Représentation des nombres

Alain MÉRIGOT

Université Paris Saclay

Plan

Bits et mots

Élement d'information : **bit** (*binary digit*)
Deux valeurs : `vrai` et `faux`, ou 0 et 1.

Bits regroupés en *mots* de n bits, généralement numérotés de $n-1$ à 0

A mot formé de n bits a_{n-1} a_{n-2} $\ldots$ a_1 a_0

bit de poids fort
most significant bit
MSB

bit de poids faible
least significant bit
LSB

Mot de n bits : n éléments pouvant prendre 2 valeurs $\Longrightarrow$ 2^n configurations différentes.

Problème du codage : définir une application, de préférence bijective, associant des mots à des élements d'autres ensembles : entiers naturels $\mathbb{N}$, entiers relatifs $\mathbb{Z}$ ou ou nombres réels $\mathbb{R}$.

Plan

Codage des entiers naturels

Soit un mot A codé sur n bits : $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$
On peut associer au mot A, sa valeur en base 2 $A' \in \mathbb{N}$ telle que que chaque bit a_i de A corresponde à la puissance 2^i dans la décomposition en base 2 de A'

Un mot $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$ est le **code binaire naturel** (CBN) sur n bits du nombre A' si

$$\begin{aligned} A' &= 2^{n-1} \times a_{n-1} + 2^{n-2} \times a_{n-2} + \cdots + 2 \times a_1 + a_0 \\ &= \sum_{i=0}^{n-1} 2^i \times a_i \end{aligned} \tag{1}$$

Nous identifierons en général A et A'.

$$A \text{ sur } n \text{ bits} \implies \underbrace{0}_{\text{code (000...0)}} \leq A \leq \underbrace{2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1}_{\text{code (111...1)}}$$

$$n = 8 : 0 \leq A \leq 2^8 - 1 = 255$$

$$n = 16 : 0 \leq A \leq 2^{16} - 1 = 65\,535$$

$$n = 32 : 0 \leq A \leq 2^{32} - 1 = 4\,294\,967\,295$$

Codage des entiers naturels (cont.)

Calcul de la valeur numérique entière associée à un mot de n bits
$\Longrightarrow$ utilisation directe de la formule (1).

Calcul du CBN d'un nombre entier naturel : différentes méthodes

Exemple : reste des divisions successives de A par 2

```
    ▷ Soit A le nombre à coder en binaire
1   pour i ← 0 à n − 1 faire
2       bit[i] ← A % 2;          ▷ Le reste de la division par 2 du nombre
3       A ← A ÷ 2;
4   fin pour
    ▷ Le tableau bit[n − 1]...bit[0] contient le code binaire naturel de A
```

Exemple : code de 57 sur 6 bits

$57 \div 2 = 28$ reste **1** (bit 0)
$28 \div 2 = 14$ reste **0** (bit 1)
$14 \div 2 = 7$ reste **0** (bit 2)
$7 \div 2 = 3$ reste **1** (bit 3)
$3 \div 2 = 1$ reste **1** (bit 4)
$1 \div 2 = 0$ reste **1** (bit 5)

D'où le code 111001 (car 57=32+16+8+0×4+0×2+1)

Plan

Code hexadécimal

binaire : code souvent long et générateur d'erreurs

$\Longrightarrow$ codage hexadécimal (base 16)

Codage d'un ensemble de 4 bits[1] par un digit d'un alphabet formé par les chiffres décimaux 0 . . . 9 et les lettres A…F (ou a…f)

Déc.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin.	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hexa.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

En C, utilisation du préfixe `0x`

Exemple : $254 = 1111\ 1110 = 15 \times 16^1 + 14 \times 16^0 \equiv$ `0XFE` (ou `0xfe`)

[1] souvent appelé *nibble* en anglais

Plan

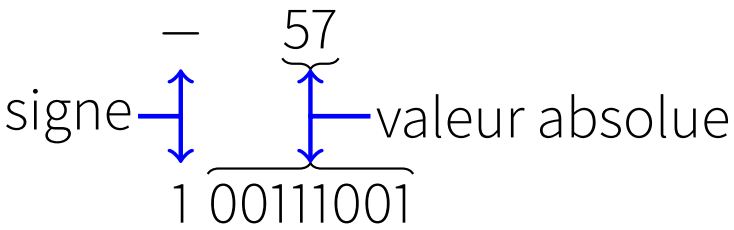
Plan

Nombres entiers signés

signe-valeur absolue

Pas de représentation naturelle pour les nombres signés ($\in \mathbb{Z}$).

Codage « signe-valeur absolue » (SVA)



Un bit (généralement le MSB) indique le signe (0 pour +[2] et 1 pour −), les autres la valeur absolue

Si A codé en SVA sur n bits, $|A|$ est codé sur $n-1$ bits et donc $|A| \leq 2^{n-1}-1$
D'ou $-(2^{n-1}-1) \leq A \leq 2^{n-1}-1$

Inconvénients :

- l'ordre des codes ne correspond pas toujours à celui des nombres :
  - si $A \geq 0$, $\text{code}(A) < \text{code}(-A)$ (inévitable si + codé par 0)
  - si $A, B \geq 0, A > B \implies \text{code}(A) > \text{code}(B)$,
    si $A, B < 0, A > B$ $(|A| < |B|) \implies \text{code}(A) < \text{code}(B)$
- deux représentations de zéro : $+0(0\,00\ldots0)$ et $-0(1\,00\ldots0)$ (ce qui complexifie, par exemple, le test d'égalité de deux nombres)

[2] pour que le code sur n bits de A (non signé) soit identique à celui de $+A$ (signé)

Plan

On appelle **complément à deux sur n bits** de A, le nombre $C_2(A) = 2^n - A$.

On appelle **code en complément à deux sur n bits** (CC_2) d'un nombre A :

si $0 \leq A \leq 2^{n-1} - 1$ **:** code binaire naturel sur n bits de A

si $-2^{n-1} \leq A < 0$ **:** code binaire naturel sur n bits de $C_2(|A|)$

Remarques :

1. Si $A < 0$, $C_2(|A|) = 2^n - |A| = 2^n + A$.
   Le code complément à 2 revient à ajouter 2^n aux nombres négatifs pour les rendre positifs et les coder en binaire naturel.
   Ceci permet de simplifier *considérablement* les opérations arithmétiques.
2. Si $A \geq 0$, comme $A < 2^{n-1}$, le MSB a_{n-1} sera à zéro
   Si $A < 0$, Comme $|A| \leq 2^{n-1}$, alors $C_2(A) = 2^n - |A| \geq 2^{n-1}$ et le MSB a_{n-1} du CBN de $C_2(A)$ est à 1.
   Le MSB a_{n-1} du CC_2 d'un nombre A est donc à 0 si $A \geq 0$ et à 1 si $A < 0$ et joue le rôle d'un **bit de signe**.

Quelques propriétés du complément à 2

- $C_2(C_2(A)) = 2^n - (2^n - A) = A$
- $A + C_2(A) = 2^n$
- $C_2(A) = \overline{A} + 1$
  En effet $A + \overline{A} = \sum_{i=0}^{n-1} a_i \times 2^i + \sum_{i=0}^{n-1} \overline{a_i} \times 2^i$
  $= \sum_{i=0}^{n-1} (a_i + \overline{a_i}) \times 2^i = \sum_{i=0}^{n-1} 2^i = 2^n - 1$
  $\Longrightarrow \overline{A} + 1 = 2^n - A = C_2(A)$
- une seule représentation de 0 :
  $-0 = C_2(0) = 2^n - 0 = 0$ (sur n bits)

A est codable en CC_2 sur n bits $\equiv -2^{n-1} \leq A \leq 2^{n-1} - 1$
Sur 8 bits : $-128 \leq A \leq 127$
Sur 16 bits : $-32768 \leq A \leq 32767$
Sur 32 bits : $-2147483648 \leq A \leq 2147483647$

Pour retrouver la valeur d'un mot $A = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ codé en complément à deux :

$$\begin{array}{lllll} A & = & \text{CBN}(A) & & \text{si } A \geq 0 \quad (a_{n-1} = 0) \\ & = & 0 \times 2^{n-1} & + \sum_{i=0}^{n-2} a_i \times 2^i & \end{array}$$

$$\begin{array}{llllll} A & = & -2^n & + \text{CBN}(A) & & \text{si } A < 0 \quad (a_{n-1} = 1) \\ & = & -2^n & + 1 \times 2^{n-1} & + \sum_{i=0}^{n-2} a_i \times 2^i & \\ & = & & 1 \times (-2^{n-1}) & + \sum_{i=0}^{n-2} a_i \times 2^i & \end{array}$$

La valeur numérique d'un nombre $A = (a_{n-1}, ..., a_0)$ codé en CC_2 sur n bits est :

$$A = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} 2^i \times a_i \qquad (2)$$

Tout se passe comme si le bit de poids fort correspondait à un poids -2^{n-1}.

Plan

Nombres entiers signés (cont.)

codes par excès

Le code complément à deux respecte l'ordre relatif des nombres de même signe,
mais les nombres positifs ont tous un code inférieur aux nombres négatifs.

Dans certains il est important de respecter cet ordre : **codes par excès**

On ajoute un décalage fixe k (excès) à *tous* les nombres à coder
et on code en binaire naturel $A + k$

Sur n bits, il faut $0 \leq A + k \leq 2^n - 1$
Un code par excès k peut coder $-k \leq A \leq 2^n - k - 1$

L'ordre relatif des nombres est parfaitement respecté, mais

- le code d'un nombre positif est différent de son code en binaire naturel
- les opérations arithmétiques ($+$, $-$, etc) sont plus complexes, car il faut retrancher l'excès avant d'effectuer le calcul.

Nombres entiers signés (cont.)

récapitulatif

Codage	Binaire naturel	Signe-Valeur abs.	Complém. à 2	Excès 8
0000	0	+0	0	−8
0001	1	+1	+1	−7
0010	2	+2	+2	−6
0011	3	+3	+3	−5
0100	4	+4	+4	−4
0101	5	+5	+5	−3
0110	6	+6	+6	−2
0111	7	+7	+7	−1
1000	8	−0	−8	0
1001	9	−1	−7	+1
1010	10	−2	−6	+2
1011	11	−3	−5	+3
1100	12	−4	−4	+4
1101	13	−5	−3	+5
1110	14	−6	−2	+6
1111	15	−7	−1	+7

croissant

croissant

décroissant

croissant

croissant

Nombres entiers signés (cont.)

récapitulatif

taille	nom	signé/non signé	en C	`stdint.h`
8 bits	octet	signé	`char`	`int8_t`
	(*byte*)	non signé	`unsigned char`	`uint8_t`
16 bits	demi mot	signé	`short`	`int16_t`
	(*half*)	non signé	`unsigned short`	`uint16_t`
32 bits	mot	signé	`int` (ou `long`)	`int32_t`
	(*word*)	non signé	`unsigned` (`int`)	`uint32_t`
64 bits	double mot	signé	`long long`	`int64_t`
		non signé	`unsigned long long`	`uint64_t`

NB : le standard C n'impose pas de taille pour les types entiers.
Il impose juste `sizeof(char)`$\leq$`sizeof(short)`$\leq$`sizeof(int)`
$\leq$`sizeof(long)`$\leq$`sizeof(long long)`.
Sur certains compilateurs `sizeof(int)`=16 ou `sizeof(long)`=64, etc.

En C99, introduction de `<stdint.h>` qui permet d'avoir des types de taille définie.

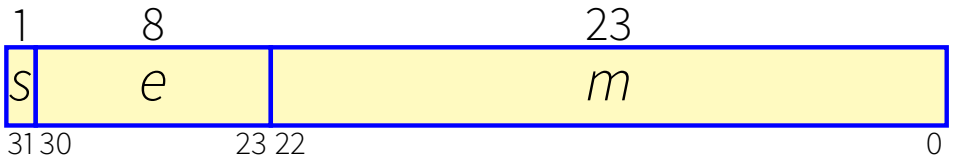
Plan

Nombres flottants

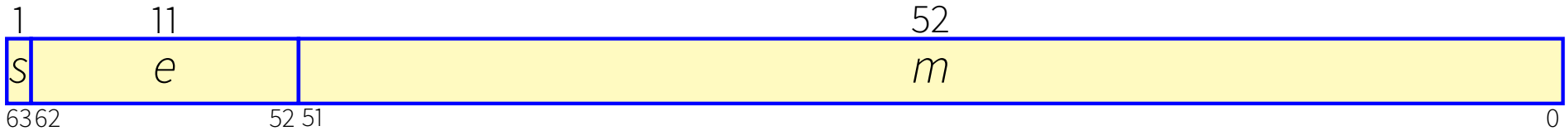
Pour de nombreuses applications, nécessité d'avoir 1/ des nombres fractionnaires et 2/ de dynamique élevée : *nombres à virgule flottante* (ou *nombres flottants*).
Nombre flottant $N \equiv$ triplet (s, m, e) *signe*, *mantisse*, *exposant* tel que
$N = (-1)^s \times m \times b^e$
Généralement $b = 2$

Standardisation **IEEE-754** : spécifie les modes de codage de s, m et e pour divers formats.

simple précision (32 bits) (`float` en C) (s : 1 bits, e : 8 bits, m : 23 bits)



double précision (64 bits) (`double` en C) (s : 1 bits, e : 11 bits, m : 52 bits)



Il existe également précision étendue (80 bits), quadruple précision (128 bits), octuple précision (256 bits), demi précision (16 bits), etc.

Pour assurer représentation unique, mantisse normalisée : $m \in [1, 2[$
Représente un nombre $1.000\ldots0 \leq M \leq 1.111\ldots1$

Comme le bits de poids fort est toujours à 1, il n'est pas codé : « bit caché »
Les bits a_i correspondent à des puissances négatives de 2.

$$\begin{aligned} \text{Si } M &= 1.a_{n-1}a_{n-2}\ldots a_1a_0 \\ M &= 1 + a_{n-1} \times 2^{-1} + a_{n-2} \times 2^{-2} + \cdots + a_1 \times 2^{1-n} + a_0 \times 2^{-n} \\ &= 1 + \sum_{i=n-1}^{0} a_i \times 2^{i-n} \end{aligned}$$

Exemple : $1.101 = 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 1.625$

Le poids de l'unité en dernière position (ULP *unit in the last place*) donne la **précision** du code.
En simple précision, $2^{-23} \approx 1.2 \times 10^{-7}$ (6 à 7 décimales précises)

Les exposants sont codés *par excès* pour permettre une comparaison simple.

En simple précision ($e = 8$ bits) : excès de 127
En double précision ($e = 11$ bits) : excès de 1023

Pour un code IEEE 754 dont l'exposant est sur l bits, l'excès sera de $2^{l-1} - 1$

En simple précision, ajouter 127 pour coder un exposant
retrancher 127 pour obtenir l'exposant depuis son code

Le code des exposants peut varier de 1 (2^{-126}) à 254 (2^{127}).

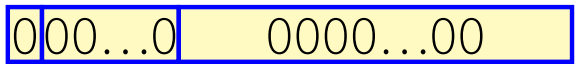
Les exposants déterminent la **dynamique** des nombres.

Nombres flottants (cont.)

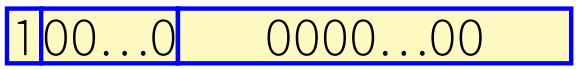
Les exposants extrèmes (0 et 255) codent des cas particuliers.

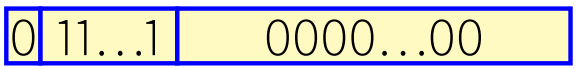
Pour un nombre flottant simple précision N codé par le triplet (s, e, m)

	$e = 0$	$1 \leq e \leq 254$	$e = 255$
$m = 0$	$N = (-1)^s \times 0.0$	$N = (-1)^s \times 1.0 \times 2^{e-127}$	$N = (-1)^s \times \infty$
$m \neq 0$	$N = (-1)^s \times 0.m \times 2^{-126}$	$N = (-1)^s \times 1.m \times 2^{e-127}$	**NaN** (Not A Number)

zéro est représenté par 

(comme les codes entiers)

Il existe un code pour -0.0 : 

Codage de $\pm\infty$ (`inf` en C) 

($e = 255, m = 0$)

$\pm$zéro correspond à des nombres trop petits pour être codables en IEEE 754, $\pm\infty$ à des nombres trop grands.

Les codes $e = 255$ et $m \neq 0$ sont dit *not a number* (**nan** en C)
$\rightarrow$ détection des erreurs (débordements numériques, $0 \times \infty$, $\div 0$)
ou codage d'informations de manière non spécifiée par la norme.
Toute opération dont un argument est à **nan** génère un **nan** en sortie.

Nombres flottants (cont.)

Les codes d'exposant nul sont dits **dénormalisés** (*subnormals* ou *denormals*)

Le *bit caché* est à 0 (au lieu de 1)
Permet un accroissement de la *dynamique* (vers les exposants < 0) en réduisant la *précision*

Exemple : $m = \underbrace{0.0000010111001\ldots010}$

dynamique accrue de 2^{-6} ↑ ↑ mais précision de 18 bits (contre 23+1)

Les nombres dénormalisés complexifient significativement les traitements flottants.
Exemple : Sur le Pentium, un calcul où un des opérandes ou le résultat est dénormalisé nécessite > 100 cycles (contre 3-20 cy suivant l'opérateur pour des nombres non dénormalisés).
Les systèmes embarqués utilisent souvent des versions simplifiées de la norme IEEE-754

FTZ (*Flush To Zero*) un résultat dénormalisé est remplacé par zéro.

DAZ (*Denormals Are Zero*) un opérande dénormalisé est traité comme 0.0

Nombres flottants (cont.)

Plus grands et plus petits nombres codables en simple et en double précision IEEE 754.

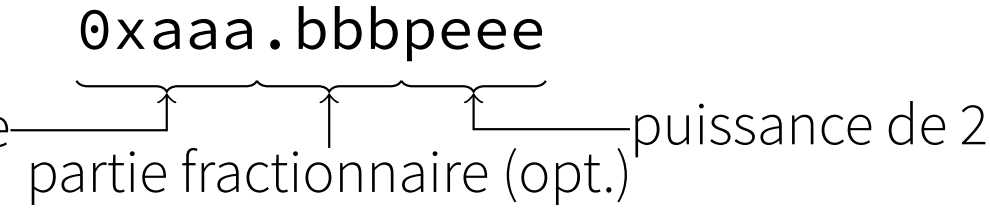
	Plus petit nombre normalisé	Plus petit nombre dénormalisé	Plus grand nombre
Simple précision	1.175494e-38	1.401298e-45	3.40282e+38
Double précision	2.225074e-308	4.940656e-324	1.79769e+308

Nombres flottants (cont.)

Depuis le standard C99, possibilité d'entrer et d'afficher en C les nombres flottants dans un format hexadécimal.

- prise en compte du codage d'un nombre
- évite les erreurs d'arrondi dans la conversion décimale ⇔ flottant

Pour coder le nombre aaa.bbb 2^{eee} (en hexadécimal) :



Spécifieur `%a` de `printf()` pour afficher les flottants hexadécimaux.

```
float  fmax=0x1.fffffep127;          // 1.111...11 2^127
double dmax=0x1.fffffffffffffp1023; // 1.111...11 2^1023
printf ("Plus grand float : %g (code:%a)\n",fmax,fmax);
printf ("Plus grand double : %g (code:%a)\n",dmax,dmax);
// affiche :
// Plus grand float : 3.40282e+38 (code:0x1.fffffep+127)
// Plus grand double : 1.79769e+308 (code:0x1.fffffffffffffp+1023)
```

Plan

Le codage en *virgule fixe* consiste à coder un nombre fractionaire avec des puissances positives et négatives de 2.
Historiquement utilisé (et encore utile dans certaines réalisation matérielles).
Intermédiaire utile pour convertir un nombre décimal en flottant.

Conversion décimal $\rightarrow$ flottant

1. coder la valeur absolue du nombre en binaire naturel avec une virgule fixe décomposition en puissances positives (pour la partie entière) et négatives (pour la partie fractionnaire) de 2.
2. décaler ce code de k vers la droite ou la gauche ($k < 0$) pour avoir un "1" à gauche la virgule
   Pour garder la valeur numérique, multiplier par 2^k
3. réaliser le codage suivant la norme IEEE 754

Exemple : $N = -1.275 \times 10^1$ à coder en flottant IEEE simple précision.
Passage en virgule fixe en l'écrivant $N = -12.75$.

1. codage en virgule fixe en base 2 de $|N|$:

$$\begin{aligned} |-12.75| &= 12.75 \\ &= 8 + 4 + 0.5 + 0.25 \\ &= 2^3 + 2^2 + 2^{-1} + 2^{-2} \\ &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &\quad +1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + \cdots \\ &= 1100.110\ldots \end{aligned}$$

2. normalisation du nombre entre 1.0 et 2.0
   décalage de 3 positions vers la droite pour avoir un nombre normalisé :
   $12.75 = 1.10011 \times 2^{+3}$
3. codage en flottant IEEE :
   $m = (1.)100110\ldots$
   $e + 127 = 130 = 10000010$
   $s = 1$
   d'où le code :

1	10000010	10011000000000000000000

Nombres flottants (cont.)

Dans le cas général, conversion d'un nombre fractionnaire décimal → binaire :

▷ *Soit N le nombre à coder en binaire. On suppose* $N < 1$

1 **pour** $i \leftarrow n-1$ **à** 0 **faire**
2 $N \leftarrow N \times 2$;
3 **si** $N \geq 1$ **alors**
4 bit[i] $\leftarrow 1$;
5 $N \leftarrow N - 1$;
6 **sinon**
7 bit[i] $\leftarrow 0$;
8 **fin si**
9 **fin pour**

▷ *Le code de N est 0.bit[n − 1]...bit[0]*

Exemple : codage en binaire fractionnaire $0.1_d = 0 \times 10^0 + 1 \times 10^{-1}$.

i	N	$2 \times N$	$N \geq 1$?	$N-1$	bit[i]
$n-1$	0.1_d	0.2_d	non		0
$n-2$	0.2_d	0.4_d	non		0
$n-3$	0.4_d	0.8_d	non		0
$n-4$	0.8_d	1.6_d	oui	0.6_d	1
$n-5$	0.6_d	1.2_d	oui	0.2_d	1
$n-6$	0.2_d	0.4_d	non		0
etc.					

L'étape $n-6$ étant identique à $n-2$, itération infinie.

$0.1_d = 0.0\underbrace{0011}\underbrace{0011}\underbrace{0011}\ldots$

Pas de représentation binaire exacte de 0.1_d.

Plan

Codage des caractères

Codage normalisé des caractères : ASCII 7 bits (1963) (ANSI X3.4 1986)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	‖	}	~	DEL

Les codes `0x00` à `0x1F` sont des caractères spéciaux.
Pour la plupart inutiles actuellement sauf NUL (`'\0'`) BEL (`'\g'`), BS (*backspace*, `'\b'`), HT (tabulation, `'\t'`), LF (*line feed*, `'\n'`) et ESC (*escape*, `'\e'`).

Étendu par l’ISO (ISO 8859) pour caractères accentués, symboles mathématiques, etc.

Actuellement, norme *unicode* pour la plupart des caractères internationaux.
variante **utf-8** (*Universal Character Set Transformation Format 8 bits*) utilise

- un octet pour les 128 premiers caractères (ASCII)
- 2 à 4 octets pour extension aux caractères internationaux.