Le stockage de l'information

STOCKAGE DE L'INFORMATION 1

LE LANGAGE BINAIRE 1
LES DONNEES STOCKEES : NOMBRES ET CARACTERES 1
NOMBRES ENTIERS 2
RAPPELS SUR LES BASES 2
CONVERSIONS ENTRE BASES 3
LES NOMBRES ENTIERS SIGNES 7
LA NOTATION EN COMPLEMENT A 2 9
LES NOMBRES REELS 11
REPRESENTATION EN VIRGULE FIXE 11
REPRESENTATION EN VIRGULE FLOTTANTE, STANDARD IEEE-754 12
EN CONCLUSION 14

MANIPULATIONS DES DONNEES, PREMIERS PROGRAMMES 15

LES TYPES DE DONNEES 15
LES VARIABLES 16
QU'EST-CE QU'UNE VARIABLE ? 16
QUELQUES ELEMENTS DE SYNTAXE POUR LE LANGAGE ALGORITHMIQUE 17
OPERATIONS ET OPERATEURS DE BASE 19
AFFECTATION 19
LES CONSTANTES 19
OPERATEURS ARITHMETIQUES 20
RETOUR SUR L'AFFECTATION : 21
OPERATEURS D'ENTREE/SORTIE 22
LES COMMENTAIRES : 25

EXEMPLES DE SYNTHESE, PARTICULARITES DES TYPES 25

AFFECTATIONS ENTRE TYPES DIFFERENTS. 26
CHANGER LE TYPE D'UNE EXPRESSION 28

Stockage de l'information

Le langage binaire

Un ordinateur ne connaît à la base, que deux valeurs : il est constitué en majorité de transistors qui agissent comme des interrupteurs : ils ont donc deux états distincts (ouvert ou fermé) qui correspondent à deux niveaux de tension électrique. Pour plus de lisibilité, on note par convention ces états 0 et 1 : c'est l'origine de l'utilisation du binaire en informatique et en électronique.

Bien entendu, on ne stocke pas uniquement que des informations binaires ! l'unité élémentaire d'information qui peut être stockée est appelée bit. Un bit stocke une valeur binaire, il est égal à 0 ou à 1. Toute information est stockée dans un bit ou dans un groupe de bits.

Un groupe de 8 bits est appelé octet (byte en anglais), un groupe de bits d'une taille supérieure à 8 est appelé mot (word) auquel on accole la taille du groupement. Ainsi, un regroupement de 32 bits est appelé : mot 32 bits.

En résumé :

Taille du groupement	Nom français	Nom anglais
1	bit	bit
8	octet	byte
16	mot 16 bit	16 bit word
32	mot 32 bit	32 bit word

Les données stockées : nombres et caractères

Un ordinateur ne stocke donc que des bits ou groupements de bits, et il faut utiliser des conventions pour décider de la manière dont il doit interpréter cette valeur. En effet, l'ordinateur ne sait absolument rien de la manière dont l'utilisateur ou le programmeur veut utiliser les valeurs qu'il stocke. Doit-il considérer qu'un groupement de bits doit stocker un nombre entier, un nombre à virgule, un caractère imprimable, du texte ? c'est au programmeur de le lui indiquer !

Nombres entiers

Un bit ne peut stocker que deux valeurs entières : 0 et 1, voilà qui est bien peu pour une machine sensée avoir des capacités de calcul extraordinaires ! On peut utiliser des groupements de bits pour stocker un nombre entier, en utilisant une convention pour connaître la valeur de ce nombre. Pour cela, on utilise simplement une notation dite positionnelle. Puisque les ordinateurs stockent des bits, on décide qu'un groupement de bits est un nombre noté en base 2, que l'on lit de la gauche vers la droite, comme pour un nombre noté en base 10, ceux que nous utilisons traditionnellement dans la vie quotidienne.

Rappels sur les bases

N'importe quel nombre peut s'écrire, en notation positionnelle (celle que nous utilisons couramment : la position d'un chiffre dans un nombre indique la puissance de la base à laquelle il correspond. La notation en chiffres romains par exemple n'est pas une notation positionnelle), dans une base b quelconque. En base b, on dispose de b symboles ou chiffres, allant en général de 0 à b-1. Pour les valeurs de $b > 10$, il est possible d'utiliser des caractères pour désigner les chiffres (nous verrons l'exemple de la base 16 ou hexadécimale).

un nombre donné p, dans la notation positionnelle en base b, s'écrit :

$a_n a_{n-1} \ldots a_0, a_{-1} \ldots a_{-m}$, avec m et $n > 0$, où les différents a_i sont les chiffres, choisis parmi les b-1 de la base. Il est possible de préciser la base dans laquelle est écrite un nombre en indiquant celle-ci en indice après le nombre : p en base b se note p_b.

La valeur du nombre p est donnée par la formule suivante :

$$p = a_n.b^n + a_{n-1}.b^{n-1} + \ldots + a_1.b^1 + a_0.b^0 + a_{-1}.b^{-1} + \ldots + a_{-m}.b^{-m} = \sum_{i=-m}^{n} a_i.b^i$$

Les bases les plus souvent employées en informatique sont les suivantes :

B	**Nom**	**Chiffres**
2	Binaire	{0; 1}
8	Octale	{0; 1; 2; 3; 4; 5; 6; 7}
10	Décimale	{0; 1; 2; 3; 4; 5; 6; 7; 8; 9}
16	Hexadécimale	{0; 1; 2; 3; 4; 5; 6; 7; 8; 9; A; B; C; D; E; F}

Table 1 : les chiffres des différentes bases couramment utilisées en informatique

Mis à part notre bonne vieille base décimale, les autres sont toutes des bases puissances de 2. La base octale est relativement peu employée (d'ailleurs nous ne l'emploierons pas dans ce cours).

Conversions entre bases

Bases 2, 8 et 16 vers base 10 :

Des bases 2, 8 et 16 vers la base décimale, il suffit d'utiliser la formule donnée ci-dessus pour obtenir la valeur du nombre. La valeur d'un chiffre a_i doit être retranscrite en base 10 pour être multipliée par b^i. En base 16, on retranscrit le chiffre **A** par sa valeur décimale **10**, B par **11**, C par **12**, D par **13**, E par **14** et F par **15**.

Exemples :

- Conversion en base 10 de $\mathbf{10011011,11011_2}$:

Chiffres a_i	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	,	a_{-1}	a_{-2}	a_{-3}	a_{-4}	a_{-5}
Ecriture du nombre	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**,**	**1**	**1**	**0**	**1**	**1**
Puissance de 2	7	6	5	4	3	2	1	0		-1	-2	-3	-4	-5
Valeur de b^i	128	64	32	16	8	4	2	1		0,5	0,25	0,125	0,0625	0,03125

D'où $\mathbf{10011011,11011_2}$ = 1x128 + 0x64 + 0x32 + 1x16 + 1x8 + 0x4 + 1x2 + 1x1 + 1x0,5 + 1x0,25+ 0x0,125 + 1x0,0625 + 1x0 ,03125 = $\mathbf{155,71875_{10}}$

- Conversion en base 10 de **625,702$_8$** :

Chiffres a_i	a_2	a_1	a_0	,	a_{-1}	a_{-2}	a_{-3}
Ecriture du nombre	**6**	**2**	**5**	**,**	**7**	**0**	**2**
Puissance de 8	2	1	0		-1	-2	-3
Valeur de b^i	64	8	1		0,125	0,015625	0,001953125

D'où **625,702$_8$** = 6x64 + 2x8 + 5x1 + 7x0,125 + 0x0,015625 + 2x0,001953125

= **405,87890625$_{10}$**

- Conversion en base 10 de **F23,A0$_{16}$**

Chiffres a_i	a_2	a_1	a_0	,	a_{-1}	a_{-2}
Ecriture du nombre	**F**	**2**	**3**	**,**	**A**	**0**
Puissance de 16	2	1	0		-1	-2
Valeur de b^i	256	16	1		0,0625	0,00390625

D'où **F23,A0$_{16}$** = 15x256 + 2x16 + 3x1 + 10x0,0625 + 0x0,00390625 = **3875,625$_{10}$**

Conversions entre les bases 2 et 16 :

Ces conversions sont les plus faciles à réaliser, puisqu'il s'agit simplement de regrouper ou de séparer les chiffres pour obtenir le résultat de la conversion.

Considérons un ensemble de 4 bits (nommé quartet ou nibble en anglais) $b_3b_2b_1b_0$: le nombre décimal représenté par ce quartet est donc : $8.b_3+4.b_2+2.b_1+b_0$. Selon les valeurs des bi (0 ou 1), ce nombre peut prendre toutes les valeurs entre 0 et 15 : il s'agit donc d'un digit hexadécimal. De même, on montre facilement qu'un digit hexadécimal correspond à un quartet binaire.

Base 2 vers base 16 : partant de la virgule (éventuelle), il suffit de regrouper les bits par quartets et de traduire chaque quartet en le digit hexadécimal correspondant. Il peut être nécessaire de compléter l'écriture binaire à gauche et à droite par des 0 afin de constituer ces quartets.

Exemple : conversion de $100111,1011011_2$ en base 16

Constitution et traduction des quartets (les 0 de remplissage sont indiqués en italique) :

$$\underbrace{00}_{}\underbrace{10}_{2}\ \underbrace{0111}_{7}\ ,\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}$$

d'où $100111,1011011_2 = 27,B6_{16}$

Base 16 vers base 2 : c'est encore plus simple, il suffit de remplacer chaque digit hexadécimal par le quartet binaire correspondant.

Conversion de la base 10 vers d'autres bases :

La conversion de la base 10 vers des bases puissance de 2 peut amener à des développement après la virgule infinis. Peu importe, puisque, très majoritairement, ce sont les notations et les représentations entières qui sont à la base des calculs effectués par un ordinateur. Il faut cependant garder ce constat à l'esprit quand on travaille avec des nombres réels, nous y reviendrons un peu plus loin dans le cours.

Conversion vers la base binaire : pour convertir un nombre décimal en base binaire, il faut diviser ce nombre successivement par 2, et noter le reste de cette division entière (qui est soit 1, soit 0). La conversion est terminée lorsque le quotient *Q* de la division est égal à 0 et le reste *R* est égal à 1. Si l'on effectue ces divisions successives ligne par ligne, on obtient le résultat en lisant la colonne des restes du bas vers le haut.

Exemple : conversion de 163_{10} en base 2

Nombre	Division	quotient *Q*	reste *R*
163	/ 2 =	81	1
81	/ 2 =	40	1
40	/ 2 =	20	0
20	/ 2 =	10	0
10	/ 2 =	5	0
5	/ 2 =	2	1
2	/ 2 =	1	0
1	/ 2 =	0	1

Lecture du résultat

D'où $163_{10} = 10100011_2$.

Conversion vers la base hexadécimale : pour convertir un nombre décimal en base 16, il faut diviser ce nombre successivement par 16, et noter le reste de cette division entière (qui est compris entre 0 et 15). La conversion est terminée lorsque le quotient *Q* de la division est égal à 0 et le reste *R* est différent de 0. Si l'on effectue ces divisions successives ligne par ligne, on obtient le résultat en lisant la colonne des restes du bas vers le haut. Chacun des restes *R* successifs fournit un chiffre hexadécimal qu'il faut donc retranscrire dans cette base.

Exemple : conversion de 49380_{10} en base 16 :

Nombre	Division	Quotient *Q*	reste *R*
49380	/ 16 =	3086	4_{10} □ 4_{16}
3086	/ 16 =	192	14_{10} □ E_{16}
192	/ 16 =	12	0_{10} □ 0_{16}
12	/ 16 =	0	12_{10} □ C_{16}

↑ Lecture du résultat

D'où $49380_{10} = C0E4_{16}$

Bit de poids fort / bit de poids faible :

Le bit de poids fort d'un nombre binaire, aussi appelée MSB (Most Significant Bit) est le bit situé à gauche de l'écriture de ce nombre. Le bit de poids faible (ou LSB pour Least Significant Bit) est celui le plus à droite. Ces bits sont ainsi dénommés en fonction de leur importance relative sur la valeur du nombre qui est représenté. Une erreur sur le bit de poids faible (provoquée, par exemple, par une interférence électrique dans un processeur ou intervenant lors d'une transmission) changera la valeur de ce nombre de 1; la même erreur sur le bit de poids fort changera ce nombre de 2^{N-1} s'il est écrit sur *N* bits.

Arithmétique des nombres entiers pour un ordinateur :

Le choix de la convention pour représenter les nombres entiers doit être utile pour les opérations arithmétiques simples, telles que l'addition par exemple. Dans le cas contraire, on serait obligé de faire plusieurs opérations avant d'obtenir le résultat d'une addition, ce qui rendrait un ordinateur peu performant ! Voyons de plus près comment un ordinateur additionne deux nombres entiers, en partant d'un exemple en base 10.

Addition de 18 et 31 : en décimal, 18 + 31 = 49, posons l'addition :

```
    18
+   31
-------
=   49
```

en binaire, on commence par convertir 18 et 31 :

$18_{(10)} = 10010_{(2)}$

$31_{(10)} = 11111_{(2)}$

il faut ici introduire la table arithmétique de l'addition en binaire, qui indique le résultat de l'addition de 2 bits :

0+0 = 0

1+0 = 1

0+1 = 1

1+1 = 0 avec une retenue de 1

```
   1111     pour noter les retenues successives
    10010
+   11111
-------
=  110001
```

que l'on retraduit en décimal : $110001_{(2)} = 49_{(10)}$: on obtient donc le bon résultat !

Les nombres entiers signés

Mais il n'est pas possible de travailler avec des nombres négatifs. En fait, nous venons de voir la représentation non signée. Comment représenter ces nombres négatifs ? un processeur ne connaît que des 0 et des 1, rien de plus, il faut donc indiquer comment traiter les nombres négatifs à partir de ces 0 et 1.

bit de signe simple :

Une solution pour cela est d'utiliser un bit qui représentera le signe du nombre. On choisit d'utiliser le bit de poids fort, que l'on dénommera *S*. le signe sera donné par la valeur $(-1)^S$. Soit un nombre p = xxx…xxx écrit en base binaire sur n bits en représentation non signée. En représentation signée avec bit de signe simple, les valeurs $+p$ et $-p$ s'écrivent donc de la manière suivante :

0 xxx…xxx est traduit en $(-1)^0.p$ donc représente le nombre $+p$

1 xxx…xxx est traduit en $(-1)^1.p$ donc représente le nombre $-p$

De même que pour la représentation non signée, la connaissance du nombre N de bits avec lesquels on représente les nombres au sein du processeur permet de calculer l'intervalle des nombres représentables. Le nombre négatif de plus grande valeur absolue est dans ce cas

$1\underbrace{11\ldots11}_{N\text{-}1\text{ bits}} = -(2^{N-1} - 1)$ (pour $N = 8$: -127, pour $N = 16$: -32767)

S

et le nombre positif de plus grande valeur absolue est $0\underbrace{11\ldots11}_{N\text{-}1\text{ bits}} = +(2^{N-1} - 1)$

S

Le choix d'utiliser le bit de poids fort d'un nombre comme bit de signe est une **convention** ! On aurait tout aussi bien pu choisir le bit de poids faible comme bit de signe, ou tout autre convention !

L'utilisation en pratique de cette représentation pose cependant quelques problèmes :

- Il existe 2 représentations pour le 0 :

S → 0 | 0…0 (+0)

S → 1 | 0…0 (-0)

- l'addition de deux nombres opposés donne un résultat pour le moins surprenant :

exemple : addition de +12 et –12, en prenant des nombres représentés sur 5 bits (signe inclus).

+12 est représenté par : $^{1}\ ^{1}0^{1}1100$

-12 est représenté par : 1 1100

somme : (1) 0 1000

La retenue propagée à un sixième bit est entre parenthèses car la représentation des nombres est faite sur 5 bits. Le résultat se lit donc sur ces 5 bits et vaut… +8 ! La représentation des nombres signés par bit de signe simple n'est donc pas consistante. Pour effectuer cette opération +12 –12, une phase d'interprétation des termes est nécessaire : si le bit de signe du deuxième terme est égal à 1, alors on doit effectuer une soustraction avec l'opposé de ce deuxième terme. En reprenant l'exemple donné ci-dessus, l'opération effectuée par le processeur sera : +12 – (+12) et non pas +12 +(-12).

La notation en complément à 2

Il existe fort heureusement une représentation consistante des nombres signés permettant d'obtenir des résultats cohérents lorsque l'on effectue des opérations avec des nombres signés, quelque soit leur signe. Il s'agit de la représentation dite en **complément à *b*** (complément à 10 en base 10, à 2 en base binaire).

Principe du complément à *b* : illustration en base 10

Soit un nombre *N*, représenté sur *n* chiffres. Le complément à 10 de *N* (noté $\check{N}$) est la valeur $10^n - N$. Attention, ce n'est pas parce que nous prenons l'exemple de la base 10 que le signe '-' ou l'existence de nombres négatifs est automatiquement garantie ! Imaginons plutôt que nous travaillons avec un ordinateur manipulant des informations élémentaires à 10 états (les *decits* au lieu des *bits* par exemple). Par convention, la représentation d'un nombre négatif -*N* sera donnée par le complément à 10 de *N* : $\check{N}$.

Cette représentation permet de s'affranchir des deux problèmes rencontrés lors de la représentation par bit de signe simple.

Avant de passer aux exemples, un dernier rappel : la représentation est toujours donnée **pour un nombre *n* de chiffres** (bits ou decits). Tout résultat occupant *n*+1 bits sera tronqué à *n* chiffres ! le chiffre *n*+1, s'il existe, sera noté en gris dans la suite du document.

- Complément de 0 : en base 10 : $10^n - 0 = 10^n$. exemple sur n=5 chiffres :

0 = 00000 ;

10^5-0 = 100000, tronqué à 5 chiffres : 00000 : Il n'y a donc qu'une seule représentation du nombre 0.

- Addition de deux nombres opposés. Soit un nombre *N* donné sur 5 chiffres, effectuons l'addition $N + \check{N}$: $N+(10^5-N) = 10^5 = 100000$, tronqué à 5 chiffres : 00000. Il n'y a donc pas besoin d'interpréter un éventuel bit de signe pour savoir si l'on doit effectuer une addition ou une soustraction.

Le complément à 2 en base binaire :

Il suffit d'appliquer le même principe, valable quelle que soit la base dans laquelle sont représentés les nombres. Soit un nombre *P* représenté sur *p* bits : son complément à 2 est $2^p - P$ (noté $-P$, attention à ne pas confondre le signe '-' de l'opération 2^p-*P* et le symbole de complémentation '-').

L'écriture de ce complément à 2 est de plus facile à réaliser d'après la formule de calcul de ce complément. $-P = 2^p - P = (2^p-1)+1-P = (2^p-1)-P+1$. Or 2^p-1 s'écrit comme un ensemble de p bits valant 1 :

$2^p-1 = 1\ldots1$

De plus, la soustraction d'un bit quelconque à 1 donne le complément de ce bit : en reprenant la table arithmétique de la soustraction binaire, on s'aperçoit que 1-0 = 1 et que 1 - 1=0.

Donc, le nombre $(2^p-1)-P$ s'obtient simplement en inversant les bits de P.

Il est facile d'en déduire que $-P$ s'obtient en inversant tous les bits de P et en ajoutant 1 à ce résultat. On pose, par convention, que les nombres positifs seront les nombres dont le bit de poids fort est égal à 0.

Complément à 2 d'un complément à 2 : soit le nombre P écrit en base binaire. $-(-P) = 2^p-(2^p-P) = P$.

Exemples d'opérations en complément à 2 en base binaire :

sur 5 bits, $N_1 = 5_{10} \Rightarrow 00101_2$ donc -5 = 11010+1 = 11011_2. Attention, il faut donc préciser que l'on travaille avec la convention du complément à 2, car $11011_2 = 27$ en représentation non signée !

$N_2 = 8 : 01000_2$, -8 = 10111+1 = 11000_2

N_1+N_2 : 00101 + 01000 = 01101 = +13

N_1-N_2 : 00101 + 11000 = 11101 = ?

$-N_1+N_2$: 11011 + 01000 = 100011 = +3

$-N_1-N_2$: 11011 + 11000 = 110011 = ?

Comment interpréter les résultats dont le bit de poids fort est égal à 1 ? Selon la convention de signe établie, il s'agit de nombres négatifs. Leur valeur absolue est donc donnée par leur complément à 2 ! Reprenons le résultat de N_1-N_2, qui est 11101 : le complément de ce nombre est 00010+1 = 00011 = 3, ce qui signifie que 11101 = -3, qui est

effectivement le résultat attendu de 5-8. De même 10011, dont la valeur absolue est 01100+1 = 01101 = 13. Ce nombre vaut donc –13.

Les nombres réels

Beaucoup de calculs effectués par des processeurs ou ordinateurs utilisent des données qui, si l'on utilise les représentations en nombre entier, seraient stockées de façon terriblement inefficaces. Les applications de physique, par exemple, travaillent couramment avec des puissances de 10 très faibles ou très élevées. Le nombre d'Avogadro ($6{,}022.10^{23}$) en est une bonne illustration. En notation entière, il vaut : $602200000000000000000000_{10}$. Un microprocesseur le stockera donc forcément en représentation binaire sous la forme : $01111100010000101010001101001101100001000100000000000000000000_2$, ce qui revient à utiliser 8 octets. Or, les très nombreux '0' de l'écriture en base 10 ne sont présents que pour donner la valeur de l'exposant. Les seules valeurs : **6,022** et **23** (l'exposant) permettent de stocker ce nombre sans perte d'information. Ce n'est pas parce que le nombre d'Avogadro s'écrit avec 24 chiffres que ces 24 chiffres sont significatifs ! Les difficultés sont même plus nombreuses si l'on s'attaque aux exposants négatifs. Comment représenter un nombre tel que $7{,}452.10^{-54}$?

Représentation en virgule fixe

De même que pour les nombres signés, tout est ici affaire de convention de codage. Pensez donc, un système qui ne connaît pas le simple signe '-' ! Comment peut-il s'occuper d'une virgule ? La première solution consiste à disposer d'une virgule dont l'emplacement est implicite et défini une bonne fois pour toutes. Par exemple, il est possible de convenir que tous les nombres à virgule contiendront un certain nombre de chiffres avant et après la virgule, ni plus, ni moins. On applique alors l'arithmétique classique sans se préoccuper de la position de cette virgule (sauf pour les multiplications et les divisions). Afin de prévoir les cas extrêmes, on peut décider d'avoir 30 chiffres (en base 10) avant la virgule, et 30 chiffres après (ce qui nous laisse une marge de 60 ordres de grandeur). Sachant que l'écriture binaire d'un nombre donné prend à peu près 3,5 fois plus de place que son écriture décimale (essayez de trouver comment on arrive à ce résultat d'ailleurs…), il faut donc la bagatelle de 210 bits, soit 26 octets pour représenter un nombre à virgule. Cette représentation présente deux inconvénients majeurs :

- Au niveau mathématique : nous avons déjà évoqué le fait qu'aucun nombre ne possède autant de chiffres significatifs en pratique ;

- Au niveau informatique : contrairement à ce que l'on pourrait penser, il ne s'agit absolument pas d'un problème de mémoire (une matrice 100 x 100 de tels nombres occuperait 256 ko 'seulement'), mais d'un problème de temps de traitement. En effet, aucun processeur à ce jour n'est capable de traiter des données de 210 bits en une seule fois. Il faudrait donc plusieurs accès à la mémoire (qui est très très lente par rapport au processeur) pour faire un simple calcul.

Représentation en virgule flottante, standard IEEE-754

Cette représentation utilisée pour les calculs scientifiques est basée sur la notation du même nom (scientifique). Un nombre *n* s'écrit sous la forme $n=f.10^e$. *f* est la **mantisse** et *e* l'**exposant**. Avant de détailler la représentation informatique de ces nombres, précisons une fois de plus qu'il s'agit d'une représentation discrète des nombres réels et que cela pose quelques problèmes. Il est donc importer de préciser quelques termes couramment utilisés en calcul scientifique (avec le terme anglais correspondant, noté en italique) :

Etendue (*range*) : l'étendue d'une représentation indique quels sont les plus petits et plus grands nombres représentables.

Précision (*precision*): la précision indique le nombre de chiffres significatifs d'un nombre.

Résolution *(accuracy)* : la résolution mesure quant à elle la validité de la valeur approchée d'un nombre réel par sa représentation informatique.

La précision est arbitrairement grande, et ne dépend que de la représentation choisie. La résolution peut, par contre, être inférieure à cette précision, notamment à cause de phénomènes d'arrondis ou de différences d'ordres de grandeur intervenant lors d'un calcul. Soit le calcul suivant effectué sur un ordinateur : $0{,}12345.10^5 + 0{,}56789.10^3$. Le deuxième terme est converti pour que son exposant soit également 10^5 : $0{,}0056789.10^5 \Rightarrow$ apparition de 2 chiffres non significatifs à droite. La précision de ce terme est de 7 chiffres, mais la résolution du résultat est donnée par celle du premier terme : 5 chiffres.

Prenons l'exemple d'une représentation $\mathbb{R}$ de nombres en virgule flottante (à l'opposé de la virgule fixe) comportant une mantisse signée sur 3 chiffres et un exposant signé sur 2 chiffres. L'ensemble des nombres représentables va donc de $-0{,}999.10^{99}$ à $-0{,}001.10^{-99}$ et de

$+0{,}001.10^{-99}$ à $0{,}999.10^{99}$, soit un intervalle de 199 ordres de grandeur, codés avec 5 chiffres décimaux et 2 signes.

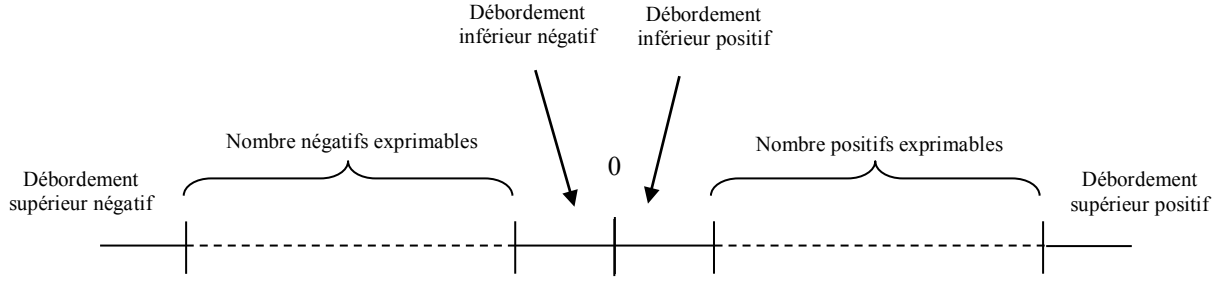
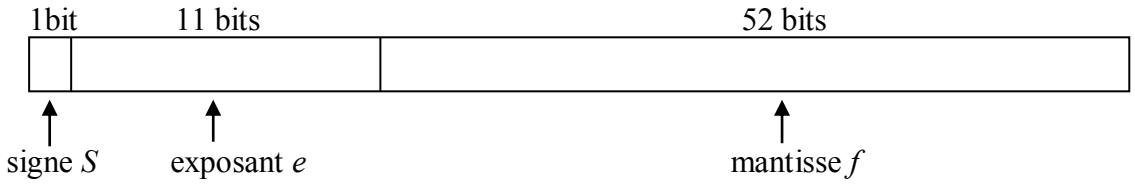


Figure 1 : Les nombres réels sont divisés en sept zones.

A l'aide de la représentation ainsi définie, il est possible de représenter 179100 nombres positifs dans la zone 'nombres positifs exprimables' de la Figure 1, autant de nombres négatifs dans la zone 'nombres négatifs exprimables' et le 0. En augmentant la taille de la mantisse, on augmente la densité des nombres et les approximations sont meilleures ; en augmentant la taille de l'exposant, on augmente l'étendue des zones des nombres exprimables.

Jusqu'au début des années 80, chaque ordinateur disposait de ses propres formats de représentation de nombres flottants. Pire encore, certains d'entre eux effectuaient des calculs incorrects tout simplement parce que leur arithmétique en virgule flottante présentait des subtilités peu évidentes pour un programmeur moyen. Pour mettre fin à cette situation, une association de professionnels des secteurs de l'électronique (l'IEEE) a décidé de définir un standard pour les calculs arithmétiques en nombres flottants. Les résultats de ces travaux ont conduit à la définition du standard IEEE-754 en 1985, dont voici le détail :

Il existe trois formats de représentation : en simple précision sur 32 bits (type `float` du C), en double précision sur 64 bits (type `double` du C) et en précision étendue sur 80 bits. Ce dernier format est surtout destiné à réduire les erreurs d'arrondi lors de calculs de précision. Nous allons détailler le format double précision (64 bits), mais le principe reste identique pour les deux autres formats.



Comme pour tous les nombres, le bit de poids fort est un bit de signe *S*. L'exposant *e* est codé en excédent à 1023 (voir formule ci-dessous). La valeur d'un nombre *x* représenté en double précision est donnée par la formule suivante :

$$x = (-1)^S.f.2^{(e-1023)} \text{ (et non pas } (-1)^S.f.10^e \text{ !)}$$

Une mantisse *f* est dite normalisée lorsque le premier bit de cette mantisse est à 1. Les auteurs du standard IEEE ont imposé que la mantisse soit systématiquement normalisée : ainsi le premier bit devrait obligatoirement valoir 1 : cela implique que l'on n'a pas besoin de le noter dans la mantisse, c'est un bit caché ou implicite. De plus ce standard dispose de cas particuliers permettant de représenter les quantités infinies ainsi que les résultats indéfinis d'opérations (infini divisé par infini par exemple), résumés dans le tableau ci-dessous.

Type	Exposant *e*	Mantisse *f*
normalisé	$0 < e < \max$	Quelconque
dénormalisé	Tous les bits à 0	au moins 1 bit à 1
zéro	Tous les bits à 0	tous les bits à 0
infini	Tous les bits à 1	tous les bits à 0
indéfini (NaN)	Tous les bits à 1	au moins un bit à 1

Table 1 : les divers formats des nombres du standard IEEE

En conclusion

Un ordinateur ne stocke que des valeurs binaires, et il faut lui indiquer quelle convention choisir lorsque l'on veut qu'il utilise une donnée (c'est à dire, une information). La convention choisie se matérialise par ce qu'on appelle le **type** de la donnée : en fonction de ce type, l'ordinateur sait comment manipuler cette donnée : comment faire des calculs, comment afficher la valeur stockée, etc…

Manipulations des données, premiers programmes

Les types de données

Puisqu'il est impératif de connaître le type des données, c'est à dire la convention à choisir pour les traduire, nous utiliserons un ensemble de types que l'on peut associer à une donnée. En fait, c'est même obligatoire !

Nous distinguerons trois types de base en algorithmique, et toute donnée utilisée aura forcément un de ces types :

Le type **entier** sera utilisé pour stocker des valeurs entières, positives ou négatives
Le type **réel** sera utilisé pour stocker les nombres à virgule
Le type **caractère** sera utilisé pour stocker les caractères.

Note sur le type **caractère** :

Le type utilisé pour stocker des caractères est un peu particulier, car un caractère est en fait un nombre entier ! l'ordinateur utilise en fait une table de correspondance qui associe une valeur entière à un caractère lorsqu'il s'agit de le manipuler, c'est à dire, la plupart du temps, de l'afficher à l'écran. Cette table de correspondance est la table des codes ASCII, que l'on trouve parfois sous cette forme : les colonnes grisées contiennent les valeurs entières, les colonnes non grisées contiennent les caractères imprimables. Par exemple, à la valeur entière 60 correspond le caractère '<'.

32	SP	44	,	56	8	68	D	80	P	92	\	104	h	116	t
33	!	45	-	57	9	69	E	81	Q	93	]	105	i	117	u
34	"	46	.	58	:	70	F	82	R	94	^	106	j	118	v
35	#	47	/	59	;	71	G	83	S	95	_	107	k	119	w
36	$	48	0	60	<	72	H	84	T	96	`	108	l	120	x
37	%	49	1	61	=	73	I	85	U	97	a	109	m	121	y
38	&	50	2	62	>	74	J	86	V	98	b	110	n	122	z
39	'	51	3	63	?	75	K	87	W	99	c	111	o	123	{

40	(	52	4	64	@	76	L	88	X	100	d	112	p	124	|
41	)	53	5	65	A	77	M	89	Y	101	e	113	q	125	}
42	*	54	6	66	B	78	N	90	Z	102	f	114	r	126	~
43	+	55	7	67	C	79	O	91	[	103	g	115	s	127	del

A chaque nombre entier correspond un caractère. La valeur stockée est donc le nombre entier, mais il peut apparaître sous la forme d'un caractère à l'écran.

Un caractère est stocké dans un octet (un groupement de 8 bits), la valeur entière peut donc aller de 0 à 255. Dans le tableau ne sont présentes que les valeurs de 32 à 127 : en deçà de 32, il s'agit de caractères non imprimables, au delà de 127, ce sont des caractères optionnels, qui sont adaptés à un type de clavier ou de langue particulier, notamment les caractères accentués (é, à, è, ô, â, etc…).

Les variables

Qu'est-ce qu'une variable ?

Une variable est une donnée qu'un programme peut manipuler. Tout variable possède :

un type (**entier**, **réel**, ou **caractère**);

un nom ou identificateur que l'utilisateur choisit; il permet au programme de reconnaître quelle donnée il doit manipuler.

une valeur qui peut évoluer au cours du programme, mais qui doit respecter le type. Une variable dont le type est **entier** ne pourra jamais contenir de valeur à virgule.

Comme nous l'avons vu précédemment, le type est choisi en fonction du rôle de la variable.

L'identificateur ou nom de la variable peut être quelconque, mais doit respecter les critères suivants :

- Il doit toujours commencer par une lettre de l'alphabet, en majuscule ou en minuscule;
- A part le premier caractère, il peut contenir : des lettres, des chiffres, et le symbole '_' (souligné ou underscore, la touche pour obtenir ce caractère est : 8).

- Il y a une différence entre les majuscules et les minuscules (c'est ce qu'on appelle la sensibilité à la casse) : les identificateurs 'toto' et 'Toto' ne sont donc pas les mêmes, par exemple.

On s'arrangera de plus, pour respecter les conventions suivantes, qui n'ont pas de caractère obligatoire :

- Un identificateur commence toujours par une lettre minuscule;
- Il doit porter avoir une relation avec le rôle de la variable et être compréhensible : une variable entière qui sert de compteur s'appellera plus volontiers 'compt' que 'xF_Gt3' par exemple.

Quelques éléments de syntaxe pour le langage algorithmique

Lorsque l'on écrit un programme en langage algorithmique, on doit, pour se repérer, donner un nom au programme et indiquer là où il débute : pour cela on utilisera le mot **programme** suivi du nom du programme, qui suit les mêmes règles de syntaxe que le nom d'une variable. On trouvera donc systématiquement au début d'un programme la ligne suivante :

```
Programme nom_du_prog
```

Où nom_du_prog sera remplacé par le nom que vous aurez choisi.

Définition et utilisation des variables

Avant d'utiliser une variable dans un programme, il faut la définir, c'est à dire indiquer : son type et son nom, dans cet ordre, suivi d'un point-virgule ';'. Ainsi, si l'on veut utiliser une variable s'appelant **taux**, et dont le type est **réel**, on écrira :

```
réel taux;
```

On dispose à cet instant d'une variable nommée **taux** qui peut stocker des nombres à virgule, et que l'on peut manipuler comme on le désire.

Des exemples de définition de variables :

`réel solution_equation;` définit une variable nommée *solution_equation* dont le type est *réel*;

`entier val_1;` définit une variable nommée *val_1* dont le type est *entier*;

entier compteur;

caractère lettre; définit une variable nommée *lettre* dont le type est *caractère*.

Définitions de plusieurs variables d'un même type : on peut définir plusieurs variables du même type en indiquant le type de ces variables puis la liste des noms de variables séparés par des virgules ','. La liste se termine par un point-virgule ';'.

Exemple : on veut définir trois variable de type entier nommées val_a, val_b et val_c. On peut écrire leur définition des deux manières suivantes :

```
entier val_a;
entier val_b;
entier val_c;
```

Ou

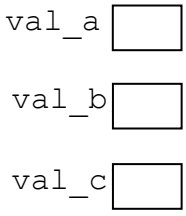
```
entier val_a, val_b, val_c;
```

les définitions de toutes les variables d'un programme doit se faire au début du programme, avant les différentes manipulations que l'on appelle des **instructions**.

Représentation des variables.

Pour aider à la compréhension d'un programme ou d'un algorithme, on peut être amené à utiliser des schémas (ou des tableaux) dans lesquels on notera la valeur des variables au cours du déroulement d'un programme. On peut symboliser une variable par une case (ou une boite) qui porte le nom de la variable et à l'intérieur de laquelle on note la valeur de cette variable.

Exemple pour les variables val_a, val_b et val_c :



Si l'on écrit : `val_a ← -6;` alors on peut le symboliser de la manière suivante :

Opérations et opérateurs de base

Affectation

La première opération que nous allons aborder est l'affectation : elle permet de donner (ou d'affecter, d'où son nom) une valeur à une variable. Sa syntaxe est la suivante :

Nom_de_variable ← valeur_à_affecter;

Le symbole ← indique le sens de l'affectation.

Les constantes

On peut affecter des valeurs numériques, appelées constantes, dans une variable. Selon le type de la variable, il y a une syntaxe à respecter.

Pour les nombres entiers, il suffit juste d'écrire la valeur en base 10, cette valeur peut être positive ou négative. La variable recevra alors la valeur choisie.

Exemple : on veut affecter la valeur 5 à une variable de type entier nommé **distance**. Ecrivons le programme correspondant : il faut nommer le programme, définir la variable distance, et ensuite l'affecter.

```
programme affectation
entier distance;
distance ← 5;
```

Les constantes pour les nombres à virgule : dans une variable de type réel, on peut bien évidemment également stocker des constantes, que l'on peut écrire sous la forme mantisse – exposant, c'est à dire la notation scientifique avec les puissances de 10 que l'on trouve sur les calculatrices. Attention, la virgule décimale se note suivant les conventions anglosaxonnes, c'est à dire par le point '.'. Une constante à virgule s'écrit donc sous la forme d'un nombre avec un point décimal pour positionner la virgule, suivant de E (E signifiant : 10 puissance), suivant de l'exposant, qui est un nombre positif ou négatif.

Exemples : `6.02E+23` signifie : $6{,}02.10^{+23}$.

`0.0721E-15` signifie : $0{,}0721.10^{-15}$.

Lors de la notation, on peut omettre certaines informations : l'exposant lorsque celui-ci est nul; ainsi on peut écrire la valeur approchée de la constante π avec : `3.14159`, sans ajouter `E+0`,

On n'est pas obligé de mettre le signe + devant l'exposant si celui-ci est positif : `1.7E8` signifie la même chose que `1.7E+8`

On n'est pas obligé de donner la partie décimale d'un nombre si celle-ci est nulle : on peut écrire `4.` A la place de `4.0`; par contre on fera toujours figurer le point décimal.

Exemple d'un programme affectant la valeur approchée de π à une variable nommée approx_pi :

```
Programme affect_pi
Réel approx_pi;
Approx_pi ← 3.141592;
```

Les constantes pour les caractères : puisqu'un caractère est aussi un nombre entier, on peut lui affecter une constante entière. Lorsqu'on l'affichera, l'ordinateur se servira de la table des codes ASCII pour savoir quel caractère afficher. Cependant, il est assez frustrant pour le programmeur de consulter cette table. Si l'on sait que l'on veut afficher, par exemple, la lettre **t** à l'écran, est-on obligé de regarder cette table pour savoir que c'est la constante 116 avec laquelle on doit initialiser la variable ? Il existe un moyen plus simple, qui est d'affecter la variable de type caractère avec le caractère à afficher, entouré de simples guillemets : 't' pour le caractère t, etc…

Exemple de programme qui affecte une variable nommé lettre, de type caractère, avec le caractère F.

```
Programme affect_carac
caractère lettre;
lettre ← 70;
lettre ←' F';
```

Ces deux instructions font exactement la même chose !

Opérateurs arithmétiques

A part affecter des constantes, il est intéressant de pouvoir effectuer des calculs. De plus, on n'est pas obligé d'affecter une constante à une variable, on peut aussi y affecter le résultat d'un calcul. Les opérateurs arithmétiques les plus classiques sont présents :

Addition :	+
Soustraction :	-
Multiplication :	*
Division :	/
Modulo :	%

Les 4 premiers opérateurs respectent les priorités arithmétiques connues, mais on peut aussi utiliser des parenthèses (et) pour forcer l'opération à se dérouler selon un ordre de priorité choisi.

Tous ces opérateurs sont dits binaires (ou 2-aires), non pas parce qu'ils s'appliquent à des valeurs binaires, mais parce qu'ils s'appliquent à 2 valeurs pour pouvoir calculer le résultat.

Avec ces opérateurs, les variables et les constantes, on écrit ce que l'on appelle des **expressions** : une expression est une suite d'opérateurs et de termes qui est compréhensible et que l'on peut calculer. Par exemple : (x+3)/2-(4-x)*7 est une expression, car on peut appliquer les opérations, mais 2)+)*5x8/(-9 n'est pas une expression, bien que tout ce qui la compose soit des opérations, des termes, et des parenthèses !

Retour sur l'affectation :

La règle, pour l'affectation est en fait la suivante :

Variable ← expression;

Et cela se passe toujours de la manière suivante :

1) l'ordinateur calcule l'expression fournie à droite de l'opérateur d'affectation;
2) cette valeur calculée est rangée dans la variable qui se trouve à gauche de l'opérateur d'affectation.

On peut ainsi écrire le programme suivant, où l'on fait intervenir plusieurs variables de types différents et plusieurs opérations de calcul :

```
programme operateurs

entier    a, b, c;
réel x_1, x_2;
caractère lettre;
```

```
a ← 5;
b ← a+2; // calcul de a(5) + 2 : 7, rangé dans la variable b
c ←4+(3*b)/(1+2*a); // calcul de 4+(3*7)/(1+2*5) = 4+21/11
a ← b % (c+1);
x_1 ← 3.1415926;
b ← c-4;
x_2 ← 2.0 * x_1 * 3.5;
x_1 ← x_1 + 1.0;
lettre ← 'X'+2;
lettre ← lettre - 'A';
lettre ← 0;
```

Ainsi, la ligne *b* ← *a*+2 ne signifie pas que la variable *b* contiendra toujours la valeurs stockée dans la variable *a* +2, mais qu'à cet instant du programme, on calcule la valeur *a*+2 (*a* vaut 5 et donc *a*+2 vaut 7) et qu'on la range dans la variable *b*. Il ne s'agit pas d'une équation, mais d'une affectation.

Opérateurs d'entrée/sortie

Les opérations que nous venons d'aborder permettent juste de faire des calculs, mais ne permettent pas encore de visualiser les résultats où d'afficher du texte, où encore de faire des saisies au clavier. Pour cela, nous utiliserons les instructions **AFFICHER** et **SAISIR**. Ce ne sont pas à proprement parler des opérateurs, mais nous les utiliserons comme tels.

AFFICHER sert, comme son nom l'indique, à afficher du texte ou les valeurs des variables.

Syntaxe d'utilisation :

- Pour afficher du texte :

On utilise AFFICHER, suivi entre parenthèses du texte à afficher qui est délimité par des double guillemets " .

Pour afficher les caractères 'bonjour' à l'écran, on écrira :

```
AFFICHER("bonjour");
```

Donne :

```
bonjour
```

- Pour afficher la valeur d'une variable :

On utilise AFFICHER, suivi entre parenthèses du nom de la variable à afficher. Attention à ne pas mettre de guillemets ! pour une variable nommée x_4, de type réel, on écrira `AFFICHER(x_4);`

Un exemple de programme :

```
programme affiche_des_valeurs;

entier a;
réel x;
caractère c;

a ← -7;
x ← 1.07E-5;
c ← '?';
afficher(a);          // -7 apparaît à l'écran
afficher(" a ");      // la lettre a apparaît à l'écran, entourée de
                      // deux espaces
afficher(x);          // 1.07E-5 apparaît à l'écran
afficher("c x ");     // les lettres c x apparaissent à l'écran,
                      //chacune suivie d'un espace, comme indiqué
                      // dans le texte
afficher(c);          // ? apparaît à l'écran
```

résultat :

```
-7 a 1.07E-5c x ?
```

- pour afficher plusieurs variables :

on peut, avec une seule commande afficher, faire apparaître à l'écran la valeur de plusieurs variables, il suffit pour cela de mette entre parenthèses la liste des variables séparées par une virgule ','.

- Pour afficher du texte et des variables :

on peut alterner, dans les parenthèses, du texte délimité par des double guillemets, et des variables, en les séparant par des virgules.

Exemple :

```
programme affiche_valeur_texte
entier age;
age ← 24;
```

```
afficher ("l'age stocke est ",age," ans");
```

Fait apparaître à l'écran :

```
l'age stocke est 24 ans
```

- Astuces de mise en page :

on utilise certains caractères dits 'spéciaux', ou encore 'non imprimables' (ce sont ceux qui sont dans la table de correspondance ASCII et qui portent un numéro inférieur à 32). Par exemple, pour effectuer un passage à la ligne, on utilise un caractère nommé '\n' (prononcer 'antislash n') que l'on peut mettre dans du texte.

Exemple :

```
afficher ("bonjour\ncomment ca va ?");
```

Fait apparaître à l'écran :

```
bonjour
comment ca va ?
```

pour afficher une tabulation, on utilisera le caractère '\t'.

exemple :

```
afficher ("hello\tles amis");
```

fait apparaître à l'écran :

```
hello       les amis
```

SAISIR permet d'initialiser une variable à partir d'une saisie faite au clavier. Contrairement à l'opérateur AFFICHER, on ne peut saisir que des variables avec SAISIR.

Syntaxe d'utilisation :

- Pour saisir la valeur d'une variable :

On utilise SAISIR, suivi entre parenthèses du nom de la variable que l'on veut saisir.

Exemple :

```
entier a;  // définition de la variable entière a

saisir(a); // saisie de la variable : l'utilisateur doit entrer une
           // valeur au clavier et valider par la touche 'entrée'
```

L'instruction SAISIR a pour seul effet d'attendre que l'utilisateur entre une valeur au clavier et la valide en appuyant sur la touche 'entrée' ou 'enter'; aucun message ne s'affiche

pour indiquer à l'utilisateur ce qu'il doit faire; c'est donc au programmeur de penser à afficher un message pour indiquer qu'une saisie doit être faite !

Le programme donné en exemple devrait donc comporter un affichage :

```
entier a;

afficher("entrez une valeur entière :");
saisir(a);
```

Contrairement à afficher, saisir ne permet pas de saisir plusieurs variables, on utilisera donc une instruction saisir par variable à saisir.

Exemple :

```
entier a;
réel val_x;

afficher("entrez une valeur entière puis une valeur à virgule :");
saisir(a);
saisir(val_x);
```

l'instruction saisir ne fait aucun contrôle sur ce que l'utilisateur entre effectivement au clavier : s'il faut saisir un entier et que l'utilisateur entre des caractères, le résultat obtenu n'est pas garanti ! Nous verrons plus tard comment ce contrôle de saisie peut être fait en langage C.

Les commentaires :

On peut insérer dans un programme des commentaires qui seront complètement ignorés par l'ordinateur mais qui sont utiles pour indiquer de manière plus claire ce que font les instructions. On indique que l'on écrit un commentaire en utilisant deux caractères / (slash) consécutifs. Sur une ligne, tout ce qui suit // est ignoré par l'ordinateur.

Exemples de synthèse, particularités des types

Pour terminer ce chapitre, nous allons revoir les différents éléments abordés au sein de différents exemples :

Opérations arithmétiques sur les nombres entiers : utilisation d'un tableau pour noter les valeurs des variables au cours du programme.

```
programme operations
entier a1, a2, b;
```

```
a1 ← 42;
b ← a1+8;
a2 ← 2 * a1;
a1 ← a2 / b;
afficher (b, a2, a1);
```

instructions	Valeur de a1	Valeur de a2	Valeur de b
a1 ← 42;	42	?	?
b ← a1+8;	42	?	50
a2 ← 2 * a1;	42	84	50
a1 ← a2 / b;	1	84	50
afficher(b,a2,a1)	50 84 1		

La division ne semble pas donner un résultat correct ! 84/50 = 1,68 d'après toute calculatrice digne de ce nom. Ce n'est pas une erreur ! Nous nous attendons à avoir un résultat à virgule, mais lorsque les deux valeurs (dividende et diviseur) sont des entiers, l'ordinateur fait une division entière (c'est à dire qu'il ne garde que la partie entière du résultat, sans faire d'arrondi). En fait, le calcul qui est fait correspond à : 84 = **1** x 50 + **34**, où **1** est le quotient et **34** est le reste.

Lorsque l'on divise deux entiers, on obtient un entier, même si la variable dans laquelle on range le résultat est un réel.

Affectations entre types différents.

Affecter une valeur de type réel à un entier

Il est possible d'affecter à une variable entière une expression réelle, par exemple, on peut écrire :

```
programme affect_reel;
entier t;
réel val_r;
val_r ← 3.94;
t ← val_r;
afficher(t);
```

```
3
```

sans que cela ne provoque d'erreur. il existe une règle que l'ordinateur utilise pour calculer la valeur qui va être affectée à la variable entière : **c'est la partie entière (sans arrondi) de la valeur réelle calculée**.

Dans l'exemple ci-dessus, t sera affectée avec la valeur 3.

Affecter une valeur de type entier à un réel

Lorsque l'on affecte une expression de type entier à une variable réelle, la conversion se fait naturellement : la valeur affectée est un nombre à virgule pour lequel la partie entière est égale à la valeur entière et la partie décimale est nulle.

Exemple :

```
programme affect_entier
entier c;
réel val_reel;
c ← -5;
val_reel ← c+1; // c+1 est de type entier et vaut -4
afficher(val_reel);
```

```
-4.0000000
```

un exemple classique : attention à la division ! (où l'on revoit le phénomène de la division entière). On fait apparaître le texte affiché ou les valeurs des variables dans un tableau à la suite du programme.

```
programme attention_a_la_division
reel resultat;
entier diviseur, divid;
afficher("entrez une valeur :");
saisir(divid);
diviseur ← 8;
resultat ← divid / diviseur;
afficher ("la division donne :",resultat);
```

on suppose que l'utilisateur entre la valeur 50 au clavier lors de l'instruction `saisir(divid);` et on sait bien que 50/8 = 6,25.

Instruction	divid	diviseur	resultat
`afficher("entrez une valeur :");`	`entrez une valeur :`		
`saisir(divid);`	50	?	?
`diviseur ← 8;`	50	8	?
`resultat ←divid / diviseur;`	50	8	6.00000
`afficher ("la division donne :", resultat);`	`La division donne : 6.00000`		

Et pourtant…le calcul donne 6.0000. Voyez-vous pourquoi ?

Explication :

On applique les règles de conversion et de traitement de la division entière que nous avons vues un peu plus haut :

Lorsque l'ordinateur exécute l'instruction : `resultat ← divid / diviseur;` il s'agit d'un affectation, donc il calcule dans un premier temps : `divid / diviseur`. Comme ces deux termes sont des entiers (car on a défini : `entier diviseur, divid;`), il effectue une division entière, c'est à dire que le résultat est tronqué : `divid/diviseur` vaut `6`, et ensuite on range la valeur `6` dans un réel (la variable `resultat`), elle est transformée en `6.0000`.

Comment faire pour que l'ordinateur fasse correctement une division pour obtenir un nombre à virgules ?

Il faut que l'un au moins des termes de la division soit de type **réel**, ce qui nécessite que les types des variables soient prévus en fonction des opérations que l'on veut leur appliquer. Ce dernier point n'est pas très satisfaisant car on choisit en général le type des variables seulement en fonction des valeurs qu'elles contiennent. Il existe heureusement une solution à cela.

Changer le type d'une expression

Il est possible de modifier le type d'une expression en la faisant précéder du nom de type voulu, entre parenthèses, avant l'expression choisie. Cette opération s'appelle le **transtypage**. Une valeur entière peut être considérée comme une valeur réelle si on la fait précéder de `(reel)` dans une expression.

Ainsi, pour le calcul de l'exemple précédent, on peut faire en sorte que les valeurs des variables `divid` et `diviseur` soient considérées comme des réels et non plus comme des entiers en leur appliquant cette opération.

```
resultat ← (reel)divid / (reel)diviseur;
```

`(reel)divid` signifie : calculer la valeur de divid (qui vaut 50) et la considérer comme un réel, donc `(reel)divid` vaut `50.000` .

Si l'on avait écrit `resultat ← (reel)(divid/diviseur);` le résultat obtenu aurait été 6.000 car le transtypage est effectué après la division, et non avant.

Permet d'obtenir le bon résultat, à savoir 6,25.

Lorsque le transtypage se fait d'un entier vers un réel, l'ordinateur se contente d'ajouter une partie décimale nulle, ce qui n'implique pas de perte de précision. Par contre, il est également possible de faier le transtypage en sens inverse : d'un réel vers un entier. Dans ce cas, la règle appliquée est celle de la division entière : la partie décimale est simplement tronquée (ce n'est pas un arrondi, mais une troncature), et il peut y avoir une perte de précision.

Exemple :

```
reel val_reelle, val_tronc;


val_reelle ← 2. * 3.1415926 * 1.54;
val_tronc ← (entier)val_reelle;
afficher(val_reelle," tronquee en :", val_tronc);
```

affiche :

```
9.676105 tronque en : 9.000000
```

la première fait le calcul désiré et range le résultat dans la variable `val_reelle`, ce qui est montré par l'affichage. la deuxième instruction calcule l'expression `(entier)val_reelle`, ce qui donne comme résultat : 9 (sans partie décimale, puisque le transtypage se fait vers une valeur entière).

Cette valeur, 9, est alors affectée à la variable réelle nommé `val_tronc`, et la conversion se fait naturellement en 9.000, valeur qui est donnée par l'affichage.

Le transtypage de ou vers le type caractère suit les mêmes règles que pour les valeurs entières, puisqu'un caractère est aussi un entier.

Attention : On ne peut pas changer le type d'une variable ! Lorsque l'on change le type par transtypage**, ces opérations de transtypages ne doivent se trouver que dans des expressions**, jamais devant des variables que l'on affecte !