

Langage C avancé : Séance 3

Représentation mémoire des types de base Opérations arithmétiques sur ordinateur

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Objectifs de la séance

On examinera ici

- Comment les différents types de base sont représentés en mémoire
- Les opérations arithmétiques sur les entiers, les nombres décimaux

Représentation mémoire des types simples

Nombre binaire (bit) et octet (byte)

Un bit est une donnée pouvant prendre une des valeurs 0 ou 1 (ou vrai/faux, ou haut/bas, etc.).

L'octet est l'unité mémoire la plus petite manipulée (lue, modifiée) par le système. Cette zone mémoire est capable de contenir une suite de 8 bits.

Un octet peut donc représenter $256 = 2^8$ suites différentes de 8 valeurs 0 ou 1.

La mémoire de travail d'un ordinateur est constituée d'un grand nombre d'unités mémoire de ce type.

Octet

On prendra ici l'exemple de l'octet :

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

La valeur de cet octet peut être vue comme

➤ un entier entre 0 et 255 (somme de puissances de 2), il y a deux possibilités:

- Little-endian (puissances de 2 croissantes)

$$1x2^0 + 0x2^1 + 1x2^2 + 1x2^3 + 0x2^4 + 0x2^5 + 1x2^6 + 1x2^7 = 205$$

- Big-endian (puissances de 2 décroissantes)

$$1x2^7 + 0x2^6 + 1x2^5 + 1x2^4 + 0x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = 179$$

Octet (2)

Toujours avec l'exemple

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

➤ un entier entre -128 et +127 (somme de puissances de 2), un des bits indique si l'entier est positif ou négatif:

- Little-endian

dernier bit = 1 : entier négatif, 0 : entier positif

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6, 1:\text{signe-} = -77$$

- Big-endian

premier bit = 1 : entier négatif, 0 : entier positif

$$1:\text{signe-} + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -51$$

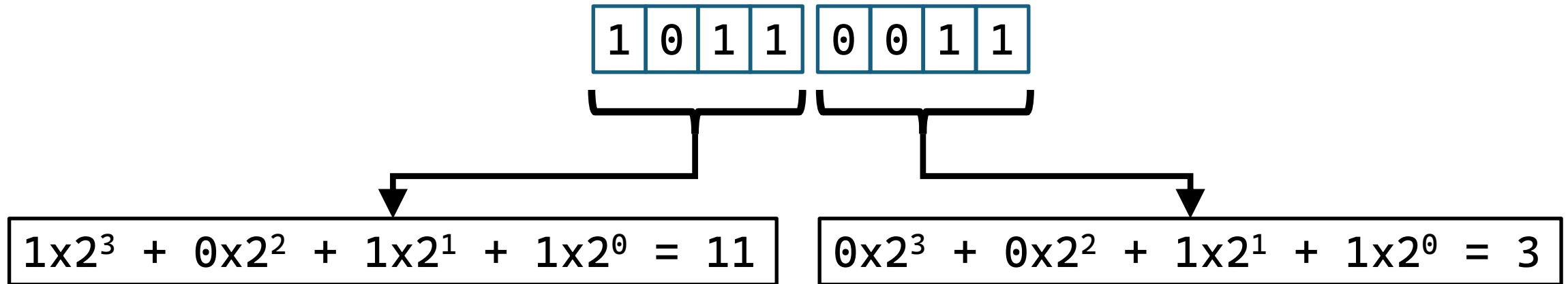
Le système le plus courant est big-endian, dans la suite on utilisera uniquement ce système.

Octet (3)

Toujours avec l'exemple

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

➤ Deux entiers entre 0 et 15 (chacun regroupe 4 bits):



La valeur de l'octet est $2^4 \times$ la valeur du premier entier + $2^0 \times$ la valeur du second entier :

$$2^4 \times 11 + 3 = 179$$

On utilise plutôt la notation hexadécimale (base 16) :

10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F

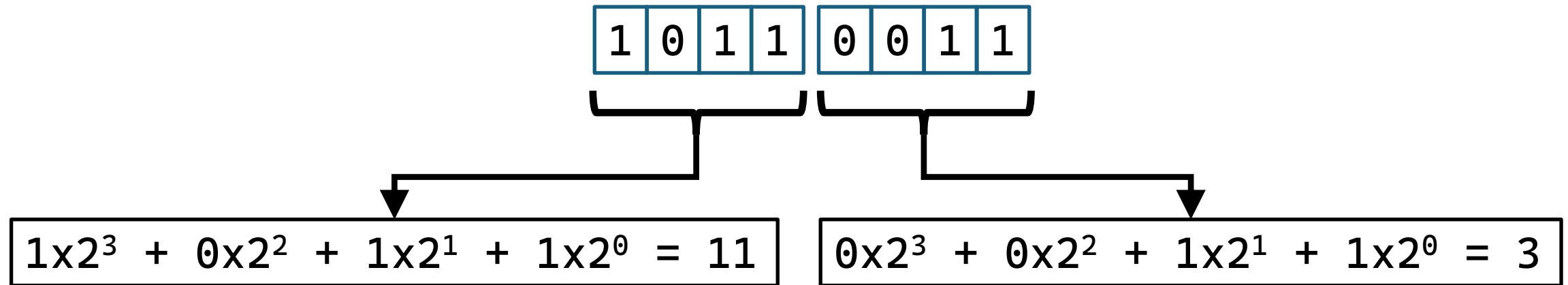
L'entier dans l'octet est noté B3.

Exemple

Toujours avec l'exemple

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

➤ Deux entiers entre 0 et 15 (chacun regroupe 4 bits):



La valeur de l'octet est $2^4 \times$ la valeur du premier entier + $2^0 \times$ la valeur du second entier :

$$2^4 \times 11 + 3 = 179$$

On utilise plutôt la notation hexadécimale (base 16) :

10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F

L'entier dans l'octet est noté B3.

Type C pour définir un octet

Il n'existe pas de type C spécifique pour définir un octet, les possibilités sont:

- Utiliser les types « `char` » ou « `unsigned char` ».

En général, les valeurs de ce type occupent un zone mémoire de 1 octet (mais il y a des exceptions, par exemple les DSP – processeurs de signaux digitaux: son, etc.)

- Utiliser les types « `int8_t` » ou « `uint8_t` ».

Ces types ne sont pas toujours définis (il faut un compilateur C assez récent, qui suit la norme C99).

Il faut inclure `stdint.h`

Exemple d'affichages d'un octet

Dans le sous-répertoire [exemples/1_octet](#), compiler et exécuter le code qui affiche un octet de type caractère, ses représentations hexadécimale et binaire.

Opérations sur un octet

Un octet peut être vu comme un entier (entre 0 et 255, ou entre -128 et 127).

On peut donc effectuer des opération arithmétiques (+, -, *, /) ou de comparaison sur un octet

Attention, le résultat doit se trouver dans l'intervalle 0 ... 255 (cas sans signe) ou -128 ... 127 (cas avec signe), sinon le résultat sera différent de celui attendu.

Opérations sur un octet (2)

Voir le code dans le répertoire [exemples/3_opOctet](#).

Exemple 1 (avec `uint8_t`):

```
uint8_t n = 70;  
uint8_t m = 90;  
uint8_t r = n+m;  
printf("n = %u m = %u n+m = %u\n", n, m, r);
```

Le résultat est bien celui attendu (160)

Exemple 2 (avec `int8_t`):

```
int8_t n = 70;  
int8_t m = 90;  
int8_t r = n+m;  
printf("n = %d m = %d n+m = %d\n", n, m, r);
```

Le résultat n'est pas celui attendu (-96)

Opérations sur un octet (3)

Exemple 3:

```
uint8_t n = 255u;  
uint8_t m = 1u;  
uint8_t p = 2u*m;  
uint8_t r1 = n + m;  
uint8_t r2 = n + p;  
  
printf("n = %u, m = %u, 2*m = %u\n", n, m, p);  
printf("n+m = %u, n+2*m = %u\n", r1, r2);
```

On n'obtient pas les mêmes résultats qu'avec des entiers mathématiques.

Remarque: C utilise «u» pour spécifier des valeurs entières positives et %u pour les afficher.

Operations sur un octet (4)

Exemple de dépassement de valeur limite

Variable n

```
uint8_t n = 255u;
```

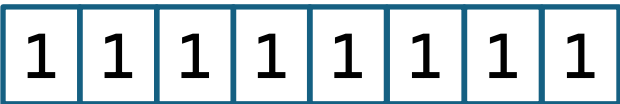
Variable m

```
uint8_t m = 1u;
```

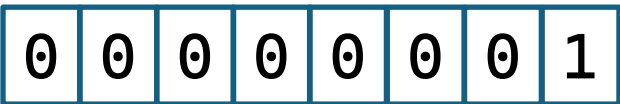
Résultat : 0

Zone mémoire

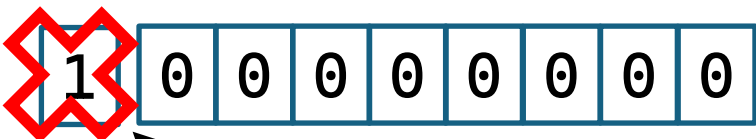
($n = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$)



+



=



Le bit supplémentaire est jeté

Représentation d'un caractère

On vient de voir qu'un caractère est enregistré en mémoire comme un octet, donc une valeur entière.

Il y a trois types caractères:
(en fait, 2 différents, char
est identique à un des autres –
dépend du compilateur)

char
signed char
unsigned char

Ces types conviennent pour les caractères dans la table de caractères de base (lettres majuscules et minuscules, chiffres, caractères de ponctuation).

Représentation d'un caractère (2)

Le type `char` est indiqué quand on utilise des chaînes de caractères (des variables contenant du texte)

L'un des autres types est utilisé quand on utilise des caractères comme des octets (bytes).

Si on veut garantir la portabilité^(*) du code, plutôt que `signed char` (resp. `unsigned char`), il vaut mieux utiliser les types `int8_t` (resp. `uint8_t`) (à condition qu'ils existent).

(*) le code fournit les mêmes résultats avec des compilateurs différents et/ou sur des machines différentes

Table des caractères de base

La table ASCII ci-contre est celle qui est utilisée pour représenter le jeu minimal de caractères (127) et appliquer les fonctions strlen, printf, ...

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Exemple de caractère non ASCII

Dans le répertoire `exemples/3_nonAscii` se trouve un code C qui utilise une chaîne contenant un caractère non-ascii.

Tables étendues de caractères

De nombreuses tables de caractères existent pour pouvoir utiliser:

- des caractères accentués (é, É, à, ...)
- des caractères d'autres alphabets : grec, japonais, arabe, ... (β, 力, ...)

Il existe une table [ASCII étendue](#) (voir page suivante), qui contient 128 caractères supplémentaires (dont les lettres accentuées en français).

Table ASCII étendue

Cette table utilise les 256 valeurs positives d'un octet:

[illegible]

Tables étendues de caractères (2)

Une des tables les plus complètes est la table **Unicode** (+ 1 million de codes disponibles, ~15% attribués à des caractères actuellement).

Voir, par exemple: <https://symb1.cc/fr/unicode-table/>

Par exemple, `code(A)` = 65 (identique code ASCII),
 `code(β)` = 03B2, (code hexadécimal)
 `code(力)` = 30AB

Codage étendu

Plusieurs codages ont été proposés pour représenter des caractères de tables étendues.

Le codage **UTF8** est utilisé dans beaucoup de pages internet.

Ce codage utilise un nombre variable d'octets pour représenter un caractère par un code entier fourni par la table Unicode.

Il faut appeler des fonctions C spécifiques, non standard, pour l'utiliser (voir, par exemple, **libunistring** <https://www.gnu.org/software/libunistring/>).

Dans ce cours, on n'approfondira pas ce point.

Types entiers

C définit de multiples types d'entiers. En fonction de l'ensemble des valeurs que l'on veut représenter, on choisira:

```
short  
int  
long  
long long
```

Chacun de ces types à une version « signed » (exemple signed long) et une version « unsigned » pour des nombres positifs (exemple unsigned int)

Ce qui donne 12 types (en fait 8 types différents), la version signed est prise par défaut.

Types d'entiers (2)

La taille et la plage de valeurs d'entiers (short, int, long long long) ne sont pas normalisées en C, chaque machine/compilateur peut choisir celles qu'il propose.

La seule certitude c'est que la taille est croissante.

Dans le répertoire `exemples/4_entiers`, différents codes affichent les tailles des types d'entier et leur intervalle de valeurs sur la machine que vous utilisez.

Nouveaux types d'entier

Si on utilise d'un compilateur assez récent (qui suit la norme C99), on dispose aussi des types entiers suivants:

avec les versions sans signe

int8_t
int16_t
int32_t
int64_t

uint8_t
uint16_t
uint32_t
uint64_t

Pour les utiliser, il faut inclure le fichier `stdint.h`

Vérifiez que ces types sont disponibles sur la machine et le compilateur que vous utilisez avec le fichier `entier2.c` (qui affiche aussi la taille et l'intervalle de valeurs de ces types)

Nouveaux types d'entier (2)

On a la garantie que ces types d'entier occupent exactement:

- 1 octet pour `uint8_t` et `int8_t`
- 2 octets pour `uint16_t` et `int16_t`
- 4 octets pour `uint32_t` et `int32_t`
- 8 octets pour `uint64_t` et `int64_t`

quelque soient la machine et le compilateur utilisé

Types d'entiers supplémentaires

- Le type entier `size_t` suffisamment long pour pouvoir contenir une adresse mémoire.
- Les types `char` et `unsigned char` qui peuvent contenir un caractère et codés dans quasiment tous les cas sur 1 octet
- Le type `bool` qui peut représenter les valeurs "vrai" et "faux"

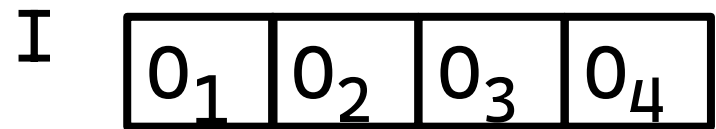
Traditionnellement, les compilateurs C considèrent que 0 signifie faux et tout autre valeur entière signifie vrai, ils utilisent un entier comme "type virtuel" `bool`

Il faut inclure le fichier `stdbool.h`.

Représentation des entiers non signés

Une variable entière non signée (positive ou nulle) de n'importe quel type entier, est représentée par un nombre fixe d'octets.

Par exemple, pour variable entière non signée, codée sur 4 octets:



La valeur de l'entier (rappel, on ne considère ici que le rangement big-endian) est

$$\begin{aligned}\text{Valeur}(I) &= \text{Valeur}(0_1) 256^3 \\ &+ \text{Valeur}(0_2) 256^2 \\ &+ \text{Valeur}(0_3) 256^1 \\ &+ \text{Valeur}(0_4) 256^0\end{aligned}$$

Représentation des entiers non signés (2)

La valeur maximale de l'entier non signé, codé sur 4 octets, est 256^4 (= 4294967296) et la valeur minimale 0

Représentation des entiers signés

Une variable entière signée (positive, négative ou nulle) de n'importe quel type entier, est représentée par un nombre fixe d'octets.

Par exemple, pour une variable entière signée, codée sur 2 octets (ou 16 bits) :

I

1	1	0	1	0	1	1	0	1	1	0	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Le premier bit indique le signe (négatif : 1, positif 0)

Les autres bits donnent la valeur de l'entier

Représentation des entiers signés (2)

La valeur maximale d'un entier signé est la moitié de la valeur maximale d'un entier non signé de même taille.

Il faut en effet attribuer une partie des contenus possibles aux valeurs négatives.

Opérations sur les entiers

Les opérations possibles sur les entiers de même type sont les opérations arithmétiques

$$c=a+b, \quad c=a-b, \quad c=a*b, \quad c=a/b$$

avec a, b, c des entiers du même type.

Ces opérations donnent un **résultat exact**, sauf:

- Si le résultat est trop grand (ou trop petit) pour pouvoir le ranger dans l'entier qui reçoit le résultat.
- Dans le cas de la division, si a n'est pas un multiple de b , le résultat sera un entier de même type proche du (mais pas égal au) résultat exact (qui n'est pas entier).

Débordement d'entier

Si a et b sont deux entiers du même type, la valeur de l'une des opérations $a+b$, $a-b$ ou $a * b$ pourrait ne pas entrer dans l'intervalle des valeurs possibles de ce type entier. On parle de **débordement (overflow)**.

Exemple: si a et b et c sont de type entier signé codé sur 2 octets (`short` ou `int16_t`),

$$a = 200, b = -300, a * b = -60000$$

n'est pas représentable avec c .

En général, le système ne prévient pas si un débordement a eu lieu

Débordement d'entier (2)

Repérer des débordements de ce type est compliqué et peut ralentir le calcul.

En général, on essaie de le faire sur des données fournies par l'utilisateur ou lues dans des fichiers, etc.

➤ Un test possible pour repérer cela est d'écrire:

```
int16_t a = 200, b = -300, c;  
  
if (a >= INT16_MAX/b || a <= INT16_MIN/b)  
    printf("erreur ...");  
else  
    c = a*b;
```

- ou passer, temporairement si c'est possible, dans un type entier plus grand:

```
int16_t a=200, b=-300, c;  
int32_t temp = ((int32_t) a) * ((int32_t) a);  
  
if (temp > (int32_t) INT16_MAX ||  
    temp < (int32_t) INT16_MIN))  
    printf("erreur ...");  
else:  
    c = (int16_t) temp;
```

Remarque : les conversions `int16_t <> int32_t` sont inutiles (effectuées sans pertes implicitement par le compilateur), sauf la dernière (certains compilateurs affichent un message)

Division d'entiers

Une division d'un entier par un autre donne comme résultat un entier, qui peut ne pas être exact.

Exemples: si x et y sont deux entiers (signés):

```
x = 10; y = 4; x/y == 2
x = -10; y = 4; x/y == -2
x = 2; y = 3; x/y == 0
x = -5; y = -2; x/y == 2
x = -6; y = -2; x/y == 3
```

L'algorithme de division entre entiers calcule l'entier le plus grand entre la division de la valeur absolue de x par celle de y (le résultat est positif)

Puis, change le signe de ce résultat si x a un signe différent de celui de y .

Division d'entiers (2)

Ce n'est pas considéré comme une erreur, puisque le résultat mathématique n'est pas entier.

Si on veut avoir le résultat exact (en tout cas plus précis), il faut convertir les deux entiers (au moins un des entiers) en un des formats réels fournis par C. Dans ce cas, le résultat est lui aussi réel.

```
int x = ... , y = ... ;  
double r;  
  
r = ((double) x)/((double) y);  
r = x/((double) y);  
r = (1.0*x)/y;
```

Ces trois versions donnent un résultat identique (on force la conversion au moins partielle int -> double)

Types réels

C fournit 3 types de nombres réels (avec partie entière et décimale) : `float`, `double`, `long double`. Tous ces types sont signés (il n'existe pas de type réel positif).

Les capacités de ces types varient (ce n'est pas normalisé), mais la règle est:

- `taille(long double)`
 - > `taille(double)` > `taille(float)`
- `précision(long double)`
 - > `précision(double)` > `précision(float)`
- `intervalle(long double)`
 - > `intervalle(double)` > `intervalle(float)`

Types réels (2)

Les types réels ne sont pas spécifiés dans la norme du langage, mais il existe la norme IEEE754 (IEEE : Institute of Electrical and Electronics Engineers) qui définit un format standard pour les différents types reel.

Presque tous les constructeurs de matériel informatique et compilateurs respectent cette norme.

IEEE754 définit non seulement les types de données mais aussi comment faire des opérations arithmétiques, arrondir les résultats, etc.

D'autres langages que C utilisent ces types.

Dans ce qui suit, on ne parlera que des types définis dans cette norme.

Exécuter le code C dans le fichier `5_reels/reel1.c` pour afficher les caractéristiques de type réels sur la machine que vous utilisez.

Remarques sur ce qui est affiché:

- min et max sont les valeurs minimale et maximale (en valeur absolue) différentes de zéro
- epsilon est la valeur positive minimale telle que
- $1.0 + \text{epsilon} \neq \text{epsilon}$

Réels normalisés

Dans la mémoire, un nombre réel est enregistré avec un format en base 2 normalisé

$$R = \pm 0.X_0X_1X_2... 2^{\pm Y_0Y_1...}$$

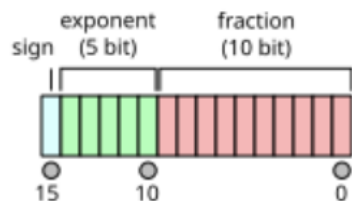
Où $X_0, X_1, \dots, Y_0, Y_1, \dots$ sont des nombres binaires (0 ou 1), mais où **X_0 doit être égal à 1** (c'est la normalisation, l'exposant est calculé en conséquence)

X_0, X_1, \dots est appelé la mantisse (ou fraction en anglais), les chiffres binaires significatifs et Y_0, Y_1, \dots est l'exposant

$$\begin{aligned}\text{Exemple : } 2.75 &= 2 + 0.5 + 0.25 = 1*2^1 + 1*2^{-1} + 1*2^{-2} \\ &= (2^{-1} + 1*2^{-3} + 1*2^{-4})*2^2 \\ &= 0.1011 2^{10} \text{ (normalisé base 2)}\end{aligned}$$

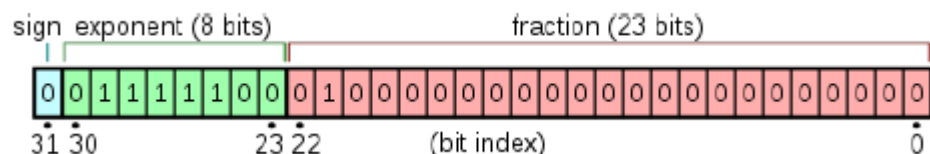
La norme IEEE754 définit 5 types de réels:

- Demi-précision (2 octets, précision ~4 décimales)



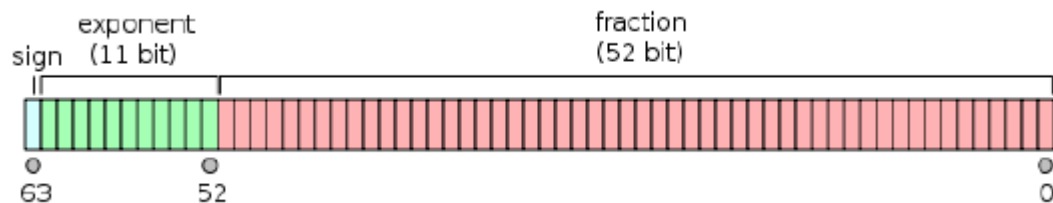
Utilisé pour des usages qui n'ont pas besoin de beaucoup de précision (rendus sur carte graphique, machine learning)

- Simple précision (4 octets, précision ~7 décimales)



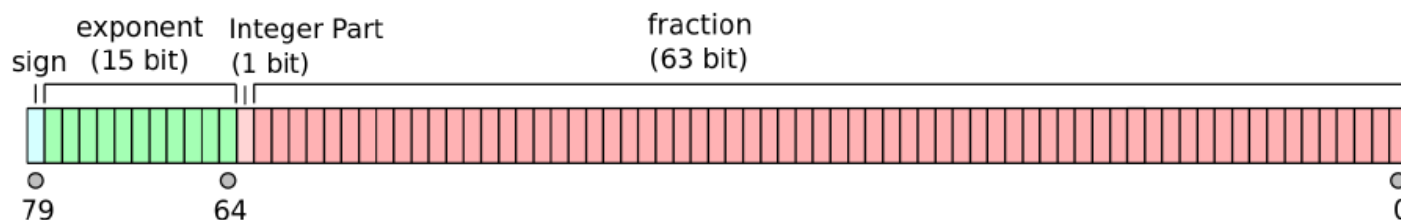
Correspond souvent au type C float

- Double précision (8 octets, précision ~16 décimales)



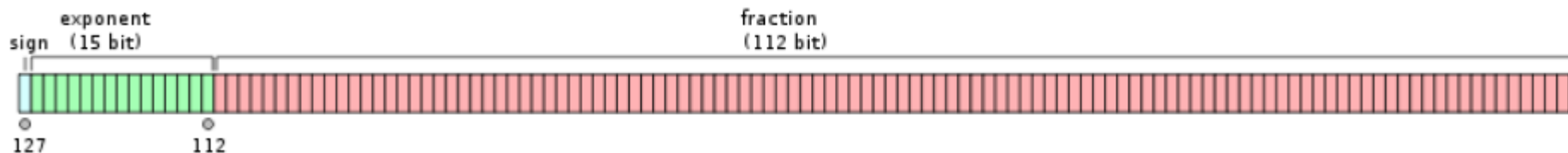
Correspond souvent au type C double

- Précision étendue (10 octets, précision ~18 décimales)



Utilisé en interne dans les processeurs pour améliorer la double précision

- Quadruple précision (16 octets, précision ~34 décimales)



Pas toujours disponible, si oui, correspond au type long double

Ces types IEEE754 ont été conçus pour obtenir le meilleur compromis entre

- La meilleure précision
- Le moins de place mémoire
- La vitesse des opérations arithmétiques

- Les réels C ne peuvent pas représenter exactement tous les réels mathématiques, on parle d'erreur de troncature (on ne garde qu'un nombre fini de chiffres derrière la virgule)
 - Exemple 1 : 0.1 n'est pas représentable exactement dans la mémoire de l'ordinateur (écrire 0.1 comme somme finie de puissance de 2 n'est pas possible)
 - Exemple 2 : 0.625 est représentable exactement :
$$0.625 = 2^{-1} + 2^{-3}$$

- Les opérations arithmétiques introduisent des erreurs d'arrondi, parce que le résultat exact à besoin de plus de décimale ou qu'on soustrait des nombres proches

Par exemple: supposons qu'on calcule en base 10, avec des nombres à 3 chiffres significatifs

$$x = 0.423 \ 10^2 \quad (= 42.3)$$

$$y = 1.21 \ 10^0 \quad (= 1.21)$$

$$x + y = (0.423 + 0.0121) \times 10^2$$

$$= 0.435 \quad (\text{avec erreur d'arrondi, sur machine})$$

$$\neq 0.4351 \quad (\text{valeur exacte qui n'est pas représentable})$$


➤ L'ordre des opérations peut être important

Par exemple, si a , b et c sont des réels mathématiques,

$$a + (b + c) \equiv (a + b) + c \equiv (a + c) + b$$

Si a , b , c sont des réels :

$$a + (b + c) \equiv? (a + b) + c \equiv? (a + c) + b$$



Pas toujours, dépend de la
valeur de a , b , et c

Exécuter plusieurs fois le code [5_reels/reels2.c](#) (qui combine de plusieurs façons 3 nombres pris au hasard)