Chapitre 3

Le langage C comme calculette... peu pratique

À l'origine le problème de la programmation consiste d'abord à faire effectuer les opérations élémentaires de façon automatique (c'est-à-dire sans qu'on ait à le faire comme à l'école élémentaire, car cela prend du temps). Il existe d'ailleurs des *machines d'aide aux calculs* avant l'apparition des ordinateurs, par exemple les bouliers et la machine à additionner de PASCAL. Vous savez que ces dernières machines sont de nos jours avantageusement remplacées par les **calculettes** (quatre opérations ou **scientifiques** ou même **alphanumériques** avec plus de possibilités ; nous oublierons les **calculettes programmables** qui sont de vrais ordinateurs, en plus petits). Du point de vue de la technologie, ces calculettes sont en fait une retombée de la conception des ordinateurs.

Un ordinateur peut faire la même chose qu'une calculette, et ceci quel que soit le langage de programmation choisi, et donc en langage C. Mais, comme nous nous en sommes aperçus à propos de notre premier exemple, il faut bien avouer que ceci est plus pénible à mettre en œuvre. Ce n'est pas grave ; comme nous le verrons dans les chapitres suivants, l'intérêt des ordinateurs ne vient pas de l'émulation des calculettes mais de la possibilité de combinaison d'opérations élémentaires déjà présentes sur les calculettes.

Nous allons étudier de façon critique dans ce chapitre l'analogue de la calculette, c'est-à-dire les opérations possibles. Vous n'êtes pas obligés de lire ce chapitre en entier avant d'aborder le suivant.

3.1 Premiers éléments de langage C

Reprenons le programme `essai.c` que nous avons vu lors de la mise en route du langage C :

```
#include <stdio.h>

void main(void)
  {
  printf("Bonjour");
  }
```

et commentons-le.

3.1.1 Forme d'un programme

Syntaxe.- Conformément à l'exemple que nous venons de rappeler, tout programme C doit au minimum être de la forme suivante :

```
#include <stdio.h>

void main(void)
  {
  INSTRUCTION
  }
```

Nous détaillerons au fur et à mesure ce qu'est une INSTRUCTION, le programme ci-dessus en donnant un exemple.

Premiers commentaires.- 1^o) Comme pour l'étude des langues naturelles, la **syntaxe** nous dit qu'elle est la forme d'un programme valide pour le langage de programmation donné. Un **programme valide** sera traduit en un programme objet par le compilateur ; sinon on aura une **erreur de syntaxe**.

2^o) Il n'y a pas de nom de programme. Un programme C est une suite de définitions de fonctions dont la fonction principale, dont le nom est le mot réservé **main**.

3^o) Il faut indiquer la nature des arguments et du résultat de cette fonction. Dans notre exemple nous avons **void** deux fois pour indiquer qu'il n'y a ni argument ni résultat.

4^o) Les accolades ouvrante '{' et fermante '}' indiquent le début et la fin d'un **bloc**.

5^o) L'instruction **printf** est une instruction d'affichage, plus précisément d'affichage formaté (d'où le **f**).

6^o) Les constantes mot (ici *Bonjour*) se placent entre guillemets verticaux " (le même symbole pour le guillemet ouvrant et le guillemet fermant).

7^o) Les constantes mot n'ont pas besoin d'être formatées, et qu'importe ce que cela veut dire pour l'instant, le langage C étant orienté manipulation des mots. C'est pourquoi notre premier exemple en C est l'affichage d'un texte et non pas un calcul numérique.

8^o) Le langage C étant lié au système d'exploitation UNIX, la différence entre les majuscules et les minuscules est importante. Par exemple **main** ne doit pas être écrit **Main**.

9^o) Toute instruction élémentaire en C se termine par un point-virgule.

10^o) Les compilateurs C purs existent de moins en moins et sont remplacés par des compilateurs C++. On peut leur demander de ne compiler que les programmes C et de respecter strictement leur syntaxe grâce à une option, mais ce n'est pas l'option par défaut.

Si on compile notre premier programme avec le compilateur `gcc`, par exemple, on voit apparaître un avertissement. On peut éviter celui-ci en réécrivant notre programme sous la forme suivante :

```
#include <stdio.h>

int main(void)
  {
  printf("Bonjour");
  return 0;
  }
```

Cela plaira au compilateur, mais avouons que c'est ridicule pour un premier programme.

Exercice 1.- 1^o) *Dans le programme précédent écrivez* **Main** *au lieu de* **main** *et faites-le tourner. Que se passe-t-il?*

2^o) *Essayez de même d'autres variations, par exemple en enlevant les* **void**, *le point-virgule ou la première ligne du programme.*

3.1.2 Fichiers inclus

Notion de fichier inclus.- La fonction **printf()** de notre premier programme est en fait une fonction *prédéfinie*. Elle se trouve dans un fichier spécial, dit de type **fichier inclus**. Ces fichiers inclus s'appellent aussi des **fichiers de définitions** ou des **fichiers en-tête** (*header file* en anglais).

Extension et répertoire spécifique.- Cette dernière dénomination explique l'extension habituelle **h** attribuée à ces fichiers, tel que le fichier `stdio.h`. Les fichiers inclus se trouvent normalement dans un répertoire nommé `include`.

Le chemin d'accès au répertoire *include* est, dans le cas d'*Unix*, en général `/usr/include`.

Les différents fichiers inclus sont organisés de manière thématique. Ainsi le fichier `math.h` contient les déclarations des fonctions mathématiques. Le fichier le plus utilisé est `stdio` (pour *standard input output*) qui renferme les déclarations des fonctions standard d'entrée-sortie, et donc entre autres la déclaration de la fonction d'affichage `printf()`.

La commande d'inclusion.- Pour indiquer au compilateur que l'on utilise un fichier inclus, le programme commence par la commande :

```
#include <nom_du_fichier>}
```

où `nom_du_fichier` est le nom du fichier inclus, par exemple `stdio.h`. Le compilateur sait où se trouve le répertoire `include` et cherche ce fichier dans celui-ci. Il y a évidemment un problème s'il ne s'y trouve pas.

Le préprocesseur.- La commande `#include` n'est pas une instruction du langage C proprement dit. C'est une commande du **préprocesseur**. Celui-ci est un programme qui modifie, avant la compilation, le texte source d'un programme. La commande principale de ce préprocesseur est

justement la commande `include`. Elle a pour effet d'ajouter le fichier `nom_du_fichier` au texte source du programme, et cela à l'emplacement de cette commande.

Bien que ces commandes puissent être placées n'importe où dans le programme, on les met habituellement au tout début de celui-ci.

Contrairement aux instructions élémentaires du langage C, les commandes du préprocesseur ne se terminent pas par un point-virgule.

Remarque.- Les fichiers inclus sont des portions de programme source et sont donc, en tant que tels, lisibles à l'aide de n'importe quel éditeur de texte. Ceci permet de voir quelles sont les fonctions disponibles dans tel ou tel fichier inclus.

Exercice.- *Trouver et lire le contenu de* `stdio.h`.

3.1.3 Compilation et édition des liens

Comme nous l'avons vu au chapitre précédent, le programme source est le contenu d'un fichier **nom.c**. La compilation proprement dite fournit un fichier, appelé **fichier objet**, de nom **nom.o** avec `cc`. Ce fichier contient le programme objet mais sans le code des fonctions prédéfinies. L'*éditeur de lien* (*linker* ou *binder* en anglais) intègre les codes machine du programme proprement dit et celui des fonctions prédéfinies, ce qui donne lieu à un **fichier exécutable**, de nom **a.out** avec `cc`.

3.1.4 Niveaux d'avertissement du compilateur

Les compilateurs C disposent en général de plusieurs niveaux d'avertissement. On décide de ceux que l'on veut voir apparaître grâce à un paramètre de celui-ci. Le niveau minimum distingue :

- les **erreurs de syntaxe** (*fatal error* en anglais) qui ne permettent pas de conduire à un programme exécutable ;
- des **avertissements** (*warning* en anglais) qui indiquent des sources possibles d'erreurs d'exécution.

3.2 Le langage C comme calculette quatre opérations

Nous allons indiquer comment se servir du langage C en tant que calculette, en commençant par la calculette quatre opérations. À chaque fois que nous définirons une instruction, nous préciserons sa **syntaxe** (c'est-à-dire comment il faut l'écrire) et sa **sémantique** (c'est-à-dire quel doit être le comportement de l'ordinateur devant l'instruction en question).

Plus exactement nous scinderons la sémantique en deux parties, à savoir la **sémantique attendue** (qui nous donnera, comme son nom l'indique, ce que l'on aimerait bien avoir) et la **sémantique réelle**, que l'on déduira de l'observation de notre système informatique considéré comme boîte noire. La différence entre la sémantique réelle et la sémantique attendue dépendra de problèmes d'implémentation ou, quelquefois, de limitations plus profondes de l'informatique (qu'il sera alors de notre devoir d'expliquer).

3.2.1 Cas des entiers naturels

Introduction.- L'ensemble $\mathbb{N}$ des entiers naturels est le premier ensemble de nombres que nous rencontrons, à l'école élémentaire. Il sert à dénombrer les ensembles finis. Nous étudions également les opérations sur cet ensemble, en particulier l'addition et la multiplication, pour simplifier le dénombrement (par exemple le cardinal de deux ensembles finis disjoints est la somme des cardinaux de ces ensembles). Reste à effectuer ces opérations. Nous avons appris des algorithmes pour les effectuer à la main à l'école élémentaire, ainsi que l'utilisation d'une calculette.

Un ordinateur peut remplacer une telle calculette.

Exemple.- Pour faire afficher le résultat de l'opération $1 + 1$ on peut utiliser, par exemple, le programme `Calc_1.c` suivant :

```
#include <stdio.h>

void main(void)
     {
     printf("%d",1 + 1);
     }
```

dont on peut vérifier qu'il donne bien ce qui est attendu.

Nous allons remplacer `1 + 1` de ce programme par d'autres expressions pour voir ce qui arrive.

3.2.1.1 Représentation et affichage des entiers naturels

Représentation.- Les entiers naturels se représentent, de façon habituelle, en base dix comme un mot sur l'alphabet **{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}** ne commençant pas par 0, sauf l'entier zéro qui se représente par 0.

Une telle représentation d'un entier naturel se notera `entier` (éventuellement suivi d'un nombre) dans la suite.

Syntaxe de l'affichage.- Une première manipulation sur les entiers consiste à les afficher (ou écrire, comme on veut) à l'écran. Ceci se fait grâce à l'instruction suivante :

```
printf("%d",entier);
```

où `entier` est un mot représentant un entier, comme nous venons de le dire.

Format.- Comme nous l'avons déjà dit le 'f' de `printf` signifie *format*. Il faut donc indiqué un **format** lors de l'utilisation de cette instruction, sauf éventuellement pour les chaînes de caractères (comme nous l'avons déjà vu à propos de l'affichage de *Bonjour* de notre premier exemple).

Une directive de format est indiquée par une chaîne de caractères précédée du symbole %.

Nous verrons les différents formats au fur et à mesure de nos besoins. Le format pour les entiers représentés en base dix est `%d` (avec *d* pour *décimal*).

Sémantique attendue.- Ce mot (représentation d'un entier) doit être recopié tel que à l'écran.

Exercice.- 1^o) *Faites afficher des entiers naturels en écrivant, par exemple* :

```
printf("%d",0);
printf("%d",137);
printf("%d",68283);
printf("%d",36482);
```

2^o) *Essayer quelques variations mineures autour de la syntaxe telle qu'elle est donnée* :

```
printf("%d",0137);
printf("%d",23 405);
```

et décrivez ce qui se passe.

3^o) *N'oublier pas de faire afficher de grands nombres tels que* :

```
printf("%d",64560982398405);
```

Que constatez-vous? Pouvez-vous expliquer les résultats?

3.2.1.2 Addition des entiers naturels

Syntaxe.- La mise en œuvre de l'addition se fait de façon naturelle en utilisant le symbole '+'. Nous l'avons d'ailleurs déjà vu à propos de notre premier exemple de calcul, celui de $1 + 1$.

Sémantique attendue.- Lorsqu'on utilise l'instruction :

```
printf("%d",entier1 + entier2);
```

on veut voir affiché à l'écran la somme de ces entiers.

Exercice.- 1^o) *Faites effectuer d'autres sommes. Commenter éventuellement.*

2^o) *N'oubliez pas des sommes faisant intervenir de grands entiers. Que constatez-vous? Pouvez-vous expliquer le résultat?*

3.2.1.3 Multiplication des entiers naturels

Syntaxe.- La mise en œuvre de la multiplication se fait comme pour l'addition sauf, qu'au lieu d'utiliser l'un des symboles traditionnels '.' ou '$\times$', on utilise le symbole '`*`'. Ce choix, traditionnel en informatique, est dû au fait que, d'une part, le point sert pour la représentation des réels et que, d'autre part, on peut aussi considérer des textes, comme nous l'avons vu, et donc que les symboles '.' et '$\times$' sont utilisés dans un autre contexte.

Exemple.- Un programme de multiplication se présentera donc sous la forme `Calc_2.c` suivante :

```
#include <stdio.h>

void main(void)
     {
     printf("%d",2*3);
     }
```

Vous remarquerez qu'il est traditionnel, contrairement à ce qui est fait pour l'addition, de ne pas laisser d'espace entre les facteurs et le signe de la multiplication.

Sémantique attendue.- Lorsqu'on utilise l'instruction :

```
printf("%d",entier1*entier2);
```

on veut voir affiché à l'écran le produit de ces entiers.

Exercice.- *Faites effectuer quelques produits. Commenter éventuellement.*

3.2.1.4 Soustraction et division

Remarque fondamentale.- La soustraction et la division ne sont pas des opérations totales sur l'ensemble $\mathbb{N}$ des entiers naturels. L'informatique ne peut pas faire de miracle, l'application des opérations '-' et '/', qui existent cependant, ne donneront donc pas nécessairement des entiers naturels.

Exercice.- *Faites effectuer quelques-unes de ces opérations, par exemple* :

$321 - 248$,
$374 - 1294$,
$4/2$,
$42/3$,
$421/5$,
$53/0$,
$392/32635$.

Commenter les résultats.

3.2.2 Cas des entiers relatifs

Représentation.- Les entiers relatifs se représentent de façon naturelle en base dix, c'est-à-dire zéro par 0, un entier relatif positif comme un entier naturel (éventuellement précédé du signe '+') et un entier relatif strictement négatif comme un entier naturel non nul précédé du signe '-'.

Format.- Le format est le même que pour les entiers naturels.

Exercice 1.- *Faites afficher quelques valeurs d'entiers relatifs et commentez.*

Syntaxe des opérations.- Les opérations habituelles (l'addition de symbole '+', la multiplication de symbole '*' et la soustraction de symbole '-') se mettent en œuvre comme dans le cas des entiers naturels.

Remarques.- 1^o) Ceci est rare pour un symbole dans un langage informatique, mais un même symbole, ici le symbole '-', a trois significations différentes (pour l'instant ; nous verrons en fait qu'il en a d'autres, car la soustraction pour les réels n'est pas la même chose que la soustraction pour les entiers, par exemple) : il sert à noter les entiers relatifs (pour indiquer leur signe), pour noter l'opération unaire d'opposition [par exemple $-(-2) = 2$] et pour noter l'opération binaire de soustraction.

2^o) La division (de symbole '/') existe mais ne donne pas le quotient (là encore l'informatique ne peut pas faire de miracle avec une opération partielle).

Exercice 2.- *Faites effectuer quelques opérations sur les entiers relatifs, en particulier les opérations suivantes* :

$23 - 7$,
$-32600 - 301$,
$(-423) * (45)$,
$(-38) * (-231)$.

Remarque.- On notera l'usage de parenthèses pour rendre certaines opérations non ambiguës.

3.2.3 Cas des rationnels

Après les ensembles $\mathbb{N}$ des entiers naturels et $\mathbb{Z}$ des entiers relatifs, l'ensemble $\mathbb{Q}$ des rationnels et l'ensemble $\mathbb{D}$ des nombres décimaux (ou nombres à virgule) ont été introduit au collège. Cependant il est traditionnel que les opérations sur les rationnels ne soient pas implémentées dans les langages de programmation, sans qu'il y ait de raisons profondes à cet état de fait.

3.2.4 Cas des réels

3.2.4.1 Sur les réels informatiques

Introduction.- Outre les entiers naturels, les entiers relatifs et les rationnels, les nombres réels ont été introduit en Mathématiques, à l'origine pour parler, par exemple de la longueur de la diagonale d'un carré de côté l'unité, puis pour de nombreuses autres applications. Les opérations habituelles d'addition, de multiplication, de soustraction et de division (par un réel non nul pour cette dernière opération) ont été introduites pour des raisons diverses. On aimerait bien pouvoir les traiter de façon automatique, comme dans le cas des entiers.

Cependant la seule façon essentielle d'entrer une donnée sur un ordinateur est de taper quelque chose sur un clavier, donc un mot fini sur un alphabet fini. La représentation habituelle des réels est un développement décimal illimité, par exemple $\sqrt{2} = 1,4142135...$ Le problème fondamental qui se pose alors pour travailler avec les réels est le suivant :

Existe-t-il une représentation des réels comme mots sur un alphabet ?

Réponse négative.- Vous n'avez pas vu de telle représentation jusqu'à présent, mais cela ne veut pas dire *a priori* qu'il n'en existe pas. Mais, malheureusement, cela est le cas. On montre qu'il n'existe pas de telle représentation (Cantor, 1874) en utilisant un argument de cardinalité (tout langage est au plus dénombrable alors que l'ensemble des réels est continupotent), comme on le voit dans le cours de Mathématiques consacré aux cardinaux infinis.

Réel (informatique) = décimal.- En fait, même si les nombres idéaux que sont les réels ont été introduits pour qu'on puisse parler de mesure exacte des longueurs, vous n'avez jamais su effectuer

d'opérations sur les réels, même à la main. On ne manipule que des *valeurs approchées* sous forme de *nombres décimaux* (qui sont en fait des nombres rationnels particuliers). C'est ce que l'on fera également en Informatique, en ne se préoccupant jamais des vrais réels (puisqu'on ne sait même pas les introduire) mais toujours de décimaux (en fait de nombres à virgule dans une certaine base).

Nous avons vu au collège les *décimaux*, ou *nombres à virgule*, puis les réels proprement dits. Tout nombre réel admet une *représentation décimale*, à savoir, pour un réel non nul, un signe, suivi d'un nombre fini de chiffres de 0 à 9, suivi d'une virgule, suivi de chiffres de 0 à 9 (en nombre fini ou infini). Cette représentation est presque unique : elle l'est si on exige que le mot formé des caractères se trouvant avant la virgule soit un entier (relatif) et que la représentation ne se termine pas par une infinité de 9 [par exemple en écrivant $0,13$ et non $0,1299999999...$]. Nous avons déjà dit qu'on ne peut pas manipuler les réels sur un ordinateur et que nous nous restreindrons aux décimaux, c'est-à-dire aux nombres réels ayant un nombre fini de chiffres non nuls après la virgule.

Remarquons que pour les décimaux il n'y aucune raison de considérer des valeurs approchées *a priori*. En conclusion rappelons-nous que :

réel (informatique) = décimal

3.2.4.2 Représentations des réels

Le second problème qui se pose à propos des réels est celui de leur représentation (celle des réels informatiques, mais nous ne le préciserons plus à partir de maintenant).

Représentation décimale

Syntaxe.- La représentation habituelle des décimaux a son analogue en langage C à une restriction près, à savoir que la *virgule décimale* (française) est remplacée par le **point décimal** (anglo-saxon). Tout décimal non nul est donc représenté par un signe (éventuellement), suivi d'un entier naturel, suivi d'un point, suivi d'un nombre fini de chiffres de 0 à 9.

Format.- Le format pour cette représentation est `%f` (pour *représentation à virgule* **f***lottante*). On écrira, par exemple :

```
printf("%f",3.14159);
```

pour afficher une valeur approchée du nombre π.

Exercice 1.- *Faites afficher des réels en écrivant, par exemple* 3.24, -3.24, 328.45, 32800.45, 0.123456789, 2398.129765432245 *et* 0.00000123986422456.

Que constatez-vous ? Pouvez-vous expliquer les résultats ?

Remarque.- Il est très important de respecter le format correspondant.

Exercice 2.- *Faites exécuter l'instruction dans un programme* `calc_3.c` :

```
printf("%d",3.14159);
```

et expliquer le résultat obtenu.

Représentation scientifique

Représentation avec mantisse et exposant.- Une représentation des réels utilisant les puissances de dix est souvent utilisée en Physique, par exemple :

$$32,456 \times 10^{-5}.$$

Dans ce cas le nombre devant la puissance (avec son signe éventuel) est appelé la **mantisse**, -5 est l'**exposant**. Ceci conduit à une représentation dite **scientifique**. Comme, d'une part, on ne peut écrire que linéairement, et non en utilisant des exposants, et que, d'autre part, la base dix est sous-entendue une fois pour toute, on écrit simplement le symbole 'E' ou 'e' devant l'exposant. Par exemple le réel ci-dessus s'écrira :

```
32.456 E -5
```

ou

```
32.456 e -5.
```

La *représentation scientifique* d'un réel est donc un entier relatif suivi d'un point suivi d'un mot composé de chiffres suivi de 'E' (ou de 'e') suivi d'un entier relatif.

Remarque.- Entre le point et 'E' on a un mot quelconque composé de chiffres et non un entier naturel, puisque ce mot peut commencer par autant de zéros que l'on veut.

Format.- Le format pour cette représentation est `%E` (ou `%e`) suivant que l'on veut voir afficher 'E' (ou 'e'). On écrira, par exemple :

```
printf("%e",.314159e1);
```

Exercice 3.- *Faites afficher quelques réels. Que constatez-vous?*

Remarque.- L'affichage **normalise** souvent le résultat, c'est-à-dire que la mantisse débute par un zéro, le point décimal et un chiffre non nul. Bien entendu l'exposant est calculé pour que ce soit bien le bon réel qui soit affiché.

Format G.- Il existe en fait un troisième format pour les réels : le format `%g` et `%G`. Suivant la précision demandée, la forme décimale ou scientifique est utilisée, en fait la plus courte des deux. On a un 'e' ou un 'E' (si c'est la représentation scientifique qui semble la plus adéquate au système) suivant que l'on a choisi %g ou %G.

3.2.4.3 Premières opérations

Mise en œuvre.- Comme pour les entiers, la mise en œuvre est facile : pour additionner, soustraire ou multiplier il suffit d'utiliser les signes respectifs '+', '-' et '*' et des réels sous l'une des deux représentations permises (décimale ou scientifique).

Exercice.- *Faites effectuer de telles opérations, en n'oubliant pas les cas limites que vous pouvez imaginer, et commenter les résultats. On considèrera, entre autres, les opérations suivantes* :

$3.23 + 4.52$,
$32600.0 + 3201.0$,
$3.4 + 2$,
$-543.673 + 43.56788$,

$5.6782E4 + 45.0$,
$5.67E45 + 0.123$,

puis recommencer en remplaçant '+' *par* '-' *puis par* '$*$'.

3.2.4.4 Division

Introduction.- Rappelons qu'en informatique *réel* désigne en fait un décimal, or la division n'est pas une opération totale sur les décimaux puisque, par exemple :

$$\frac{1}{3} = 0,333333...$$

avec une infinité de chiffres après la virgule.

La division existe cependant (en utilisant le symbole classique '/') mais le résultat est un décimal qui est une **valeur approchée**. La précision dépend de l'implémentation du langage et on ne peut pas la dominer.

Exercice.- *Faites effectuer quelques divisions.*

3.3 Le langage C comme petite calculette scientifique

Nous venons de voir que le langage C peut jouer le rôle de calculette quatre opérations (bien que plus difficile à mettre en œuvre qu'une vraie calculette). Nous allons voir qu'il peut aussi jouer le rôle d'une petite calculette scientifique.

3.3.1 Utilisation de la bibliothèque mathématique

Il faut utiliser la bibliothèque mathématique et donc utiliser le fichier inclus correspondant. Tous les programmes suivants commenceront donc par :

```
#include <math.h>
```

Dans ce cas, lors de la compilation avec un vrai compilateur C il faut indiquer *explicitement* d'utiliser la bibliothèque mathématique :

```
# cc -lm nom.c
```

mais ceci n'est plus nécessaire avec les compilateurs actuels.

3.3.2 Nouvelles opérations sur les entiers

Deux autres opérations sont implémentées en langage C pour les entiers (relatifs) : la valeur absolue et la division euclidienne. Par contre, et cela est volontaire, l'exponentiation n'est pas implémentée.

3.3.2.1 Valeur absolue

Syntaxe.- La syntaxe de la valeur absolue, représentée par `abs(.)`, qui est une opération unaire, c'est-à-dire une application, se comprend sur l'exemple suivant :

```
printf("%d",abs(-3));
```

Exercice.- *Faites les exercices habituels pour tester cette opération.*

3.3.2.2 Division euclidienne

Introduction.- La division n'est une opération totale ni sur $\mathbb{N}$ ni sur $\mathbb{Z}$. Rappelons cependant qu'il existe une autre sorte de division, la *division euclidienne*, qui est une opération totale avec deux résultats. Pour un entier a et un entier non nul b, il existe un unique couple (q, r) d'entiers tels que :

$$a = b.q + r \ et \ 0 \leq r < |b|.$$

L'entier q est appelé le *quotient* (*exact*) et l'entier r le *reste* dans la division euclidienne de a par b.

Syntaxe.- Cette opération existe en C, en fait sous la forme de deux opérations : une pour obtenir le quotient et une pour obtenir le reste. Le symbole pour obtenir q est « / », celui pour obtenir r est « % ». Ainsi :

```
22 / 7 = 3, 22 % 7 = 1.
```

Exercice.- *Faites effectuer quelques opérations dont l'opérande est / ou %, par exemple* $-23/7$, -23 % 7 *et* $23/-7$. *Commenter éventuellement les résultats.*

3.3.3 Formatage de l'affichage des réels

Introduction.- Il faut bien dire que l'affichage *par défaut* des réels, sous forme scientifique, n'est pas très élégant. On peut heureusement obliger l'affichage à être sous une forme voulue, en indiquant le nombre de chiffres désiré ainsi que le nombre de chiffres après la virgule (l'exposant étant alors recalculé pour qu'il s'agisse bien du même nombre).

Syntaxe.- La syntaxe du format est la suivante. Au lieu de `%f` on indique le format :

%Entier1.Entier2f

où `Entier1` et `Entier2` sont des entiers naturels. On peut aussi l'utiliser avec %e, %E, %g et %G.

Sémantique.- `Entier1` indique le nombre de chiffres total à afficher et `Entier2` le nombre de chiffres à afficher après la virgule (c'est donc un entier inférieur à `Entier1`). Si nécessaire les valeurs sont arrondies ou les chiffres après la virgule sont complétés par des 0.

Attention ! `Entier1` représente bien le nombre total de chiffres et non le nombre de chiffres avant la virgule, comme la syntaxe du format pourrait le laisser entendre.

Exemple.- *Essayer l'instruction suivante* :

```
printf("%9.8f",23.0/30);
```

Exercice.- *Faites afficher d'autres réels formatés pour explorer cette possibilité. N'oublier pas les cas limites (maintenant vous devriez être habitués), par exemple affichage de zéro, cas où* `Entier2` *est plus grand que* `Entier1`...

3.3.4 Fonctions prédéfinies sur les réels

Introduction.- Vous connaissez un certain nombre de fonctions de $\mathbb{R}$ dans $\mathbb{R}$, par exemple les fonctions trigonométriques (sinus, cosinus, tangente, cotangente) et les fonctions logarithmiques (exponentielle, logarithme népérien, logarithme décimal). Il existe d'autres fonctions, comme les fonctions trigonométriques réciproques (arc sinus, arc cosinus, arc tangente) et les fonctions de trigonométrie hyperbolique (sinus hyperbolique, cosinus hyperbolique, tangente hyperbolique, cotangente hyperbolique, argument sinus hyperbolique, argument cosinus hyperbolique, argument tangente hyperbolique). Il faut pouvoir manipuler ces fonctions, en particulier les calculer de façon approchée.

Problème de l'implémentation.- Il n'est pas question d'implémenter ces fonctions sur un ordinateur de façon exacte, puisque la valeur d'une telle fonction, même pour un argument décimal, n'est pas nécessairement un décimal. Mais il est intéressant d'en obtenir une **valeur approchée**.

Vous ne savez pas nécessairement comment on calcule une valeur approchée, à tant près que l'on veut, de la valeur d'une telle fonction en un point donné ; ceci est un des objets de l'*analyse numérique*. Mais vous savez qu'en consultant des tables, ou en manipulant quelque peu votre calculette scientifique, vous obtenez des résultats, qu'en général vous acceptez comme valables sans discussions.

Dans un langage de programmation tel que le langage C, le calcul approché (**avec une précision non dominée**) d'un certain nombre de ces fonctions est implémenté, tout au moins en faisant appel au fichier inclus `math.h`.

Liste de quelques fonctions.- Dans le tableau ci-dessous, l'écriture de ce qui est dans la colonne de gauche permet de calculer une valeur approchée de ce qui est indiqué dans la colonne de droite :

Fonction	Signification
fabs(x)	valeur absolue de x
acos(x)	arc cosinus de x,
asin(x)	arc sinus de x,
atan(x)	arc tangente de x,
cos(x)	cosinus de x,
sin(x)	sinus de x,
tan(x)	tangente de x,
exp(x)	exponentielle de x,
log(x)	logarithme (népérien) de x,
log10(x)	logarithme décimal de x,
pow(x,y)	x à la puissance y, pour des réels,
cosh(x)	cosinus hyperbolique de x,
sinh(x)	sinus hyperbolique de x,
tanh(x)	tangente hyperbolique de x,
ceil(x)	plus petit entier supérieur ou égal à x, considéré comme réel,
floor(x)	partie entière de x, considérée comme un réel,
sqrt(x)	racine carrée de x (pour *squareroot*).

Exercice 1.- *Certaines de ces fonctions peuvent donner des valeurs exactes, c'est-à-dire que l'image d'un décimal est un décimal. Indiquer lesquelles?*

Exercice 2.- *Explorer ces fonctions, en calculant par exemple* :

`ln 2`,
`e` (la base des logarithmes népériens),
π,
`tan 3`,
`sin(`π`/6)`,
`ln(-2)`,
$(\sqrt{2})^2$.

Remarque.- Il y a moins de fonctions implémentées en langage C que sur une calculette scientifique. Une raison est que l'on peut programmer d'autres fonctions (en fait toutes celles qui sont calculables).

3.3.5 Partie entière de $\mathbb{R}$ dans $\mathbb{N}$

On aura pu remarquer que, parmi les fonctions prédéfinies, s'il existe bien une fonction partie entière dont le résultat est un réel, nous n'avons pas donné de fonction partie entière dont le résultat est un entier. Il existe en langage C une façon de convertir des entités de types cohérents d'un type à l'autre que nous verrons dans le chapitre suivant, qui permettra de résoudre le problème.

3.4 Le langage C comme calculette améliorée

Nous venons de voir que le langage C peut jouer le rôle d'une petite calculette scientifique, donnant peu de chiffres significatifs et n'ayant qu'un jeu restreint d'opérations. Nous avons déjà dit que c'est la possibilité de combiner ces instructions élémentaires qui donnera toute la puissance d'un tel langage. Mais, avant même d'aborder ce point de vue (dans les chapitres suivants), nous allons voir une première amélioration par rapport aux calculettes scientifiques, à savoir la composition d'opérations et la manipulation d'objets autres que des nombres. Ceci correspond à ce qu'on appelle les **calculettes alphanumériques**.

3.4.1 Évaluation d'une expression numérique

Introduction.- Là où l'ordinateur, à travers un langage tel que le langage C, commence à se démarquer d'une calculette (scientifique simple, c'est-à-dire non alphanumérique) est qu'il peut effectuer des calculs complexes, sachant travailler sur des expressions numériques.

Exercice.- *Calculer une valeur approchée de* :

$$\frac{sin^2 3 + \frac{e^2}{cos(3,5)}}{ln\, 3 + cos^3(2,2)}.$$

[Il suffit de réécrire pratiquement la même chose mais *linéairement*, à savoir :

(pow(sin(3),2) + (exp(2)/cos(3.5)))/(log(3) + pow(cos(2.2),3)).]

Remarques.- 1°) Le problème peut se poser alors de savoir ce qu'est une **expression numérique**. Rappelons-nous que nous sommes dans une étude exploratoire, à la fois de la programmation et du langage C. On peut donner cependant, après avoir vu quelques notions préliminaires sur les *langages formels*, une définition précise de la syntaxe du langage C et, en particulier, de ce qu'est une *expression numérique* (en langage C). L'important ici est de prendre conscience de l'intérêt de cette notion (déjà vue au collège d'ailleurs) et de s'en servir intuitivement.

2°) La contrainte qui nous oblige à écrire linéairement est un héritage des premiers ordinateurs et de leurs périphériques d'entrée rudimentaires. On pourrait de nos jours entrer comme lorsqu'on écrit au tableau, ou plus exactement que l'affichage d'écho à l'écran soit fait ainsi. C'est d'ailleurs ce qui se passe pour les logiciels de calcul formel tels que *Mathematica* ou *Mapple V*.

Conclusion.- Nous venons de voir enfin un premier point sur lequel un langage de programmation (comme le langage C) se démarque des calculettes. Il faut bien avouer que la plupart des calculettes scientifiques peuvent réaliser ceci à coup de couples de parenthèses, mais la lecture en est nettement moins lisible et les risques d'erreurs beaucoup plus grands. On peut même écrire sur certaines calculettes une *expression* mais, d'une part, la taille de l'écran est plus petite et, d'autre part, il s'agit souvent de calculettes programmables (donc en fait de vrais petits ordinateurs).

3.4.2 Manipulation de textes

Introduction.- Une autre amélioration du langage C par rapport aux calculettes est qu'il peut manipuler des textes et plus particulièrement, dans une première étape, les afficher (à l'écran). Nous avons déjà vu comment afficher une constante texte (à savoir *Bonjour* dans notre premier exemple de programme en langage C).

Symboles spéciaux.- On écrit un texte, tel qu'on veut le voir affiché, entre guillemets verticaux, sauf pour les quatre symboles suivants :

l'apostrophe	'	doit être écrite	\' ,
le guillemet (vertical)	"	doit être écrit	\" ,
la contre-oblique	\	doit être écrite	\\ ,
le signe de pourcentage	%	doit être doublé	%% ,

pour les raisons que nous allons voir, à savoir que ces symboles simples servent déjà à autre chose.

La seule contrainte est que le texte doit être entièrement écrit sur une ligne, le guillemet vertical fermant doit être sur la même ligne que le guillemet vertical ouvrant.

Séquences d'échappement.- Pour formater le texte, on utilise les **séquences d'échappement** suivantes :

- \n permet d'aller à la ligne (d'après l'anglais *newline*),
- \t pose une tabulation (horizontale),
- \b permet le retour d'un caractère en arrière (d'après l'anglais *backspace*),
- \r provoque un retour chariot sans aller à la ligne (d'après l'anglais *carriage return*),
- \f provoque un saut de page (d'après l'anglais *form feed*),
- \a déclenche un signal sonore (d'après l'anglais *alarm*).

Exercice.- *Explorer cette possibilité d'afficher des textes, par exemple grâce au programme contenant la ligne suivante* :

```
printf("Essai de \n formatage de texte \a \t pour voir.");
```

sans oublier les cas limites tels qu'un texte écrit sur plusieurs lignes.

Remarque.- Il faut bien faire attention au fait que les espaces dans un texte ne sont pas anodins. Ils sont le résultat de la juxtaposition d'un certain nombre d'un caractère, au même titre que 'A' ou 'B', appelé **blanc** ou *espace*.

3.5 Sémantique réelle

Comme nous l'avions annoncé en introduction à la section 2, et comme vous vous en êtes aperçus en faisant les exercices proposés, la réaction de l'ordinateur n'est pas tout à fait la même que celle qui est attendue. Nous allons donc préciser la sémantique réelle ici et expliquer pourquoi une telle implémentation du langage C a été choisie.

3.5.1 Cas des entiers naturels

Explication 1.- (**Affichage**) Il existe un entier naturel, disons `MAX`, dépendant du compilateur, tel qu'un programme ne puisse afficher que les entiers naturels compris entre 0 et `MAX` inclus. Aucune erreur n'est signalée si l'entier n'est pas compris entre ces deux nombres, un entier relatif est affiché à la place suivant une règle que nous verrons ci-dessous.

Exercice 1.- *Essayer de déterminer la valeur de* `MAX` *de votre compilateur en faisant afficher plusieurs entiers.*

Explication 2.- L'explication pour les résultats farfelus de certains affichages tient à ce que nous venons de voir. Le reproche que l'on peut faire à cette façon d'implémenter un langage est que nous ne sommes pas prévenus du **dépassement de capacité**, c'est-à-dire lorsque le résultat du calcul dépasse `Max`.

Perspective.- Que faire lorsqu'on a besoin de calculs sur de plus grands entiers que `Max`, ou plus exactement dont le résultat est plus grand que `Max` ? sur des entiers arbitrairement longs ? Vous savez peut-être que ce dernier cas est réalisable par des publicités pour des logiciels de calcul formel tels que *Mathematica* ou *Mapple V*. Il s'agira de programmer, et nous verrons alors toute la puissance de la combinaison des instructions élémentaires par rapport à une simple calculette.

Explication 3.- (**Addition et multiplication**) Tout se passe comme si ces opérations étaient effectuées correctement mais, en cas de dépassement de capacité, l'affichage donne un résultat non correct. De plus rien n'avertit de ce dépassement de capacité.

Explication 4.- (**Soustraction**) Les langages de programmation considèrent, de façon naturelle, que le résultat d'une soustraction de deux entiers naturels est un entier relatif, élément de $\mathbb{Z}$. Il faut donc étudier la représentation des éléments de cet ensemble pour pouvoir expliquer plus en détail ce qui se passe lorsqu'on effectue une soustraction.

Explication 5.- (**Division**) Beaucoup de langages de programmation considèrent que le résultat de la division d'un entier naturel par un entier naturel est un réel, élément de $\mathbb{R}$. Ce n'est cependant pas le cas du langage C qui utilise le symbole de division pour une autre opération, la division euclidienne.

Remarques.- 1^o) La division (euclidienne) par zéro n'est pas définie, ce qui est tout à fait normal. Mais ceci est également vrai de la division par certains entiers liés à `MAX`, puisqu'ils se comportent exactement comme zéro.

2^o) Un moyen de faire des calculs exacts avec des entiers un peu plus grands est de les considérer comme des réels, c'est-à-dire de les faire suivre d'un point décimal et d'un zéro.

3.5.2 Cas des entiers relatifs

On trouve une restriction analogue à celle rencontrée pour les entiers naturels.

Explication 1.- (**Affichage**) L'affichage est exact modulo `2.(MAX + 1)`, avec le représentant compris entre $-MAX-1$ et MAX.

C'est peut-être le moment de revoir votre cours de mathématiques sur les congruences modulo un entier naturel non nul, c'est-à-dire sur l'arithmétique modulaire.

Exercice.- *Pouvez-vous expliquer les valeurs souvent rencontrées lors des implémentations de* $MAX = 32\,767$?

Explication 2.- (**Opération**) Les résultats des opérations seront affichés avec des valeurs comprises entre $-MAX-1$ et MAX. Nous pouvons vérifier que les opérations sont effectuées de façon exacte, mais que le résultat est donné modulo quelque chose. On voit pourquoi dans le cours d'architecture. En fait il n'y a pas deux opérations (calcul exact puis calcul de l'affichage) mais une seule dont le comportement revient à cela.

3.5.3 Cas des réels

Affichage.- 1^o) L'affichage par défaut est la représentation scientifique.

2^o) Un certain nombre (prédéterminé) de chiffres (le plus souvent onze) sont affichés au plus pour la mantisse. Si la mantisse a plus de onze chiffres alors le réel est **tronqué** (c'est-à-dire que seuls les onze premiers chiffres, en commençant par un chiffre non nul, sont conservés). Cette valeur (de onze) dépend évidemment de l'implémentation.

3^o) L'exposant doit être compris entre deux entiers donnés, le plus souvent -39 et $+39$. Si l'exposant est en-dehors de l'intervalle permis (dépendant, là encore, de l'implémentation) alors un message d'erreur de *dépassement de capacité* est indiqué : *underflow* ou *overflow* suivant le cas.

Exercice.- *On voit souvent écrit dans les manuels, après avoir expliqué ce qui est ci-dessus, quelque chose comme :* « *ainsi on peut représenter tous les réels positifs compris entre* 10^{-39} *et* 99999.10^{39} ». *Est-ce exact* ?

Opérations.- Les opérations sur les réels semblent correctement effectuées (vérifiez éventuellement) mais l'affichage se fait suivant les règles vues ci-dessus. On peut en particulier avoir un tronquage du résultat ou un dépassement de capacité.

3.5.4 Exemple commenté

Si on fait exécuter le programme suivant :

```
#include <stdio.h>

void main(void)
   {
   printf("%d\n'', 12345678901);
   }
```

on obtient :

```
E:\>gcc semantique.c -o semantique.exe

E:\>semantique
-539222987

E:\>
```

Cela peut sembler surprenant. En plus le nombre affiché est négatif.

Ceci est dû au fait que la sémantique réelle des entiers (naturels ou relatifs) sur un microprocesseur n'est pas celle de $\mathbb{N}$ mais l'*arithmétique modulaire* de $\mathbb{Z}/N.\mathbb{Z}$, avec N un entier assez grand.

Tout le monde connaît l'arithmétique modulaire modulo 12 puisqu'elle est utilisée sur les horloges analogiques, ainsi que sur les horloges digitales (en concurrence avec l'arithmétique modulaire modulo 24 sur celles-ci) : s'il est 9 h et que le cours dure 4 heures, il terminera à 13 h en France, sans surprise, mais indiqué 1 h sur l'horloge digitale. Autrement dit, lorsqu'on compte on passe de 1 à 2, de 2 à 3, ..., de 11 à 12, puis de 12 à 1 et ainsi de suite en recommençant à partir de 1.

Plus généralement, en arithmétique modulaire modulo N, on commence en général à 0, pour égrener les nombres 1, 2, ..., $N-2$, $N-1$, puis à nouveau 0, 1, ...

On implémente une arithmétique modulaire sur les microprocesseurs puisque les *registres* de ceux-ci ne peuvent contenir qu'un entier compris entre 0 et une valeur MAX, dû au nombre limité de *bits* qu'ils peuvent enregistrer.

Un *vrai* microprocesseur n bits manipule les entiers compris entre 0 et $2^n - 1$.

Vous avez certainement un microprocesseur vendu fièrement comme 64 bits. Il devrait donc manipuler les entiers de 0 à :

$$1\ 844\ 674\ 073\ 709\ 551\ 616,$$

et afficher correctement ce qu'on lui demande d'afficher. On ne devrait avoir de problème qu'avec :

```
printf("%d\n'', 18446744073709551618);
```

par exemple.

Par contre on comprend que le problème apparaît avec un microprocesseur 32 bits.

Le problème pourrait venir du système d'exploitation et du compilateur. En fait, s'il existe de *vrais* microprocesseurs 64 bits, comme le *Dec Alpha* datant de 1993, les fondeurs tels que *Intel* ont essayé de produire des microprocesseurs 64 bits, tels que l'*Itanium*, mais n'y sont pas parvenus de façon satisfaisant jusqu'à ce que *AMD* se contente d'un compromis : sur l'*AMD64*, il y a 64 broches d'entrées-sorties, ce qui permet en particulier des transferts plus rapides qu'avec des microprocesseurs 32 bits, mais les calculs sur les entiers (addition, multiplication) reprennent

l'implémantation des microprocesseurs de la génération précédente et s'effectuent donc sur 32 bits.

Un microprocesseur 32 bits manipule les entiers de 0 à :

4 294 967 295.

Cela n'explique toujours pas l'affichage obtenu. En fait l'implémentation des entiers relatifs est prioritaire par rapport à celle des entiers naturels. Un microprocesseur n bits manipule donc les entiers relatifs compris entre 2^{n-1} et $2^{n-1} - 1$, soit entre :

- 2 147 483 648 et 2 147 483 647

pour un microprocesseur 32 bits.

On commence à comprendre puisque 12 345 678 901 est supérieur à 2 147 483 647.

Puisqu'on est en arithmétique modulaire, 12 345 678 901 se représente de la même façon que :

```
   12 345 678 901
-   4 294 967 296
-----------------
    8 050 711 605
```

et que :

```
    8 050 711 605
-   4 294 967 296
-----------------
    3 755 744 309
```

Le nombre obtenu est inférieur à 4 294 967 296 mais supérieur à 2 147 483 648. Pour obtenir une représentation comprise entre - 2 147 483 648 et 2 147 483 647, il faut encore soustraire 4 294 967 296. Cela donne un nombre négatif de valeur absolue :

```
    4 294 967 296
-   3 755 744 309
-----------------
      539 222 987
```

soit le nombre affiché.

Programmation avancée

Nous rencontrerons de temps en temps des problèmes de programmation que nous ne résoudrons pas dans le chapitre que nous sommes en train d'étudier. Nous les indiquerons à la fin du chapitre sous la rubrique *programmation avancée*. Ceci ne signifie pas particulièrement qu'il existe un cours d'initiation à la programmation suivi d'un cours de programmation avancée. Quel que soit votre niveau de connaissance en programmation, vous rencontrerez toujours des problèmes de programmation avancée. Nous mettrons dans cette rubrique les problèmes qui ont un rapport direct avec le contenu du chapitre, que l'on peut se poser naturellement, mais que nous ne programmerons que plus tard pour des raisons diverses. L'intérêt de dégager ces problèmes est, d'une part, de s'en souvenir lors des relectures successives et, d'autre part, d'avoir une liste d'exercices moins triviaux pour ceux qui en connaîtraient un peu plus que les autres.

Nous avons vu que la sémantique réelle ne correspond pas nécessairement à la sémantique attendue alors que cette première n'était pas toujours irréaliste. Nous allons demander d'émuler (dans un langage quelconque, en particulier en langage C) des instructions de remplacement dont la sémantique réelle correspondra cette fois ci à la sémantique attendue.

Exercice 1.- *Concevoir un programme qui, lorsqu'on entre au clavier un entier naturel de longueur aussi grande que l'on veut affiche ce même entier à l'écran (après avoir appuyé sur la touche* return*).*

Soyons cependant réaliste, le « aussi grand que l'on veut » sera limité par la mémoire de l'ordinateur. Il est de toute façon limité par le nombre fini de particules élémentaires de l'univers.

Exercice 2.- *Concevoir un programme qui demande deux entiers naturels, de longueur aussi grande que l'on veut, et qui affiche leur somme à l'écran.*

Exercice 3.- *Concevoir un programme qui demande deux entiers naturels, de longueur aussi grande que l'on veut, et qui affiche leur produit à l'écran.*

Exercice 4.- *Concevoir un programme qui demande deux entiers relatifs, de longueur aussi grande que l'on veut, et qui affiche leur différence à l'écran.*

Exercice 5.- *Concevoir un programme permettant de saisir, d'afficher et d'effectuer des calculs sur les rationnels.*