- Introduction à l'algorithmique et au langage C -

Représentation des données

TP n°5

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2015 - 2016

Avant propos

Les ordinateurs ne manipulent et ne mémorisent que des 0 et des 1. Il faut donc réussir à établir une correspondance entre la représentation habituelle de l'information (un caractère, un nombre, etc.) et sa représentation dans la machine. C'est tout l'objet de ce TP. Et pour ce faire, nous allons mettre à contribution ce que nous avons appris dans le TD « Bases ».

1 Introduction

Un ordinateur est une grosse calculatrice. Il ne manipule donc que des nombres... et en particulier que des nombres représentés en binaire sous la forme d'une succession d'états hauts (le courant passe) ou d'états bas (le courant ne passe pas).

Cela conditionne l'ensemble de l'informatique moderne (du simple calcul flottant au système de freinage du TGV) et c'est donc essentiel de bien comprendre **comment un ordinateur représente des données en machine**.

Bit : Abréviation de **b**inary dig**it**, le mot bit correspond aux 0 et aux 1 manipulés par les ordinateurs.

QUESTION 1

En prenant en compte ce que vous avez appris lors du TD sur les bases, à quoi pourrait correspondre la séquence suivante : 0100 0111 0100 1111 0011 0000 0110 0100

- si c'était un nombre entier positif ?
- si c'était une suite de 4 caractères ?

QUESTION 2

Comment l'ordinateur peut-il faire la différence ?

2 Représentation des données en machine

2.1 Représentation des entiers naturels $\mathbb{N}$

Nous venons de revoir que les entiers naturels (positifs ou nuls) étaient représentables simplement dans toutes les bases et donc en binaire.

$(12)_{10} \Rightarrow (1100)_2$

Cependant, nous pouvons nous interroger sur la manière dont l'ordinateur gère les très grands nombres.

EXERCICE 1

Recopiez le code suivant puis testez-le pour des valeurs de `VAL` égales à 100, 100000 (cent milles), 100000000 (cent millions). (Ne vous préoccupez pas du `"%u"` pour le moment.)

```c
#include <stdio.h>
#define VAL 100

int main() {

   int i;
   for (i = 0 ; i < 100 ; i++) {
      printf("%u ", i * VAL);
   }

   return 0;
}
```

QUESTION 3

Que se passe-t-il pour des nombres très grands ? Quelle semble être la limite ? Quelle est la représentation binaire du dernier nombre représenté correctement ? Celle du premier nombre erroné ? Celle du nombre affiché à la place ? Qu'en déduisez-vous ? *Indice : utilisez la calculette Ubuntu en mode « programmation » pour faciliter vos conversions !*

Nous constatons donc que pour écrire des très grands nombres, il faut une très grande place en mémoire. L'ordinateur semble avoir des nombres butoirs au-delà desquels... il tronque le résultat !

Les ordinateurs sont allergiques à l'infini

Un ordinateur ne gère, par nature, que des quantités finies d'information ce qui le rend allergique à deux niveaux à la notion d'infini : infinité des bornes et continuité.

Le type indispensable

Le nombre de bits utilisé pour représenter un nombre est défini par le **type** de la variable. Grâce à lui, l'ordinateur sait quelle place réserver en mémoire et comment interpréter la série de `0/1` qui s'y trouve.

Pour les entiers naturels (noté `unsigned` car strictement positifs), le nombre de bits maximum n détermine le nombre de valeurs différentes représentables (2^n) et donc les bornes (de 0 à $2^n - 1$).

Type	Architecture [1]	Nb octets	Nb bits	Borne min.	Borne max.		Descripteur
`unsigned char`	x86/x64	1	8	0	$2^8 - 1$	255	`"%hhu"`
`unsigned short` `unsigned short int`	x86/x64	2	16	0	$2^{16} - 1$	65535	`"%hu"`
`unsigned int`	x86/x64	4	32	0	$2^{32} - 1$	$\simeq 4 \times 10^9$	`"%u"`
`unsigned long`	x86	4	32	0	$2^{32} - 1$	$\simeq 4 \times 10^9$	`"%lu"`
`unsigned long int`	x64	8	64	0	$2^{64} - 1$	$\simeq 1.8 \times 10^{19}$	
`unsigned long long` `unsigned long long int`	x86/x64	8	64	0	$2^{64} - 1$	$\simeq 1.8 \times 10^{19}$	`"%llu"`

Octet : Dans un ordinateur, les bits sont regroupés par 8 sous forme d'octet. Un octet peut donc prendre 2^8 soit 256 valeurs différentes. Un nombre sur 2 octets peut prendre 2^{16} soit 65536 valeurs différentes, etc.. L'octet est l'unité de référence pour la taille des fichiers et des mémoires informatiques.

Plusieurs types pour les entiers ?
Jusqu'ici nous avions limité, par soucis de simplicité, le stockage des entiers au type `int`. Nous voyons aujourd'hui qu'il existe une multitude d'autres types ! Mais rassurez-vous, **tout ce que vous avez appris s'applique aussi sur ces nouveaux types** : printf/scanf, fonctions, tableau, etc.

2.2 Représentation des entiers relatifs $\mathbb{Z}$

À une limite arbitraire près, nous pouvons représenter simplement les entiers positifs, mais comment faisons-nous pour représenter un nombre négatif en binaire ?

2.2.1 Solution du bit de signe

La première idée qui nous vient consiste à utiliser le bit de poids fort (le plus à gauche) comme bit de signe. S'il vaut 0, le nombre est positif, s'il vaut 1, il est négatif. Par exemple, avec cette convention le nombre binaire 0010 serait un nombre positif (2 en décimal) et le nombre binaire 1010 serait quant à lui négatif (-2 en décimal).

Nombre négatif et taille en bits
Comme nous utilisons le bit de poids fort comme bit de signe, la représentation des nombres négatifs dépend du nombre de bits choisi ! Ainsi le nombre $(-1)_{10}$ vaudra $(1001)_2$ sur 4 bits et $(1000\ 0001)_2$ sur 8 bits.

1. Pour les processeurs les plus courants (Intel / AMD), nous parlons d'architecture x86 pour ceux en 32 bits et x86-64 ou x64 pour ceux en 64 bits. Depuis plusieurs années la majorité des processeurs sont compatibles avec les deux modes de fonctionnement, seul le choix de l'OS (32 ou 64 bits) détermine alors le comportement des programmes.

Non respect de l'arithmétique entière

Le problème de la solution du bit de signe vient du fait qu'elle ne respecte pas la règle arithmétique : $n - n = n + (-n) = 0$!

En effet (le bit de signe est noté en rouge) :

$$\begin{array}{rcccccc} & (& \mathbf{0} & 0 & 1 & 0 &)_2 \\ + & (& \mathbf{1} & 0 & 1 & 0 &)_2 \\ \hline & (& \mathbf{1} & 1 & 0 & 0 &)_2 \end{array} \qquad \begin{array}{r} (2)_{10} \\ (-2)_{10} \\ \hline (-4)_{10} \text{ !!!} \end{array}$$

Pour utiliser cette représentation, il aurait fallu rajouter de nouveaux circuits électroniques pour la soustraction... Nous devons donc trouver une autre solution.

2.2.2 Solution du Complément à 2

Nous cherchons un système de représentation des nombres négatifs qui respecte la règle arithmétique $n - n = n + (-n) = 0$. Il nous faut donc trouver le nombre x qui, en base 2, résout l'équation $n + x = 0$.

Si nous prenons comme exemple -2, nous cherchons x tel que l'équation $0010 + x = 0000$ soit vraie. Nous avons donc $x = 1110$. Nous appelons ce nombre le **complément à 2** de n et nous le noterons $Ca2(n)$.

Il existe un moyen simple de trouver ce complément à 2 qui consiste à calculer d'abord ce que nous appelons le complément à 1.

Ca1(n) : Le complément à 1 d'un nombre se calcule :
- en prenant la valeur absolue du nombre,
- en inversant cette valeur bit à bit (0 pour 1 et 1 pour 0),

Ca1(n) = $|n_2| \oplus (2^n - 1)$

Ca2(n) : Le complément à 2 d'un nombre se calcule :
- en prenant le Ca1 du nombre,
- en ajoutant 1 au résultat.

Ca2(n) = Ca1(n) + 1.

Prenons un exemple : soit le nombre décimal -4, nous cherchons sa notation binaire en complément à deux sur 4 bits. Nous avons alors :

$(n)_{10}$	$(\|n\|)_{10}$	$(\|n\|)_2$	Ca1(n) (inversion bit à bit)	Ca2(n)
-4	4	0100	1011	1100

Avec le détail :

$$\begin{array}{rcccccc} & (& 0 & 1 & 0 & 0 &)_2 \\ \oplus & (& 1 & 1 & 1 & 1 &)_2 \\ \hline & (& 1 & 0 & 1 & 1 &)_2 \end{array} \qquad \begin{array}{r} (|n|)_2 \\ 2^n - 1 \\ \hline Ca1(n) \end{array} \qquad \begin{array}{rcccccc} & (& 1 & 0 & 1 & 1 &)_2 \\ + & (& 0 & 0 & 0 & 1 &)_2 \\ \hline & (& 1 & 1 & 0 & 0 &)_2 \end{array} \qquad \begin{array}{r} Ca1(n) \\ 1 \\ \hline Ca2(n) \end{array}$$

Nous pouvons vérifier que nous avons bien $4 + Ca2(-4) = 0$.

$$\begin{array}{rcl} {\scriptstyle 1}\quad {\scriptstyle 1} & & \\ (\ 0\ 1\ 0\ 0\ )_2 & & (4)_2 \\ +\ (\ 1\ 1\ 0\ 0\ )_2 & & Ca2(-4) \\ \hline (\ 0\ 0\ 0\ 0\ )_2 & & 0 \end{array}$$

Retenue perdue ?

Qu'arrive-t-il à la dernière retenue ? Nous avons vu avec les nombres entiers positifs que les chiffres qui dépassent le nombre de bits réservés sont tronqués ! Nous avons donc bien 0.

Représentation de tous les entiers relatifs

Nous avons vu que le complément à 2 permettait de représenter les nombres négatifs de manière à respecter les règles algébriques. Comment représentons-nous les nombres positifs ? Comme nous le faisions avec les entiers naturels.

Nous allons traiter différemment les nombres positifs et les nombres négatifs :

- Les nombres positifs seront simplement représentés en base 2.
- Les nombres négatifs seront représentés en complément à 2.

Comment reconnaitre un nombre négatif ?

Une fois en binaire, comment savoir si le nombre représenté est un nombre positif ou négatif ? Nous pouvons remarquer que par construction tous les nombres en Ca2 commencent par le bit 1 ! Le bit de poids fort nous sert donc (à nouveau) comme bit de signe pour l'identification, tout en respectant l'algèbre.

QUESTION 4

Représentez les nombres suivants en binaire (sur 8 bits). Attention, les bits non significatifs en binaire (0 devant) sont très importants quand nous utilisons le complément à 1, n'oubliez donc pas de marquer les **8** bits systématiquement !

n	$(\lvert n \rvert)_2$	(Si nécessaire) Ca1(n) (inversion bit à bit)	(Si nécessaire) Ca2(n)	Représentation signée
-18				
-128				
96				

QUESTION 5

Retrouvez les nombres qui correspondent à la représentation suivante (sur 8 bits).

Représentation signée	(Si nécessaire) Ca2(n)	Ca1(n) (inversion bit à bit)	$(\|n\|)_2$	n
1110 0101				
0110 0101				
1111 1111				

Sans dessus-dessous

Affichons côte à côte la représentation binaire (sur 4 bits) et leur correspondance en entier naturel ($\mathbb{N}$) et en entier relatif ($\mathbb{Z}$).

Représentation binaire (4 bits)	$\mathbb{N}$	$\mathbb{Z}$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Que remarquez-vous ?

Les représentations coïncident sur la première moitié de l'espace (de 0 à $2^{n-1}-1$), puis divergent.

En complément à 2, nous observons un saut de la plus grande valeur représentable (7) vers la plus petite valeur représentable (-8). C'est ce que nous appellerons un débordement (*integer overflow*) et nous en verrons les dangers un peu plus tard.

En revanche nous retrouvons bien (-1) + 1 = 0 car $(1111)_2 + (0001)_2 = (0000)_2$ à la retenue près.

Nous pouvons noter qu'en représentation signée les bornes vont de -2^{n-1} à $2^{n-1}-1$, le zéro étant compté dans les positifs.

Division de l'ensemble des nombres

En utilisant 1 bit pour le signe, nous divisons par deux l'ensemble des nombres positifs représentables (jusqu'à $2^{n-1}-1$), l'autre moitié est alors utilisée par les nombres négatifs.

Nous découvrons donc nos nouveaux types correspondants aux types signés (le mot clef `signed` est optionnel : `signed char` ⇔ `char`).

Type	Architecture	Nb octets	Nb bits	Borne min.	Borne max.		Descripteur
`char`	x86/x64	1	8	-2^7	2^7-1	127	`"%hhd"`
`short` `short int`	x86/x64	2	16	-2^{15}	$2^{15}-1$	32767	`"%hd"`
`int`	x86/x64	4	32	-2^{31}	$2^{31}-1$	$\simeq 2\times10^9$	`"%d"`
`long`	x86	4	32	-2^{31}	$2^{31}-1$	$\simeq 2\times10^9$	`"%ld"`
`long int`	x64	8	64	-2^{63}	$2^{63}-1$	$\simeq 9\times10^{18}$	
`long long` `long long int`	x86/x64	8	64	-2^{63}	$2^{63}-1$	$\simeq 9\times10^{18}$	`"%lld"`

EXERCICE 2

Sauvegardez la valeur `0x7FFFFFFF` dans une variable `iVal` de type `int`. Affichez-la une première fois. Incrémentez-la puis affichez-la à nouveau. Quel est le phénomène que vous observez ?

QUESTION 6

Que produit le code suivant ? Que faut-il faire pour le corriger ?

```c
#include <stdio.h>

int main() {
   unsigned int iVal = 10;

   while (iVal >= 0) {
      printf("iVal = %u\n", iVal);
      iVal--;
   }

   return 0;
}
```

2.3 Représentation des nombres réels $\mathbb{R}$?

Nous avons vu comment représenter des nombres entiers, positifs ou négatifs. Mais comment faire pour les calculs mathématiques qui demandent des nombres réels : sinus/cosinus, pourcentage, calculs matriciels, etc. ?

Les ordinateurs sont définitivement allergiques à l'infini...

Les nombres réels sont deux fois plus indigestes pour un ordinateur car non seulement ils sont en nombre infini comme dans $\mathbb{N}$ mais ils sont en plus **continus**. C'est-à-dire qu'entre deux nombres réels, il y a encore une infinité de nombres !

Nous allons donc représenter les nombres réels par des approximations : les **nombres à virgules flottantes (dits aussi flottants)**. Il s'agit en quelque sorte d'une forme de notation scientifique tronquée.

Rappel sur la notation scientifique

Cette notation consiste à écrire les nombres réels comme le produit d'un nombre réel n (tel que $1 <= n < 10$) et d'une puissance de 10.
Par exemple : $1234 \Rightarrow 1,234 \times 10^3$ ou $32,11 \Rightarrow 3,211 \times 10^1$ ou encore $0,02 \Rightarrow 2,0 \times 10^{-2}$.

Une notation scientifique tronquée se traduit par le fait que le nombre de chiffres après la virgule est majoré (d'où l'approximation). Il en va de même pour la taille de l'exposant (puissance de 10).

Tout cela peut se traduire en binaire de manière similaire. Pour cela nous utilisons une représentation binaire **à virgule**.

Nombre binaire à virgule ? 0_o quésako ?

Et bien le principe est le même qu'en base 10. Tous les digits devant la virgule se verront attribuer des puissances de 2 positives (ce que nous avons vu jusqu'à maintenant). Et tous ceux qui se trouvent derrière se verront attribuer des puissances de 2 négatives.

$$\begin{aligned}(1101.11)_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \\ &= 8 + 4 + 1 + \tfrac{1}{2} + +\tfrac{1}{4} \\ &= 13,75\end{aligned} \quad .$$

Et la notation scientifique pour le même nombre donne : $(1101.11)_2 = +(1.\mathbf{10111})_2 \times 2^{\mathbf{3}}$

Un nombre flottant en binaire, comme en notation décimale est un nombre dont la quantité de chiffres après la virgule ainsi que la taille de l'exposant sont majorées.

Dans l'ordinateur : signe, exposant et mantisse

Les nombres à virgules flottantes sont mémorisés dans l'ordinateur par trois séries de 0/1 qui se suivent :
- Le **bit de signe** qui indique si le nombre est positif (0) ou négatif (1).
- L'**exposant** qui est codé sous la forme d'un nombre positif biaisé [3].
- La **mantisse**, qui correspond au nombre en notation scientifique mais dont nous stockons uniquement les chiffres après la virgule (le premier 1 est implicite).

En C, les flottants sont codés selon la norme IEEE 754 qui définit les types suivants :

Type	Architecture	Nb octets	Nb bits				Biais	Descripteur
			Total	Signe	Mantisse	Exposant		
`float`	x86/x64	4	32	1	23	8	127	`"%f"`
`double`	x86/x64	8	64	1	52	11	1023	`"%lf"`
`long double`	x86 x64	12 16	80 [4]	1	64	15	16383	`"%Lf"`

3. Nous soustrayons à ce nombre une valeur fixe qui permet d'obtenir les nombres négatifs par exemple, si l'exposant est codé sur 8 bit, le biais vaudra $2^7 - 1 = 127$.
4. Les `long double` ne sont pas précisément standardisés par la norme et résultent d'un consensus chez les constructeurs. Ce qui explique leur implémentation étrange sur 80 bits. Le nombre d'octets donné correspond à l'espace mémoire réservé par le système.

Représenter l'infini

Notons que la représentation à virgule flottante intègre deux valeurs particulières supplémentaires : `INFINITY` qui vaut l'infini et `NAN` (Not a Number) qui correspond au résultat d'opérations impossibles : $\sqrt{-1}$ ou $\frac{0}{0}$ par exemple.

Écrire une valeur flottante

Si vous ne le précisez pas, les valeurs à virgule que vous écrivez dans un programme sont considérés comme des `double` (`3.14`). Pour écrire un `float` nous devons rajouter le suffixe `f`, soit `3.14f` et pour un `long double` le postfixe `L`, soit `3.14L`.

Afficher plus de décimales

Nous avons vu que les descripteurs des types flottants en `"%f"`, `"%lf"` et `"%Lf"`. Par défaut, ces descripteurs affichent 6 chiffres après la virgule. Pour en afficher plus, nous utilisons la notation `"%.Yf"` où Y correspond au nombre de chiffres après la virgule que nous souhaitons afficher.
Par exemple : `"%.3lf"`.

QUESTION 7

Combien de décimales exactes de PI pouvons-nous stocker dans un `float`, dans un `double` et dans un `long double` ? Pour répondre à cette question, écrivez un programme qui :

- Définit la constante PI 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196L.
- Définit une variable `fVar` de type `float` et qui lui affecte la constante PI.
- Définit une variable `dVar` de type `double` et qui lui affecte la constante PI.
- Définit une variable `ldVar` de type `long double` et qui lui affecte la constante PI.
- Affiche sur quatre lignes la valeur exacte puis les valeurs des trois variables avec 20 chiffres après la virgule.

QUESTION 8

Qu'affiche le code suivant ? Que s'est-il passé ?

```c
#include <stdio.h>

int main() {
   double dSum = 0;
   int i;
   for (i = 0; i < 10; i++) {
      dSum += 0.1;
   }
   printf("dSum = %lf\n", dSum);
   if (dSum == 1.0) {
      printf("The continum of reality is safe.\n");
   } else {
      printf("Nothing makes sense anymore... are we in the matrix?\n");
   }

   printf("0.1 = %.20lf\n", 0.1);
   printf("1.0 - dSum = %.20lf\n", 1.0 - dSum);

   return 0;
}
```

Précision infinie et fration infinie

Nous nous rappelons qu'en base 10 certaines fractions possèdent une infinité de décimale ($\frac{1}{3} = 0.3333...$ par exemple). Ces nombres vont poser problème lors de leur représentation sous la forme de nombres à virgule flottante car leur mantisse sera tronquée. **Les nombres à virgule flottante ne sont donc que des APPROXIMATIONS des nombres réels et il faut les utiliser avec prudence !**

Attention, dans un ordinateur les nombres sont stockés sous forme binaire, ce sont donc les fractions qui possèdent une infinité de décimales **en base 2** qui poseront problème.

En particulier $(0.1)_{10} = (0.00011001100110011...)_2$.

2.4 Représentation du texte

Jusqu'ici nous avons considéré que les variables de type `char` contenaient des caractères (c'est-à-dire des symboles représentables comme des lettres, des chiffres, des symboles de ponctuation ou d'espacement). Et pourtant nous avons aussi placé le type `char` dans la catégorie des entiers signés (sur 1 octet : de -128 à +127) !

En fait, les deux définitions sont vraies. Les caractères et les entiers sont en effet reliés par une **table de codage**. Dans le cas du C, c'est la table ASCII qui est utilisée et qui relie les nombres de 0 à 127 aux caractères courants non accentués.

Nous avons vu, particulièrement dans et le TP « *Chaînes de caractères* » comment manipuler des textes avec des tableaux de caractères et nous ne reviendrons pas dessus ici.

2.5 Représentation des booléens

Pour représenter des booléens (variables manipulées par l'algèbre de Boole qui valent VRAI ou FAUX) utilisés dans les tests de comparaison, il nous suffirait d'un seul bit (0 : FAUX ou 1 : VRAI).

En C, les booléens sont représentés par des entiers (`int`) sur le principe « valeur égale 0 » ⇒ FAUX et « valeur différente de 0 » ⇒ VRAI, ce qui peut entrainer des situations problématiques.

Ne confondez pas test et expression

Le choix de représentation du C permet des écriture de test (valant VRAI ou FAUX) sous la forme d'expression (valant une valeur numérique). C'est une très mauvaise habitude à prendre qui entraine des erreurs fréquentes et une difficulté de relecture.

Par exemple, ne pas écrire :

```c
if (val%2) { // UGLY
   printf("Le nombre est impair\n");
}
```

mais plutôt :

```c
if (val%2 == 1) { // NICE
   printf("Le nombre est impair\n");
}
```

2.6 Représentation des adresses

Nous avons vu dans le TP sur les pointeurs que les adresses sont des nombres. L'ordinateur les manipules aisément. Nous noterons simplement que le nombre de bits nécessaire à les stocker dépend de l'architecture : 32 bits sur une architecture x86 et 64 bits sur une architecture x64.

Nous nous rappellerons aussi que les pointeurs n'ont pas de type propre, mais que les types pointeurs sont construits à partir des types des variables : `unsigned char*`, `short*`, `long double*`, etc.

2.7 Représenter des mesures physiques : images et sons

Les ordinateurs ne se contentent pas de manipuler des abstractions (nombres, lettres), ils sont aussi capables de manipuler des informations provenant du monde physique (température, pression, lumière, rayonnement ou ondes sonores, etc.).

Comment font-ils pour représenter ces informations continues par nature ?

En effectuant une numérisation, c'est-à-dire une série de mesure des grandeurs que nous désirons représenter :

- selon un axe spatial (tous les millimètres par exemple) : nous parlons alors d'**échantillonnage spatial**,
- selon un axe temporel ou fréquentiel (toutes les microsecondes par exemple) : nous parlons alors d'**échantillonnage temporel**,
- selon un axe d'intensité (tous les millipascal par exemple) : nous parlons alors de **quantification**.

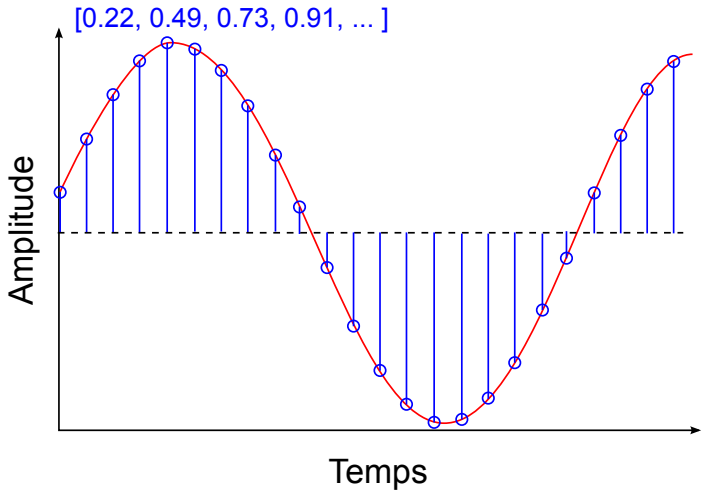


Figure 1 – ***Exemple de numérisation d'une onde sonore.***

Lorsque nous cherchons à représenter un son dans la mémoire de l'ordinateur, nous allons stocker les échantillons mesurés selon l'axe temporel (comme nous pouvons le voir sur la figure 1 page 11), dans un tableau de flottants. Nous pouvons sauvegarder, manipuler ou restituer cette information.

Traitement du signal
Vous verrez dans le cours de Signal, les contraintes posées par cette numérisation.

Lorsque nous cherchons à représenter une image, nous allons avoir une triple numérisation : selon les deux axes du plan (échantillonnage spatial) plus une quantification de la couleur ! Nous verrons par la suite, comment sont stockées ces informations en mémoire, soit avec des tableaux tridimensionnels, soit avec des tableaux unidimensionnel en utilisant plusieurs astuces.

QUESTION 9

À votre avis, combien de numérisations nécessite une vidéo ?

3 Transtypage

Transtypage : Le transtypage (appelé *cast* en anglais) consiste à transférer le contenu d'une variable à une autre variable d'un type différent (`char` ⇒ `int`, `unsigned int` ⇒ `int` ou `double` ⇒ `int`).

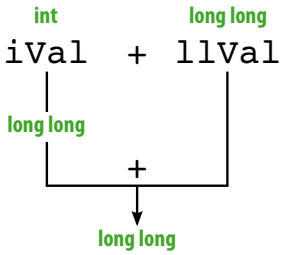
Le transtypage est utilisé de manière **implicite** lors de l'utilisation des opérateurs arithmétiques, des opérateurs de comparaison ou lors des passages d'arguments pour permettre un comportement adéquat du programme.

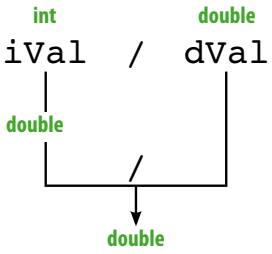
Il peut aussi être demandé de manière **explicite** par le programmeur.

3.1 Promotion numérique

Les opérateurs arithmétiques (addition, ...) nécessitent des opérandes de même type pour pouvoir fonctionner sans heurt.

Or nous venons de voir que les variables peuvent avoir des tailles très différentes. Que se passe-t-il lors de l'addition ou de la division de deux variable de type différent ?





Promotion implicite

Lors de l'exécution des opérateurs, les variables "plus petites" seront converties implicitement dans un type "plus grand" selon la série suivante : `char` ⇒ `short` ⇒ `int` ⇒ `long long` ⇒ `float` ⇒ `double` ⇒ `long double`. (La conversion conserve la valeur autant que possible : à la précision de la représentation flottante près.)

Ce transtypage s'effectue aussi lors de l'affectation :

```
int iVal = -12;
long long llVal = iVal; // llVal equal -12
```

Transtypage signé / non signé

Attention, le transtypage d'un type signé vers un type non signé conserve la signature des bits et non pas la valeur !

```
int iVal = -12;
unsigned int uVal = iVal; // -12 is not a natural integer !
printf("uVal = %u\n", uVal); // ?
```

QUESTION 10

Recopiez le code précédent. Quelle est la valeur de `uVal` ? Qu'observez-vous ?

3.2 Troncature

Que se passe-t-il si l'on essaye d'affecter la valeur d'une variable dans une variable "plus petite" ?

QUESTION 11


À l'aide du programme ci-dessous, complétez le tableau suivant.
(Remarque : `"%x"` permet d'afficher la représentation hexadécimal d'un entier).

```c
#include <stdio.h>
int main() {
   unsigned long long llVal = 0xFFFFFFFFFFFFFFFF;
   unsigned int iVal = llVal;
   unsigned short sVal = iVal;
   unsigned char cVal = sVal;

   printf("llVal = %016llx %llu\n", llVal, llVal);
   printf("iVal = %016x %u\n",  iVal,  iVal);
   printf("sVal = %016x %u\n",  sVal,  sVal);
   printf("cVal = %016x %u\n",  cVal,  cVal);

   return 0;
}
```

`long long`	8o	$(FFFFFFFFFFFFFFFF)_{16}$	$(18446744073709551615)_{10}$
`int`	4o		
`short`	2o		
`char`	1o		

QUESTION 12


Qu'observez-vous ?

3.3 Transtypage explicite

Nous pouvons forcer le transtypage de manière explicite avec la notation

```
(type) variable
```

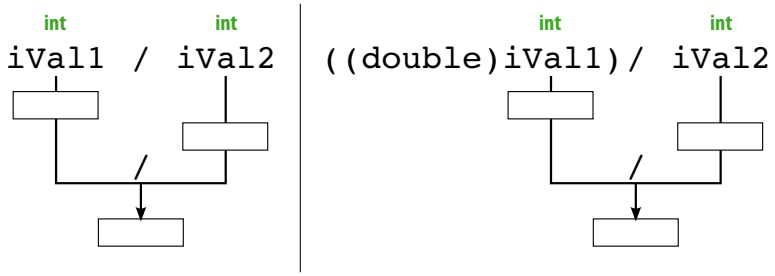
où `type` est le type de destination.

Pour prendre un exemple concret :

```c
double dVal;
int iVal1 = 12, iVal2 = 7;
dVal = (iVal1 / iVal2);
printf("%d / %d = %lf\n", iVal1, iVal2, dVal);
dVal = ((double)iVal1 ) / iVal2;
printf("%d / %d = %lf\n", iVal1, iVal2, dVal);
```

QUESTION 13


Qu'affiche le programme précédent ? Complétez le schéma suivant.



3.4 Transtypage et fonctions

Lorsqu'on passe une variable en argument d'une fonction qui attend un paramètre une valeur d'un autre type, le programme effectue un transtypage.

Cela peut provoquer des comportements étranges comme nous allons l'illustrer immédiatement.

Imaginons une fusée et un circuit de test, qui déclenche l'autodestruction de cette même fusée au cas où celle-ci retomberait. Notre simulateur de fusée hautement perfectionné envoie les altitudes successives à notre circuit de test et nous allons pouvoir vérifier que tout va bien.

EXERCICE 3 (*Autocorrect*[5])

 Écrire le programme minimal.

EXERCICE 4 (*Autocorrect*)

 Écrire la fonction `rocketTest` qui retourne un `int` et qui prend en paramètre deux variables `oldAltitude` et `newAltitude` de type `short`. Cette fonction :

- Affiche le message `"old altitude: XX, new altitude: YY\n"` ou `XX` et `YY` sont remplacés par les valeurs d'altitude.
- Teste si la fusée est en montée ou en stationnaire. Dans ce cas, elle renvoie la valeur `1`. Dans le cas contraire, elle affiche le message `"Falling!\n"` puis renvoie la valeur `0`.

Attention, les textes affichés doivent respecter la consigne au caractère près.

Exemples d'affichages :

```
old altitude: 0, new altitude: 1000
```

```
old altitude: 2000, new altitude: 1000
Falling!
```

EXERCICE 5 (*Autocorrect*)

 Écrire la fonction `runLoop` qui ne retourne rien et qui prend en paramètre une variable `speed` de type `int`. Cette fonction :

- Effectue une boucle qui incrémente une variable `altitude` de type `int` de 0 à 300000 mètres d'un pas égal à `speed` (i.e. à chaque tour l'altitude augmente de `speed` mètres).
- Dans cette boucle :
  - On affiche le message `"Actual altitude: XX, "` où `XX` correspond à l'altitude réelle. (La suite du message sera produit par la fonction appelée, ne mettez pas de `'\n'`.)
  - On appelle la fonction `rocketTest` en lui passant l'altitude précédente et l'altitude actuelle. (Attention, il faut donc sauvegarder la valeur d'altitude précédente qui est initialisée à 0 au début de la fonction !)
  - Dans le cas où cette fonction retourne `0`, on affiche le message `"Boom!\n"` et on sort de la boucle avec `break`.
  - Mettez à jour la valeur sauvegardée de l'altitude précédente.

Attention, les textes affichés doivent respecter la consigne au caractère près.

Exemple d'affichage :

```
Actual altitude: 0, old altitude: 0, new altitude: 0
Actual altitude: 1000, old altitude: 0, new altitude: 1000
Actual altitude: 2000, old altitude: 1000, new altitude: 2000
```

QUESTION 14

 Qu'observez-vous ? Comment expliquez-vous ce phénomène ?

Pour en savoir plus sur les évènements qui ont inspiré ce TP :
http://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5

5. http://autocorrect.esiea.fr ;-)