Types primitifs de données

- Entiers
- Flottants
- Caractères

Représentation des nombres entiers

Principe

1. coder les valeurs à représenter sous forme binaire
2. stocker ces valeurs sous forme d'une séquence de bits

   dans une ou plusieurs cellules de la mémoire.

Ecriture en base 2

Chaque chiffre du nombre multiplie une puissance de 2

110101

représente le nombre

$\mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0$

$= \mathbf{1} \times 32 + \mathbf{1} \times 16 + \mathbf{0} \times 8 + \mathbf{1} \times 4 + \mathbf{0} \times 2 + \mathbf{1} \times 1$

= 53 en décimal

Remarques

$\mathbf{2^n}$ = **100……00** (1 suivi de **n zéros)**

$\mathbf{2^n - 1}$ = **11……11** (n fois le chiffre 1)

Transformer du décimal en binaire :

- diviser le nombre par 2
- écrire le reste de la division à gauche du chiffre précédent
- recommencer jusqu'à ce que le nombre soit nul
-

43 = 2 x 21 + 1	1
21 = 2 x 10 + 1	11
10 = 2 x 5 + 0	011
5 = 2 x 2 + 1	1011
2 = 2 x 1 + 0	01011
1 = 2 x 0 + 1	**101011**

Mais pourquoi utiliser le binaire ?

Stockage et transmission : il suffit d'un mécanisme physique à deux états, beaucoup moins cher qu'un mécanisme à 10 états.

Calcul : les tables sont simples => circuits électroniques simples

Table d'**addition** binaire :

```
   0      0      1      1
+  0   +  1   +  0   +  1
------ ------ ------ ------
   0      1      1    1 0
```

Table de **multiplication** binaire :

```
   0      0      1      1
×  0   ×  1   ×  0   ×  1
------ ------ ------ ------
   0      0      0      1
```


Nombres entiers de 0 à 2^n–1

Une cellule de **n bits** peut représenter les nombres de

de **00……00** à **11……11** (n chiffres)

c'est à dire

de **0** à **2^n – 1**

P.ex. dans un cellule de **8 bits** on peut représenter les entiers positifs

de **0** = [00000000] à **255** = [11111111]

Pour stocker un nombre plus grand ou égal à 2^n

on utilise un nombre suffisant de cellules adjacentes.

Exemple: nombres de 0 à 2'000'000'000 => 4 cellules de 8 bits.

Entiers relatifs (nombres négatifs)

Technique du complément à 2.

Pour représenter **–X** avec **n bits** on utilisera le nombre $\mathbf{2^n - X}$.

Exemple (n = 8), pour représenter **–6** on fait

	(b8)	b7	b6	b5	b4	b3	b2	b1	b0	
	1	0	0	0	0	0	0	0	0	**$= 2^8$**
–		0	0	0	0	0	1	1	0	**= 6**
=	0	1	1	1	1	1	0	1	0	**= – 6**

Avec **n bits** on représente les entiers relatifs

de $\mathbf{-2^{n-1}}$ à $\mathbf{2^{n-1} - 1}$.

Propriété du complément à 2

- Il n'y a qu'un zéro
- X + –X = 0

	(b8)	b7	b6	b5	b4	b3	b2	b1	b0	
		0	0	0	1	0	1	1	0	**= 22**
+		1	1	1	0	1	0	1	0	**= –22**
=	1	0	0	0	0	0	0	0	0	**= 0**

Trouver la représentations de –X en binaire

- calculer Y = X – 1
- calculer la représentation binaire de Y
- remplacer les 0 par des 1 et les 1 par des 0

Opérations sur les entiers à n bits

Représentation des entiers limitée

=> la définition standard des opérations +, –, × ne fonctionne pas

Exemple

si n = 7

les entiers vont de $-128 = -2^7$ à $2^7 - 1 = 127$,

que vaut 127 + 3 ?

que vaut –100 – 55

Arithmétique modulaire

Principe : prendre le reste de la division par 2^n après chaque opération.

== on ne considère que les n premiers bits du résultat.

Exemple

calcul de 127 + 3 :

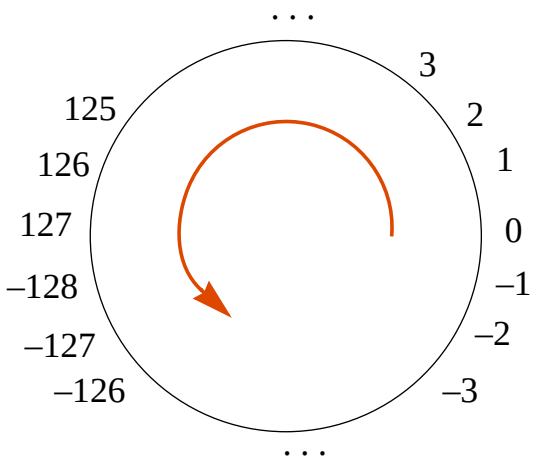
	(b8)	b7	b6	b5	b4	b3	b2	b1	b0
		0	1	1	1	1	1	1	1
+		0	0	0	0	0	0	1	1
=	1	1	0	0	0	0	0	1	0

donc

$$127 + 3 = -126.$$

Cercle des nombres

Tout se passe comme si les nombres étaient arrangés sur un cercle :



127 + 1 = –128

Multiplication

La multiplication est également modulaire

Sur les entiers à n bits

$a \times b = ab$ modulo 2^n

Exemples sur 4 bits

-2×-2

$= 1110 \times 1110 = 11000100$

= 0100 (on garde les 4 derniers bits) = 4

$3 \times -1 = 0011 \times 1111 = 101101 = 1101 \pmod{2^4} = -3$

Multiplication par une puissance de 2

2^k = 100……00 (k zéros)

donc $2^k \times m$ = m00……00 (ajouter k zéros à gauche → facile à calculer)

Division et reste

Division entière

p / q = le plus grand entier r tel que

$$rq \leq p < (r+1)q$$

p mod q = l'entier m tel que

$$(p / q) \times q + m = p$$

La division par 0 n'est pas définie.

Division par une puissance de 2

Diviser par 2 revient à enlever le chiffre le plus à gauche.

Diviser par 2^k : enlever les k chiffres les plus à gauche

Les modèles courants de nombres entiers proposés par les processeurs

Ces modèles utilisent en général un multiple de 8 bits, on trouve:

le byte: entier de 8 bits, de -128 à +127

le mot: entier de 16 bits, de -32768 à +32767

le double mot: entier de 32 bits, de -2^{31} à $2^{31}-1$

le quadruple mot: entier de 64 bits, de -2^{63} à $2^{63}-1$

Conversions

dans un sens pas de problème (petit -> grand)

dans l'autre sens: s'assurer que le nombre à convertir est suffisamment petit

Les modèles proposés par les langages de programmation

1. Modèle d'entiers = celui du **processeur** utilisé: **C, Pascal, C++**, etc.

   => le même programme a des comportement différents suivant les machines !!!

2. Un langage comme **Ada** permet de **définir des types d'entiers**

   p.ex. **type** MesNombres **is** –2 .. +3000

3. Le langage **Java** propose des types standards **indépendants du processeur**:

- int : entier de 32 bits
- long : entier de 64 bits
- byte : entier de 8 bits
- short : entier de 16 bits

   => même comportement numérique sur toutes les machines

   ! conversions = troncations (= modulo) => effets inattendus

Nombres entiers illimités

Technique : attribuer à chaque nombre une quantité de mémoire suffisante
sous forme de cellules contigües.
=> pas de soucis de débordement (jusqu'à la taille de la mémoire)

Prix à payer: performance
les processeurs ne sont pas prévus pour traiter ce type de représentation.
=> écrire des procédures spécifiques
qui utilisent l'arithmétique modulaire du processeur
traiter les nombres par morceaux de n bits.

Moèle d'entiers des langages Smalltalk, Maple, Mathematica
Procédures disponibles en Java, C++, etc.

Les nombres flottants

Représentation des réels

Erreurs de représentation

Erreurs de calcul

Analyse numérique

Représentation des réels

Représentations exacte de l'ensemble des réels

Impossible car

- l'ensemble des réels est infini
- non dénombrable (on ne peut numéroter les réels)

=> aucun codage ne peut représenter un intervalle des réels (aussi petit soit-il)

Représentation d'un nombre réel = approximation par un nombre proche

Exemple: calculette à dix chiffres

1/3 représenté par 0.3333333333, erreur < 0.00000000003333…

π représenté par 3.141592653, erreur < 0.0000000005898

=> un nombre "réel" de la machine représente plusieurs vrais nombres réels

Représentation de taille fixe

Séquences de bits de taille finie et fixée.

Le format classique des nombres flottants est composé de

mantisse (nombre entre 1.0 et 1.5) ; **exposant** (de 2); **signe**.

le nombre représenté est

$$\text{signe} \times \text{mantisse} \times 2^{\text{exposant}}$$

Standard IEEE-754 (32 bits):

bit no.	31	30	...	23	22	...	0
	signe	exposant+127			mantisse		

[s e_7 e_6 … e_0 m_{23} m_{22} … m_1 m_0]

représente le nombre

$$\text{sgn}(s) \times 1, m_{23}m_{22}\ldots m_1 m_0 \times 2^{e_7 e_6 \ldots e_0 - 127}$$

Exemple

[1 10000000 01001000000000000000000]

représente

$$- 1.01001 \times 2^{10000000-127}$$

$$= - (1 + 1/2^2 + 1/2^5) \times 2^{128-127}$$

$$= - (2 + 1/2 + 1/2^4)$$

$$= 2.5625$$

Chiffres binaires après le point

multiplient les puissance négatives de 2 (ou puissances de 1/2)

Représentation de 1/3

1/3 = 1/4 + 1/16 + 1/64 + 1/256 + …

$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$

= 0.01010101010101010101... en binaire

$= 1.01010101010101010... \times 2^{-2}$

donc:

signe: 0

exposant: 127–2 = 125 = 01111101 en binaire

mantisse: 01010101010101010101010

Représentation IEEE sur 32 bits:

[0 01111101 01010101010101010101010]

Le format double précision (64 bits)

bit no.	63	62	...	52	51	...	0
	signe	exposant+1023			mantisse		

Le **plus petit nombre strictement positif** que l'on peut représenter est

en simple précision: 1.4×10^{-45}

en double précision: 4.9×10^{-324}

Le **plus grand nombre positif** que l'on peut représenter est

en simple précision: 3.4×10^{38}

en double précision: 1.8×10^{308}

Des configuration de bits réservées à cet effet permettent de représenter

l'infini positif, l'infini négatif (résultats de divisions par 0, p.ex.), ainsi que l'indéterminé (0/0).

Précision de la représentation

La précision de la représentation =

l'erreur que l'on commt en représentant un nombre.

Par exemple,

la représentation de 1/3

[0 01111101 01010101010101010101010]

$2^{-2} \times 1.01010101010101010101010$

néglige les chiffres après la 23^{e} position.

erreur = 0.0....(23 fois 0) ...01010101010 $\times 2^{-2} \cong 1.5 \times 10^{-8}$.

Erreur relative de la représentation

Plus grande erreur arrive lorsque les chiffres négligés sont tous des 1.

$$\text{erreur} = 0.0...(23 \text{ fois})...011111111... \times 2^{\text{exp}-127}$$

$$= 2^{-23} \times 2^{\text{exp}-127}.$$

Erreur relative

Le rapport entre l'erreur et le nombre représenté :

$$(2^{-23} \times 2^{\text{exp}-127}) \,/\, (1.m_{23}\, m_{22} \ldots m_1\, m_0 \times 2^{\text{exp}-127})$$

$$\leq 2^{-23} \cong 10^{-7}$$

=> erreur relative maximum inférieure à 10^{-7};

les 7 premiers chiffres sont représentés correctement.

Double précision => l'erreur relative est inférieure à $2^{-52} \cong 10^{-16}$.

Signification

7 chiffres de précision = erreur **relative** de 10^{-7}

1000333.444555 $\rightarrow$ 1000333.~~444555~~

0.00000000000012 $\rightarrow$ 0.00000000000012

car erreur < 0.00000000000012 $\times 10^{-7}$ = 0.000000000000000000012

0.0000000000012345678765 $\rightarrow$ 0.000000000001234567~~8765~~

49384877236362281 $\rightarrow$ 4938487~~7236362281~~ = 49384870000000000

.

Précision des calculs et analyse numérique

La représentation des nombres n'étant pas exacte

=> le résultat d'un calcul ne correspond pas forcément à la valeur exacte.

En théorie on a

$$a + b = c$$

sur la machine on a

$$a + b = c\ (1 + \varepsilon_{ab})$$

ε_{ab} = l'erreur d'arrondi, dépend de a et de b.

Conséquence sur une machine

$$(a + b) + c \neq a + (b + c) \text{ en général.}$$

Par exemple. On veut calculer

$$s = \sum_{n=1}^{100000} \frac{1}{n^2}$$

méthode 1 : s1 = (…((((1+1/4)+1/9)+1/16)+1/25)+…)+1/10000000000

méthode 2 : s2 = 1+(1/4+(1/9+(1/16+(1/25+(…+1/10000000000)…))))

Résultats

un calcul théorique montre que

$| s - s1 | \leq 1.6\ 10^5 \varepsilon$

$| s - s2 | \leq 9.2\varepsilon$

où e est la précision de représentation des nombres (p.ex. 10^{-7} pour les float).

Si l'on effectue le calcul avec un programme Java en utilisant des nombres de type float on obtient

s1 = 1.64473

s2 = 1.64493

alors que le s exact est 1.64492407…

Propagation des erreurs

La propagation des erreurs peut devenir énorme dans certains cas. Prenons par exemple le calcul de la fonction exponentielle avec la formule de Taylor:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!} + \ldots$$

On utilise l'algorithme suivant pour calculer une approximation de ex en prenant les 20 premier termes de la série de Taylor:

t ← 1; e ← 1;

pour n de 1 à 20 {

t ← t * x / n

e ← e + t

}

Résultats

En programmant cet algorithme en Java avec des nombres de type float on obtient les résultats suivants:

x	e^x calculé	e^x exact
-1	0.367879	0.367879…
-5	0.00670682	0.0067379…
-10	–27.7064	0.0000454…
-20	–21866000	0.0000000020612

La branche des mathématique qui s'intéresse aux algorithmes numériques et aux problèmes de précision de calculs s'appelle l'*analyse numérique*.

Les travaux en analyse numétique on mis en évidence deux notions fondamentales:

conditionnement des problèmes et stabilité de algorithmes

Un Exemple

On souhaite calculer sur une machine à 8 chiffres de précision

$$x = a + \sqrt{a^2 - b}$$

pour a = 1000 et b = 999999.987654321.

```
    a²   1000000.00
-    b    999999.99
=             0.01
```

=> x = 1000 + $0.01^{1/2}$ = 1000.1

la valeur exacte est 1000.111111… .

Il ne reste donc que 5 chiffres corrects.

Pourquoi ?

Problèmes mal conditionnés

Un problème consistant à calculer y = F(x)

est mal conditionné

si une petite variation de la valeur de x

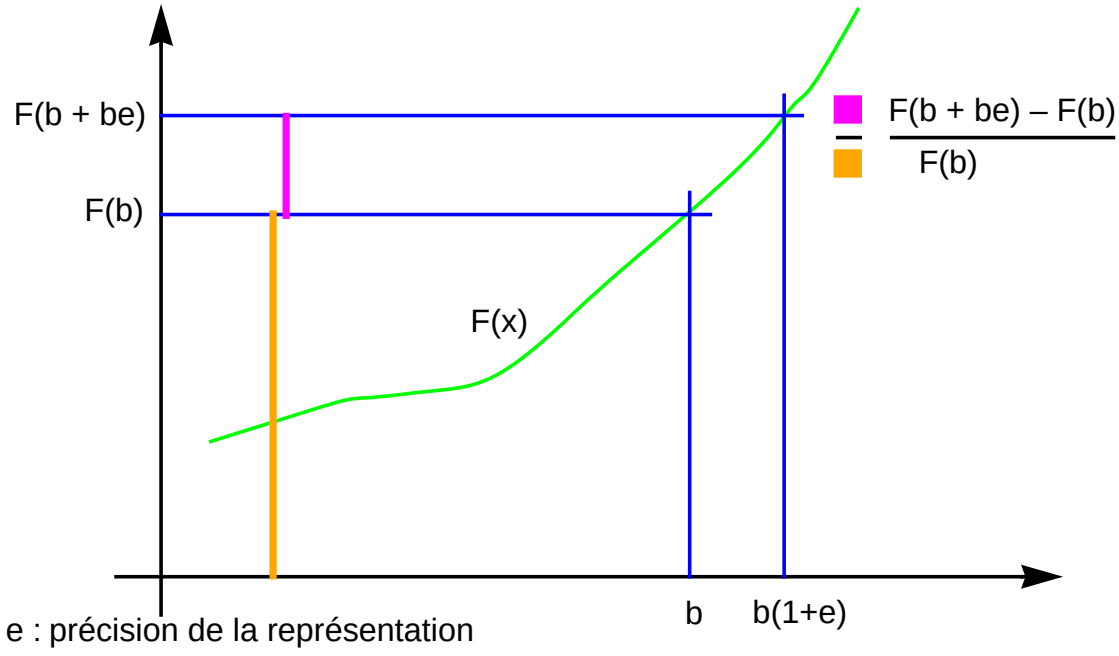
entraine une grande variation de y.

Le nombre de condition C d'un problème y = F(x), pour une valeur b du paramètre x est défini comme

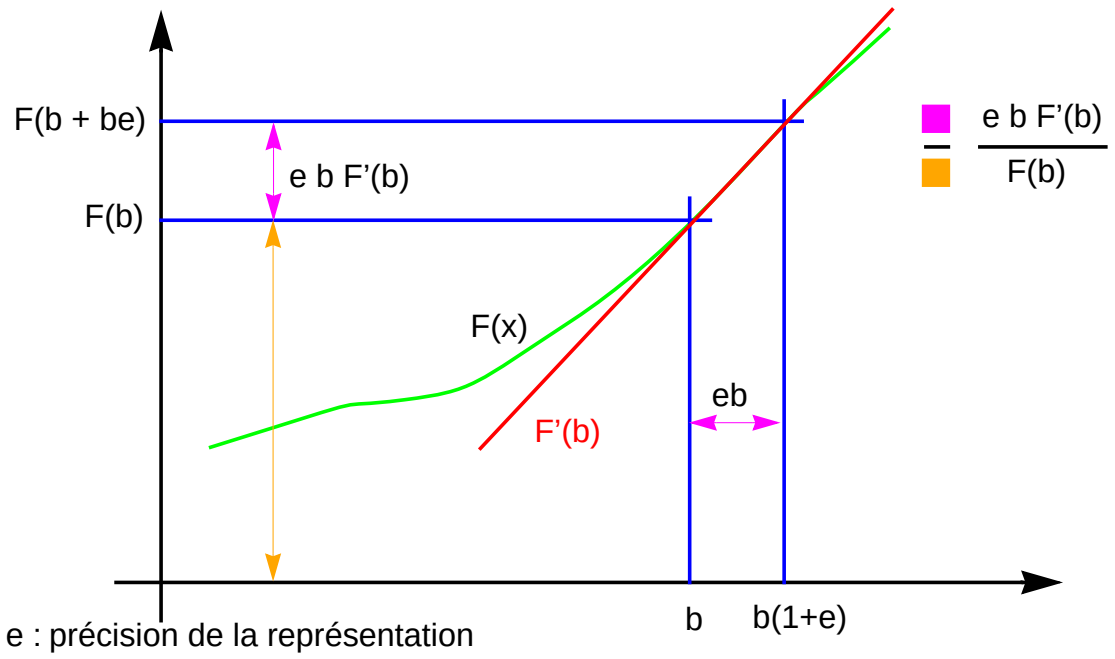
$$C = \left| \frac{bF'(b)}{F(b)} \right|$$

Si C est grand le problème est dit "mal conditionné".

Graphiquement



Graphiquement (+)



Exemples

addition : F(x) = a + x (a et x positifs)

F'(x) = 1

C = x / (a + x) : toujours < 1, bien conditionné

soustraction : F(x) = a – x (a et x positifs)

F'(x) = –1

C = x (–1) / (a – x) : très grand si x – a petit

ne jamais soustraire des nombres proches l'un de l'autre

multiplication : F(x) = ax (a et x positifs)

F'(x) = a

C = xa / ax = 1 : bien conditionné

inversion (division) : F(x) = 1/x, F'(x) = $-1/x^2$,

C = x $(-1/x^2)$ / (1/x) = (–1/x) / (1/x) = –1 : bien conditionné

Stabilité numérique

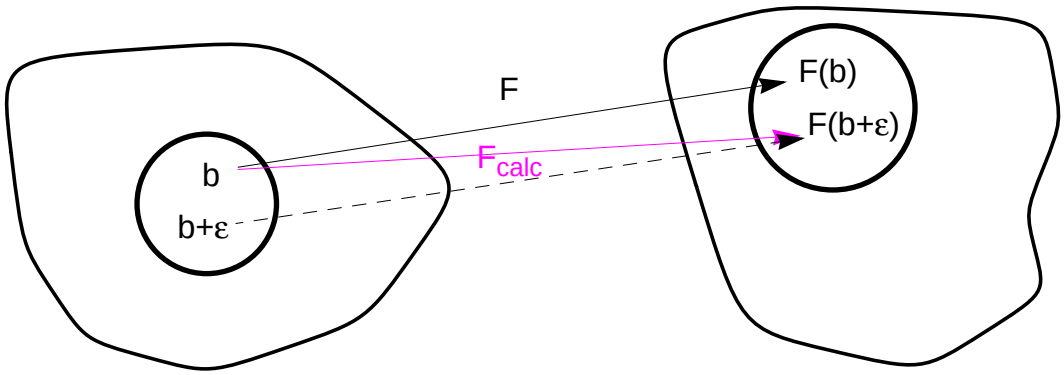
Problème bien conditionné **nécessaire** mais **pas suffisant**

pour qu'un algorithme donne une réponse précise.

Algorithme numériquement stable

si la valeur $F_{calc}(b)$ calculée par l'algorithme est la solution d'un problème proche.

$$F_{calc}(b) = F(b + \varepsilon) \text{ avec } \varepsilon \text{ petit.}$$



Algorithme stable pour e^x

exp(x) est bien conditionné:

$$x \exp'(x) / \exp(x) = x \exp(x) / \exp(x) = x$$

L'algorithme de calcul précédent n'est pas stable

pour b = –10 : $F_{calc}(b) = -27.7064$,

il n'existe pas d'ε, même assez grand, tel que

$e^{-10+\varepsilon} = -27.7064$.

Il est stable pour la valeurs de b comprises entre –1 et +1.

On en déduit un nouvel algorithme:

Soit y la partie entière de x et z = x–y

donc $e^x = e^{y+z} = e^y e^z$

1. calculer e^y par multiplications (et inversion à la fin si y < 0)
2. calculer e^z par l'algorithme précédent
3. multiplier les deux résultats

Résumé

Arithmétique des machines =/= arithmétique des nombres mathématiques.

Il n'y a pas de correspondance bi-univoque entre l'arithmétique en virgule flottante des machines et celle du corps **R** des réels.

Les calculs en entiers peuvent "déborder" (arithmétique modulaire)

Les calculs en flottants sont faux (un peu, beaucoup, énormément)

=> tout algorithme qui fait des opérations sur les flottants

est susceptible de produire n'importe quoi

Conversions de type

En mathématique : **Naturels** $\subseteq$ **Relatifs** $\subseteq$ **Rationnels** $\subseteq$ **Réels**

Pas en informatique :

1) il y a différents types d'entiers et de flottants

2) il "manque" des nombres correspondances.

l'entier 123 456 789 012 345

ne peut se représenter exactement avec un flootant 32 bits

le mieux qu'on peut faire sera une approximation : 123456800000000.

=> il faut des opérations de conversion

Elargissement

élargissement = passage à un type "plus grand"

toute valeur de T1 peut se représenter dans T2

exemples:

entier 8 bits $\rightarrow$ entier 16 bits, etc.

entier 8 ou 16 bits $\rightarrow$ flottant 32 bits (car mantisse de 24 bits)

entier 8 ou 16 ou 32 bits $\rightarrow$ flottant 64 bits (car mantisse de 54 bits)

flottant 32 bits $\rightarrow$ flottant 64 bits

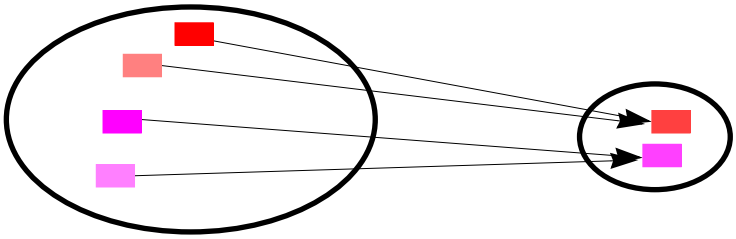
La conversion ne pose pas de problèmes.

Elle est en général implicite dans les langages de programmation

Java: **double x = 23; // conversion int (32 bits) à double (fottant 64 bits)**

Rétrécissement

Passage d'un type plus grand à un plus petit, pas injectif.



Différentes approches

Entier(m) → Entier(n) (n < m): garder le n bits inférieurs (= modulo 2^n)

si -2^{n-1} <= x < 2^{n-1} : conversion exacte, sinon "mauvaise surprise"

Flottant → Entier : arrondissement ou troncation puis comme pour les entiers

=> aussi de mauvaises surprises.

Exemple: Java

Tous les rétrécissement doivent être indiqués explicitement

car potentiellement dangereux.

```
      int i = 65537;    //                  i = [00000000000000010000000000000001]
**    short s1 = i      // erreur de syntaxe, le compilateur refuse cette instruction
      short s2 = (short)i // syntaxe OK
                              // résultat :      s2 =          [0000000000000001] = 1
**    float f = 23      // erreur de syntaxe int -> float est un rétrécissement
      float f = (float)1666777888
                        // conversion, f = 1666778000, 7 chiffres de précision
      double g = 45     // OK int -> double est un élargissement
```

Attention ! les règles changent suivant les langages (C, Pascal, Ada, Excel, SQL, …)

Un exemple de conversion ratée

Tout le logiciel de la fusée **Ariane 5** est programmé en Ada, langage très sûr.

Et pourtant …

l'échec du premier tir d'Ariane 5 a été causé par

une erreur du système informatique de guidage.

Cette erreur est survenue lors d'une conversion de type

qui a causé un dépassement de capacité d'une variable.

Parmi les recommandations émises suite à cet accident on notera :

Identifier toutes les hypothèses implicites faites par le code et ses documents de justification sur les paramètres fournis par l'équipement. Vérifier ces hypothèses au regard des restrictions d'utilisation de l'équipement.

Vérifier la plage des valeurs prises dans les logiciels par l'une quelconque des variables internes ou de communication.

Caractères

Représentation

Evolution des codages

Unicode

Représentation

La représentation des caractèes est une convention

- associe à chaque caractère de l'alphabet un nombre binaire arbitraire

Il n'y a pas de représentation canonique

- il existe un ordre sur certains caractères 'a' < 'b' < 'c'
- mais pas sur tous ';' < '+' ?

Le codage devrait respecter l'ordre alphabétique quand il existe

- code('a') < code('b') < code('c')

pour simplifier les traitements

Opérations sur les caractères :

- comparaison (=), ordre lexicographique (<), transcodage

Evoluation du codage

Nécessité de définir un standard pour l'échange de données entre machines

Codage défini par

- nb. bits pour représenter un caractère
- alphabet pris en compte
- codes assignés aux caractères (table de codage)

Premier (?) code : Telex

système électromécanique, stockage sur bande papier perforée.

5 trous sur la largeur de la bande => 5 bits par caractère => 32 caractères

=> 26 lettres majuscules + ponctuation

Codages

6 bits

Permet de représenter les 26 lettres majuscules latines, les chiffres 0..9, les symboles de ponctuation: , : ; . ( ) etc.

Suffisant pour écrire des programmes et imprimer des résultats.

7 bits

128 caractères

Standard ASCII (American standard for communication and information interchange)

adapté à la langue américaine :

- lettres majuscules et minuscules sans accents,
- chiffres, symboles de ponctuation.

Codages suite

8 bits

256 caractères possibles

3 "standards"

IBM-PC : majuscules, minuscules, lettres accentuées occidentales

Apple Macintosh : idem mais autre codage

ISO-xxx

Ensemble de standards

adaptés à différents groupes linguistiques

(p.ex. ISO-Latin-1 => langues latines)

Utilisé sur les Unix modernes, et sous Windows

Standard Unicode

- Capacité d'encoder tous les caractères du monde
- Codage sur 16 bits
- Mécanisme d'extension UTF-16 (1 mio car.)

Caractères traités

- Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, modern Korean Hangul, Chinese/Japanese/Korean (CJK) ideographs.
- Bientôt: Ethiopic, Canadian Syllabics, Cherokee, additional rare ideographs, Sinhala, Syriac, Burmese, Khmer, and Braille.
- Ponctuation, diacritiques, math., technique, flèches, dingbats
- Signes diacritiques de modification de prononciation (n + '~' = ñ)
- 18,000 codes en réserve

Elements de texte

ce n’est pas un caractère

En espagnol historique “ll” compte comme un seul élément => influence sur le tri des mots.

Unicode définit des éléments de code (caractères)

“ll” --> deux codes: ‘l’ + ‘l’

chaque lettre maj. et min. est un élément de code

Caractères et Glyphs

Différence fondamentale :

valeur d’un code ≠ son affichage sur l’écran

Notion de caractère abstrait

"LATIN CHARACTER CAPITAL A"

"BENGALI DIGIT 5"

Glyph = la marque faite sur un écran ou sur papier

représentation visuelle d’un caractère

Unicode ne definit pas les glyphs

ne spécifie pas la taille, forme, orientation des caractères sur l’écran

a a *a* ***a*** a

Caractères composites (â)

Formé de

- une lettre de base (occupe un espace) "a"
- un ou plus marques (rendus sur le même espace) "^"

Unicode spécifie

- l'ordre des car. pour créer un composite
- la résolution des ambigüités
- la décomposition des caractères précomposés

"ü" peut être encodé par U+00FC (un seul car. 16-bits)

ou bien décomposé en U+0075 + U+0308 ("u"+"¨").

=> compatibilité avec ISO-Latin-1 etc.

Définition des codes

Inclusion de standards précédents (0 .. 256 = Latin-1).

Notion de script :

- système cohérent de caractères (related characters)
- regroupe plusieurs langues
- évite la duplification (p.ex. chinois, japonais, koréen => script CJK)

Texte = séquence de codes correspondant à l'ordre de frappe au clavier

Caractères de changement de direction

Assignation des codes

Un nombre de 16 bits est assigné à chaque élément de code du standard.

Ces nombres sont appelés les valeurs de code

U+0041 = nombre hexadécimal 0041 = décimal 65 représente le caractère "A" .

Chaque caractère reçoit un nom

U+0041 s'appelle "LATIN CAPITAL LETTER A."

U+0A1B s'appelle "GURMUKHI LETTER CHA."

(standard ISO/IEC 10646)

De blocs de codes de taille variable sont aloués aux scripts

L'ordre "alphabétique" est si possible maintenu

Organisation de l'espace des codes

Code elements are grouped logically throughout the range of code values, called the codespace. The coding starts at U+0000 with the

[standard ASCII (Latin-1)] - [Greek] - [Cyrillic] - [Hebrew] - [Arabic] - [Indic] - [other scripts] - [symbols and punctuation] - [Hiragana] - [Katakana] - [Bopomofo] - [unified Han ideographs] - [modern Hangul] - [surrogate characters] - [reserved for private use (promised)] - [compatibility characters.

Base de donnée de codes

0 Code value
1 Character Name.
2 General Category
3 Canonical Combining Classes
4 Bidirectional Category.
5 Character Decomposition
6 Decimal digit value
7 Digit value.
8 Numeric value
9 ? "mirrored"
10 Unicode 1.0 Name
11 10646 Comment field.
12 Upper case equivalent mapping.
13 Lower case equivalent mapping
14 Title case equivalent mapping

Quelques caractères

```
0041;LATIN CAPITAL LETTER A
     ;Lu;0;L;;;;;N;;;;0061;
005E;CIRCUMFLEX ACCENT;Sk;0;ON;<compat> 0020
     0302;;;;N;SPACING CIRCUMFLEX;;;;
0F19;TIBETAN ASTROLOGICAL SIGN SDONG TSHUGS;
     Mn;220;ON;;;;;N;;dong tsu;;;
112C;HANGUL CHOSEONG KAPYEOUNSSANGPIEUP;
     Lo;0;L;<compat> 1107 1107 110B;;;;N;;;;;
1EE4;LATIN CAPITAL LETTER U WITH DOT BELOW;
     Lu;0;L;0055 0323;;;;N;;;;1EE5;
FC64;ARABIC LIGATURE YEH WITH HAMZA ABOVE WITH
     REH FINAL FORM;Lo;0; R;<final> 0626 0631;
     ;;;N;;;;;
```

ftp://ftp.unicode.org/Public/2.1-Update3/UnicodeData-2.1.8.txt

Formes d’encodage

UTF-16.

- caractères 16-bits
- paires de 2 x 16-bits pour extension

UTF-8.

- codage à longueur variable

  0x00 .. 0x7F ==> 1 byte (même codes que ASCII)

  0x80 .. 0x3FF ==> 2 bytes

  0x400 .. 0xD7FF, 0xE000 .. 0xFFFF ==> 3 bytes

  0x10000 .. 0x10FFF ==> 4 bytes
- conversion sans perte vers et de UTF-16