Architecture et Système

Stefan Schwoon

Cours L3, 2025/2026, ENS Paris-Saclay

Entiers

Les entiers (tels qu'on les utilise dans les langages de programmation habituels) sont stockés dans un mot de taille fixe (typiquement 8, 16, 32 bits).

Types en C: `char`, `short int`, `int`, `long int`, `long long int`

En C, les tailles de ces types ne sont pas précisement définis par le langage de C, seulement leurs tailles minmales : 8, 16, 16, 32, 32, où `int` est un mot de registre.

On peut obtenir les valeurs concrètes avec `sizeof(char)` etc.

Entiers avec/sans signe

Un mot de n bits peut représenter 2^n valeurs différentes. Les opérations arithmétiques travaillent implicitement modulo 2^n.

Deux interprétations :

`unsigned` (sans signe), le domaine est de 0 à $2^n - 1$.

`signed` (avec signe) / complément à deux : le domaine est de -2^{n-1} à $2^{n-1} - 1$

Pour les valeurs non-négatives, le bit le plus significatif (MSB) est de 0.
Pour les valeurs négatives, le MSB est de 1.

On obtient la représentation de $-i$ en prenant la négation (bit par bit) de i, puis en rajoutant 1. P.ex., $-1 \mathrel{\widehat{=}} 11\cdots1$ et $-2^{n-1} \mathrel{\widehat{=}} 10\cdots0$.

Big vs little endian

Un mot de taille > 8 s'étend sur plusieurs octets, p.ex. avec 32 bits:

12	34	56	78

Sous l'interprétation big-endian, ceci représente $(12345678)_{16}$ (les bits les plus significatifs sont dans le premier octet).

Little-endian: c'est l'inverse, on interpréte cela comme $(78563412)_{16}$.

Le mode d'interprétation devient important lors des échanges des données binaires (fichiers, réseau). P.ex., l'Internet protocol (IP) définit cet ordre comme big-endian.

Fonctions en C : `ntohl`, `ntohs`, `htonl`, `htons`

Valeurs réelles

Les valeurs réelles sont typiquement représentées dans un format virgule flottante, dans un mot de taille fixe, avec une précision limitée.

Idée en général : tuple $\langle s, m, e \rangle$ avec l'interprétation $\pm 2^e \cdot m$.

s est le signe (un seul bit, `0` non-négatif, `1` négatif);

m est la mantisse;

e est l'exposant.

Compromis entre précision des valeurs représentables et taille de leur représentation. Exemple: Format des `float`:

s	$e_7 \cdots e_0$	$m_{22} \cdots m_0$

IEEE 754

Le standard le plus important qui règle les virgules flottants s'appele IEEE 754.

Ce standard règle les problèmes suivants :

- Repartition des bits entre taille et mantisse
- Représentation canonique, on élimine les représentations non uniques : $\langle s, m, e\rangle \equiv \langle s, 2m, e-1\rangle$
- Traitement des cas spéciaux (division par zéro) et des arrondis
- Algorithmes pour les opérations arithmétiques

En C : `float` en C = IEEE 754, 32 bit; `double` = IEEE 754 (64-bit).

Dans le suivant, on discutera la partie 32 bit, les autres parties étant similaires.

IEEE 754 (variante 32-bit)

IEEE 754 spécifie les conventions suivantes :

Signe : 1 bit; 0 pour positif, 1 pour négatif

Exposant : 8 bits, interprété sans signe, mais décalé de 127. P.ex., $(10000001)_2 = 129$, mais l'exposant effectif est de 2. Si tous les bits de l'exposant sont 1, voir ci-dessous.

Mantisse : 23 bits, interprété comme $1 + (m/2^{23})$, ce qui donne un résultat dans $[1, 2)$.

Remarques :

Cette interprétation de la mantisse garantit une représentation unique.

Cas spéciaux, si exposant est 1111 1111 : $\pm\infty$ (avec $m = 0$)
ou NaN (not a number, avec $m \neq 0$)

Addition dans les flottantes

Le standard IEEE définit la procédure à suivre pour effectuer des opérations arithmétiques (comment arrondir, comment traiter des cas spéciaux, ...).

Discutons le cas d'une addition:

Soient $x = 2^{e_x} \cdot m_x$ et $y = 2^{e_y} \cdot m_y$ et $x > y$
(par simplicité on suppose qu'ils sont tous les deux positifs).

P.ex. $x = 2.5$ et $y = 0.75$, du coup,
$e_x = 1, m_x = 1.25, e_y = -1, m_y = 1.5$.

Représentation binaire :

0	1000 0000	0100 0000 0$\cdots$0
0	0111 1110	1000 0000 0$\cdots$0

On isole désormais la mantisse, mais en prenant compte de la “une cachée” :

$$\boxed{\begin{array}{l} 1\ 0100\ 0000\ 0\cdots\ 0 \\ 1\ 1000\ 0000\ 0\cdots\ 0 \end{array}}$$

L’exposant d’y est moins grand que celui de x de 2.
Du coup, on décale la mantisse d’y de 2 positions.

$$\boxed{\begin{array}{l} 1\ \ 0100\ \ 0000\ 0\cdots\ 0 \\ 0\ 0\ 110\ 0000\ 0\cdots\ 0 \end{array}}$$

L’addition des deux mantisses donne désormais :

$$\boxed{1\ 1010\ 0000\ 0\cdots\ 0}$$

Comme le résultat n'excède pas les 24 bits (une cachée plus 23 bits), on garde l'exposant de x et on enlève simplement l'une cachée de i_Z pour obtenir la mantisse du résultat.

0	1000 0000	1010 0000 0$\cdots$0

Si l'addition des mantisses avait débordé les 24 bits, il aurait d'abord fallu décaler i_Z à droit par une position (perdant un bit de précision).

Problèmes de virgule flottante

Précision limitée : certaines valeurs “rondes” comme 0.1 ou 2.3 ne sont pas représentables.

Erreurs d’arrondi; p.ex. $x + y$ donne x si x beaucoup plus grand qu’y.

Du coup, certaines lois comme distributivité ne sont plus valables.

Codage et traitement des caractères

Importance : échange de données, comparaisons, enjeux socio-culturel

Complications:

- beaucoup de normes différents et partiellement compatibles ;
- ignorance ou indifférence des programmeurs/utilisateurs/institutions
- difficile à debugger

Sujet connexe : fonctionnement de base d'un terminal

Problème de codage

Normes pour représentation de texte

Dans un premier temps, un standard d'encodage affecte des caractères à des entiers, appelés *points de code*. P.ex. `A` ≙ 65, `$` ≙ 36 etc

Les standards les plus importants :

ASCII : 7 bit

Latin-1 (= ISO-8859-1) : 8 bit

Unicode : un million de caractères, évolue encore

Solution technique pour représenter les points de code:

un octet par caractère pour ASCII/Latin-1, plus compliqué pour Unicode !

Tableau ASCII (de Wikipédia)

PDF : en [archive] v · d · m	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
U+0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
U+0020	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
U+0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
U+0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
U+0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
U+0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
U+0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Codes à 7 ou 8 bits

ISO-646 in 1963/1972, version américaine : ASCII

codes de 7 bits, caractères de contrôle avec codes < 32 ;

suffit pour programmer et pour écrire du texte anglais, mais pas plus.

Codes de 8 bit :

Codepage 437 (et d'autres), dans les IBM PC (1981) ; permet de dessiner des tableaux en mode texte.

ISO-8859 (à partir de 1985) : diverses variants, les plus importantes étant ISO-8859-1 (= Latin-1) pour les langages ouest-européens, et ISO-8859-15 (= Latin-9) qui contient tous le caractères français

Windows-1252 (arrivée avec MS Windows) : remplit quelques positions inutilisés de Latin-1

Unicode/ISO 10646

(à partir de 1991) effort pour définir un codage *universelle*, c'est à dire pour pouvoir représenter tous les langages du monde.

Unicode permet 1 million de caractères différents (dont 144,000 utilisés).

On affecte un point de code hexadécimal à chaque caractère :
p.ex. A = U+0041, é = U+00E9, etc

Les points de code d'Unicode sont compatible en arrière avec Latin-1.
(Mais pas avec Windows-1252.)

Usage actuel : plus de 80% des sites Internet en Unicode, 10% en Latin-1

Représentation des points de code

ASCII/Latin-1/Windows : facile, un caractère égale un octet

Unicode : plusieurs standards différents

- UTF-32 (BE/LE): 4 octets par caractère
- UTF-16 (BE/LE): 2 ou 4 octets par caractère
- UTF-7: codage vers des codets ‘imprimables’ en ASCII
- UTF-8: codage à longueur variable, le plus utilisé

Découpage UTF-8

Codes représentés par un ou plusieurs octets.

Pour les ASCII: un seul octet entre 0..127.

Pour les autres : 2 à 4 octets entre 128..255. Exemple :

> Les codes entre 128 (= 0x80) et 2047 (= 0x7FF) prennent deux octets.
>
> Le code pour “é” égale 233, en binaire sur 11 positions: 00011101001
>
> Le premier octet est composé de 110 suivi par les cinq premiers bits du code, le deuxième de 10 suivi par les six autres bits.
>
> Ça donne 11000011 10101001 = 0xC3 0xA9.

Les terminaux

Terminal : à l'origine, les *terminaux* (ou *consoles*) sont des simples machines branchées sur un ordinateur central.

Un terminal est équipe d'un clavier et d'un écran capable d'afficher du texte (mais pas du graphisme).

Fonctionnement d'un terminal

Un terminal fonctionne d'une manière très simple :

Échange des caractères avec l'ordinateur central

Affichage des caractères saisis sur clavier
+ transfert vers l'ordinateur

Affichage des données venues de l'ordinateur

Distinction entre caractères dits imprimables ou de contrôle (retour chariot CR, saut de ligne LF, tabulation TAB, appel BEL, retour arrière BS, . . .)

Émulation de terminal

Les applications dites ’terminal’ (ou similaire) dans un Unix/MacOS d’aujourd’hui simulent des terminaux historiques.

Démonstration (keydump.c)

Mode echo : choisit si le terminal affiche les caractères venus du clavier ou pas

Mode canonique : choisit si les touches sont dispos tout de suit ou seulement après une nouvelle ligne

Voir aussi : `stty`

Approfondissement : gestion des touches

Mais comment le terminal apprend-t-il que l'utilisateur a tapé un `A` ?

Auprès du clavier, toute touche possède un numéro. Disons que la touche `A` possède le code 38.

Lorsqu'on tape sur une touche, le clavier signale une interruption auprès du processeur central. Le processeur va alors interrompre son travail actuel et sauter vers un soi-distant *gestionnaire d'interruption*.

Le système d'exploitation fait pointer ce gestionnaire vers son propre code qui reçoit alors le code 38.

Le système fait appel à un pilote qui traduit le code de touche vers un symbole. Dans notre cas, un pilote bien configuré traduit le code 38 en un symbole `a` ou `A` etc en fonction des *modificateurs* (Shift, Ctrl, Alt) en vigueur.

Cas spécial de Linux/X11:

> Le gestionnaire génère un évènement auprès de l'application de premier plan qui contient le nom de touche et le symbole (voir `xev`, `xmodmap`).
>
> Si cette application est un terminal, il traduit le symbole en fonction de l'encodage en vigueur et met le résultat à la disposition d'un processus.

Terminal et encodage

Demonstration : Utiliser `keydump` avec encodage UTF-8 et ISO-8859-1 (Latin1) et avec caractères accentués.

L'encodage en vigueur guide l'interprétation (par le terminal) des données qui lui sont envoyées.

Il est essentiel que le terminal et ses applications sont d'accord sur l'encodage !

Séquences de contrôle dans le terminal

Certains caractères non-imprimables ont une interprétation spéciale dans le terminal :

des contrôles simples (nouvelle ligne etc)

des séquences commençant par ‘escape’ (27), p.ex.:
ESC[31m : afficher texte en rouge
ESC[42m : arrière-plan en vert
ESC[0m : arrière-plan en noir
ESC[2J : effacer l’écran
ESC[10;20H : mettre le cursor sur position (10,20)

Le caractère ESC correspond à `\e` en C. Dans le shell, `echo -e "\e..."`

En C, la bibliothèque `ncurses` permet de gérer ces séquences.

Parenthèse : nouvelle ligne

Pour des raisons historiques, il existe deux standards pour représenter une nouvelle ligne :

format Unix : un seul octet (10 décimal) entre deux lignes

format DOS: nouvelle ligne représentée par CR+LF = octets 13+10

La plupart des éditeurs de texte sait traiter les deux formats.

Conversion explicite : `dos2unix`, `unix2dos`

Traitement de text sous Unix

La plupart des outils (`grep` etc) ignorent les codages ;
ils traitent des séquences d'octets.

Des éditeurs de texte (`vi`, `emacs`) ou navigateurs (firefox) encodent selon les options en vigueur (ou selon leur propre 'intelligence'.

`file` essaie à déterminer le codage des fichiers texte ;
(sinon, `hexdump -C` s'avère utile. . .)

`iconv` sait convertir entre une grande variété d'encodages.

Faire connaître le codage d'un document

En HTML ou XML, dans les entêtes :

HTML : `<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"/>`

depuis HTML5 : `<meta charset="ISO-8859-1">`

XML: `<?xml version="1.0" encoding="UTF-8"?>`

Pour des fichiers texte : pas de standard, il faut spécifier autrement !

Unicode : formes composées et décomposées

Problème spécifique dans Unicode : un même caractère peut être codé sous plusieurs formes !

P.ex., é = U+00E9 (forme composée) ou U+0065 U+0301 (forme décomposée)

En fait, la partie U+03xx de Unicode contient des diacritiques qui se combinent avec le caractère précédent.

Une application conforme aux standards Unicode est censée traiter ces séquences identiquement !

Les équivalences et formes normales

Unicode connaît deux formes d'équivalence entre (séquences de) caractères :

Canonical equivalence (équivalence forte) – deux chaînes fortement équivalent sont à traiter identiquement dans tous les contextes

Compatibility equivalence (équivalence faible) – équivalence entre caractères qui ont la même apparence mais une semantique différente selon le contexte (p.ex. 2 normal et ^{2} superscript)

Formes normales canoniques : NFC/NFD (composé/décomposé)

Formes normales de compatibilité : NFKC/NFKD (composé/décomposé)

L'utilitaire `uconv` sait convertir un texte aux formes normales.

Forme de codage

Les codets sont typiquement sous la forme d'un valeur 0..255.

Normalement, on représente un codet dans sa forme binaire dans un octet.

Mais parfois, on souhaite une représentation "imprimable" :

> P.ex., les protocoles Internet (mail, web) ne permettent que des caractères imprimables dans les entêtes, dans les adresses, . . .
>
> Les formes imprimables résistent mieux aux erreurs car tout codage habituel est compatible avec ASCII.

Exemples de forme de codage non-trivial

Codage ‘pourcent’ dans les adresses web, p.ex. :

`https://fr.wikipedia.org/wiki/Caract%C3%A8re`

Codage ‘quoted-printable’, p.ex. dans les entêtes mail :

`Subject:  =?utf-8?Q?T=C3=A9st?=`

(donne ‘Tést’ comme sujet du mail)

Codage ‘base64’, pour le même contenu :

`Subject:  =?utf-8?B?VMOpc3Q=?=`

Dans base64, on se donne 64 symboles (A-Za-z0-9+/) qui code des séquences de 6 bits ; quatre symboles font trois codets.