011 & 7 \mapsto 0111 & \mathrm{B} \mapsto 1011 & \mathrm{F} \mapsto 1111
\end{array}
$$

Du binaire à l'hexadécimal

Pour aller de la représentation binaire à la représentation hexadécimale, il suffit de **grouper les bits par** 4 et d'associer le « chiffre » hexadécimal correspondant.

Exemples : convertir 11100011 en hexadécimal

$$\underbrace{1110}_{\text{E}}\,\underbrace{0011}_{3} \qquad \mapsto \qquad \text{E3}$$

Du binaire à l'hexadécimal

Pour aller de la représentation binaire à la représentation hexadécimale, il suffit de **grouper les bits par** 4 et d'associer le « chiffre » hexadécimal correspondant.

Exemples : convertir 11010 en hexadécimal

$$\underbrace{{\color{red}000}1}_{1}\,\underbrace{1010}_{\mathsf{A}} \quad \mapsto \quad 1\mathsf{A}$$

si le nombre de bits n'est pas un multiple de 4, on rajoute des 0 devant.

Du binaire à l'hexadécimal

Pour aller de la représentation binaire à la représentation hexadécimale, il suffit de **grouper les bits par** 4 et d'associer le « chiffre » hexadécimal correspondant.

Exemples : convertir 1001110001011101 en hexadécimal

$$\underbrace{1001}_{9}\,\underbrace{1100}_{C}\,\underbrace{0101}_{5}\,\underbrace{1101}_{D} \quad \mapsto \quad 9C5D$$

De l'hexadécimal au binaire

Dans l'autre sens, on associe à chaque chiffre le groupe de 4 bits correspondants et on concatène.

Exemples : convertir AE5F en binaire

$$\mathrm{AE5F} \mapsto 1010111001011111$$

car

$$\mathrm{A} \mapsto 1010 \qquad \mathrm{E} \mapsto 1110 \qquad 5 \mapsto 0101 \qquad \mathrm{F} \mapsto 1111$$

De l'hexadécimal au binaire

Dans l'autre sens, on associe à chaque chiffre le groupe de 4 bits correspondants et on concatène.

Exemples : convertir 26E1 en binaire

$$26E1 \mapsto \cancel{00}10011011100001$$

car

$$2 \mapsto 0010 \qquad 6 \mapsto 0110 \qquad E \mapsto 1110 \qquad 1 \mapsto 0001$$

et on peut se débarasser des 0 devant l'encodage de 2 vu que c'est le premier chiffre.

Nombres en `C`

On peut écrire directement en `C` des nombres exprimés dans une autre base que 10.

Si le nombre commence par `0b`, alors il est interprété en binaire.

```c
int x = 0b1001; // équivalent à 'int x = 9;'
```

Si le nombre commence par `0x`, alors il est interprété en hexadécimal.

```c
int x = 0x1A; // équivalent à 'int x = 26;'
```

Boutisme

De façon usuelle, les chiffres d'un nombre sont écrits, de gauche à droite, du plus important au moins important :

$$135 = 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

Boutisme

De façon usuelle, les chiffres d'un nombre sont écrits, de gauche à droite, du plus important au moins important :

$$135 = 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

Mais on pourrait prendre la convention inverse et écrire à la place :

$$531 = 5 \times 01^0 + 3 \times 01^1 + 1 \times 01^2$$

Boutisme

De façon usuelle, les chiffres d'un nombre sont écrits, de gauche à droite, du plus important au moins important :

$$135 = 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

Mais on pourrait prendre la convention inverse et écrire à la place :

$$531 = 5 \times 01^0 + 3 \times 01^1 + 1 \times 01^2$$

- la première convention est le **gros-boutisme** (*big-endian*)
- la deuxième convention est le **petit-boutisme** (*little-endian*)

Boutisme et ordinateurs

Comme l'unité de base est l'octet dans les ordinateurs, le boutisme est paramètre pertinent uniquement pour les types à k octets avec $k > 1$ (`int`, `long`, *etc.*).

Boutisme et ordinateurs

Comme l'unité de base est l'octet dans les ordinateurs, le boutisme est paramètre pertinent uniquement pour les types à k octets avec $k > 1$ (`int`, `long`, *etc.*).

Exemple :

```
int x = 1;
char *ptr = (char*) &x;
printf("ptr[0]: %d et ptr[3]: %d",(int)ptr[0],(int)ptr[3]);
// que va-t-il s'afficher ?
```

Boutisme et ordinateurs

Comme l'unité de base est l'octet dans les ordinateurs, le boutisme est paramètre pertinent uniquement pour les types à k octets avec $k > 1$ (`int`, `long`, *etc.*).

Exemple :

```
int x = 1;
char *ptr = (char*) &x;
printf("ptr[0]: %d et ptr[3]: %d",(int)ptr[0],(int)ptr[3]);
// que va-t-il s'afficher ?
```

La plupart des ordinateurs utilisent le petit-boutisme (*little-endian*) :

```
ptr[0]: 1 et ptr[3]: 0 // le '1' est sur le premier octet
```

C'est donc l'inverse de la convention habituelle pour la base 10 !

Boutisme et ordinateurs

Comme l'unité de base est l'octet dans les ordinateurs, le boutisme est paramètre pertinent uniquement pour les types à k octets avec $k > 1$ (`int`, `long`, *etc.*).

Exemple :

```
int x = 1;
char *ptr = (char*) &x;
printf("ptr[0]: %d et ptr[3]: %d",(int)ptr[0],(int)ptr[3]);
// que va-t-il s'afficher ?
```

Mais un code `C` portable doit prendre en compte les ordinateurs gros-boutistes aussi :

```
ptr[0]: 0 et ptr[3]: 1 // le '1' est sur le dernier octet
```

Bits et octets

Nombres négatifs

Notation hexadécimale

Boutisme

Opérations sur les bits

Opérations logiques

Vous connaissez déjà les opérateurs logiques en `C` :

`&&`, `||`, `!`

Ces opérateurs travaillent sur des `int`

- une valeur de 0 est considérée comme fausse
- une valeur $\neq 0$ est considérée comme vraie

Exemple : tableau de `&&`

`x`	`y`	`x && y`
0	0	0
$\neq 0$	0	0
0	$\neq 0$	0
$\neq 0$	$\neq 0$	1

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `&` : cet opérateur applique **bit à bit** la table logique du « et » :

P	Q	P et Q
0	0	0
1	0	0
0	1	0
1	1	1

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `&` : cet opérateur applique **bit à bit** la table logique du « et » :

P	Q	P et Q
0	0	0
1	0	0
0	1	0
1	1	1

Exemples (`x` et `y` sont ici des `unsigned char`) :

`x`	01100111
`y`	01011101
`x & y`	01000101

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `|` : cet opérateur applique **bit à bit** la table logique du « ou » :

P	Q	P ou Q
0	0	0
1	0	1
0	1	1
1	1	1

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `|` : cet opérateur applique **bit à bit** la table logique du « ou » :

P	Q	P ou Q
0	0	0
1	0	1
0	1	1
1	1	1

Exemples (`x` et `y` sont ici des `unsigned char`) :

`x`	01100111
`y`	01011101
`x	y`	01111111

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `~` : cet opérateur applique **bit à bit** la table logique du « non » :

P	non P
0	1
1	0

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `~` : cet opérateur applique **bit à bit** la table logique du « non » :

P	non P
0	1
1	0

Exemples (`x` est ici un `unsigned char`) :

`x`	01100111
`~x`	10011000

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `^` : applique **bit à bit** la table logique du « ou exclusif » (*xor*) :

P	Q	P xor Q
0	0	0
1	0	1
0	1	1
1	1	0

Opérateurs bit à bit

On a des opérateurs analogues qui travaillent sur la représentation binaire :

`&`, `|`, `~`, `^`

Opérateur `^` : applique **bit à bit** la table logique du « ou exclusif » (*xor*) :

P	Q	P xor Q
0	0	0
1	0	1
0	1	1
1	1	0

Exemples (`x` et `y` sont ici des `unsigned char`) :

`x`	01100111
`y`	01011101
`x ^ y`	00111010

Complément à 2 revisité

Que vaut `x + ~x` en général ?

Complément à 2 revisité

Que vaut `x + ~x` en général ?

Exemple : pour `x` = 01101010

`x`	01101010
`~x`	10010101
`x + ~x`	11111111

Complément à 2 revisité

Que vaut `x + ~x` en général ?

Exemple : pour `x` $= 01101010$

`x`	01101010
`~x`	10010101
`x + ~x`	11111111

Bilan :

$$\texttt{x + \~{}x} = 11...11 = \texttt{-1}$$

Complément à 2 revisité

Que vaut `x + ~x` en général ?

Exemple : pour `x` $= 01101010$

`x`	01101010
`~x`	10010101
`x + ~x`	11111111

Bilan :

`x + ~x` $= 11...11 =$ `-1`

On retrouve l'opération du complément à 2 :

`~x + 1` $=$ `-x`

Opérateurs de décalage

Deux opérateurs importants qui agissent sur la représentation binaire sont les **opérateurs de décalage** `<<` et `>>`.

Ils translatent les bits de la représentation binaire à gauche et à droite.

Exemples sur 8 bits :

`x`	00000101	5
`x << 1`	00001010	$10 = 5 \times 2$
`x << 2`	00010100	$20 = 5 \times 4$
`x << 3`	00101000	$40 = 5 \times 8$
`x << 4`	01010000	$80 = 5 \times 16$

On remarque que calculer `x << k` revient à multiplier `x` par 2^{k}.

Opérateurs de décalage

Deux opérateurs importants qui agissent sur la représentation binaire sont les **opérateurs de décalage** `<<` et `>>`.

Ils translatent les bits de la représentation binaire à gauche et à droite.

Exemples sur 8 bits :

`x`	00001100	12
`x >> 1`	00000110	$6 = 12/2$
`x >> 2`	00000011	$3 = 12/4$
`x >> 3`	00000001	$1 = 12/8$
`x >> 4`	00000000	$0 = 12/16$

On remarque que calculer `x >> k` revient à diviser (entièrement) `x` par 2^{k}.

Puissance de 2

En particulier, on peut calculer les premières puissances de 2 rapidement :

```
1 << k
```

En effet, comme on l'a vu, cela revient à calculer $1 \times 2^{\texttt{k}} = 2^{\texttt{k}}$.

Puissance de 2

En particulier, on peut calculer les premières puissances de 2 rapidement :

`1 << k`

En effet, comme on l'a vu, cela revient à calculer $1 \times 2^{\texttt{k}} = 2^{\texttt{k}}$.

Bien sûr, il ne faut pas dépasser la capacité du type et le bit de signe.

```c
int k = ??;
printf("%d",1 << k);
```

`k`	`1 << k`
0	1
3	8
30	1073741824
31	-2147483648
32	0

Options

On peut utiliser la représentation binaire pour passer des « options » (*flags* en anglais) à des fonctions.

Le nombre d'options que l'on peut gérer est égale au nombre de bits du type que l'on utilise.

À chaque option, on lui associe une puissance de 2. On combine alors ces options avec l'opérateur `|`.

Le cas de la SDL

La SDL est une librairie `C` permettant de coder des applications multimédia (écran, clavier, joystick, 2D, son, *etc.*).

Elle est organisée en modules gérant chacun un domaine que l'on peut choisir de charger ou pas au démarrage de l'application.

- `SDL_video.h` : gère les fenêtres et la 2D
- `SDL_events.h` : gère les entrées (clavier, souris, *etc.*)
- `SDL_audio.h` : gère le son
- `SDL_timer.h` : gère le temps (chronomètres, temporisations, *etc.*)

Chargement des modules

Pour pouvoir utiliser les différents modules, il faut les charger avec la fonction

```
int SDL_Init(unsigned int flags);
```

Le paramètre `flags` définit les modules à charger. Pour le définir, on combine des constantes représentant les modules par l'opération `|`.

```
#define SDL_INIT_TIMER  0b0001
#define SDL_INIT_AUDIO  0b0010
#define SDL_INIT_VIDEO  0b0100
#define SDL_INIT_EVENTS 0b1000
```

Exemple : charger `AUDIO` et `VIDEO` :

```
SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO);
```

Options et représentation binaire

À chaque constante est associé un bit sur les 32 :

```
#define SDL_INIT_TIMER  0b0001
#define SDL_INIT_AUDIO  0b0010
#define SDL_INIT_VIDEO  0b0100
#define SDL_INIT_EVENTS 0b1000
```

Quand on les combine avec `|`, on obtient un nombre dont chaque bit à 1 représente une option :

`flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO` : deux bits à 1 (le 2ème et le 3ème)

On peut alors tester si une option est demandée avec `&` :

```
if(flags & SDL_INIT_VIDEO){
  // l'option VIDEO est activée
}
```