

## Chapter 3

# Variable, affectation et lecture

Nous venons d'écrire un grand nombre de programmes très simples en langage C en utilisant l'ordinateur comme une calculatrice. C'est déjà pas mal, les premiers appareils pouvant faire cela ne datent que des années 1940. Mais on peut faire plus que cela avec un langage de programmation. Nous allons voir dans ce chapitre l'intérêt d'utiliser des *variables*, et les deux instructions fondamentales d'*affectation* et de *lecture* pour pouvoir les utiliser.

### 3.1 Un premier programme

Donnons un problème (de programmation) à résoudre et voyons comment le recours à un petit programme peut nous aider.

Problème.- Soit la fonction réelle de la variable réelle  $f$  définie par :

$$f(x) = \frac{\sin x + \ln x}{e^x + 2}.$$

On veut étudier cette fonction.

Choix de la stratégie.- Une façon de faire est d'étudier l'allure de la courbe comme on le fait au lycée. Mais à un certain moment, soit ceci vous semble trop difficile et vous allez essayer de construire l'allure de la courbe point par point, soit, pour obtenir un graphe plus précis, vous décidez de calculer un grand nombre de points.

Vous pouvez pour cela utiliser une calculette scientifique, ou le langage C en tant que calculette scientifique, et écrire un certain nombre de fois l'expression numérique avec des valeurs différentes pour  $x$ . Mais ceci risque d'être assez long, sans oublier les possibilités d'erreurs dans l'écriture répétée de l'expression numérique.

Il est vrai que vous pouvez être aidé par l'éditeur de textes. Mais il faut quand même remplacer à chaque fois trois valeurs numériques.

La programmation permet d'*affecter* une seule fois la valeur à  $x$  (exactement comme en Mathématiques) et de faire afficher la valeur  $f(x)$ .

Un programme.- Voici un exemple de programme pour cela, qui se comprend de lui-même, et que nous commenterons ensuite pour que vous soyez capables d'écrire des programmes analogues.

```
/* fonct\_1.c */
#include <stdio.h>
#include <math.h>

void main(void)
{
    float x, y;
    x = 2;
    y = (sin(x) + log(x))/(exp(x) + 2);
    printf("f(%f) = %f", x, y);
}
```

Utilisation de ce programme.- Lorsqu'on fait exécuter ce programme, on obtient une ligne de la forme suivante :

```
f(2.000000) = 0.170672
```

Il suffit de modifier légèrement le programme, en ne changeant que la valeur affectée à  $x$ , autant de fois que nécessaire pour obtenir les valeurs désirées.

Commentaires.- 1°) On utilise des **variables**, nommées ici  $x$  et  $y$ .

2°) Toute variable doit être **déclarée**, et sa nature doit être donnée. Techniquement, on doit spécifier le **type** (ici de type réel flottant). C'est l'objet de la septième ligne du programme.

3<sup>o</sup>) La huitième ligne est une instruction d'**affectation** : elle indique qu'il faut donner (on dit **affecter**) telle valeur (ici 2) à telle variable (ici  $x$ ).

La neuvième ligne est également une instruction d'affectation, un peu plus complexe.

Exercice 1.- Écrire le programme précédent et le faire exécuter. Recommencer en modifiant la valeur de  $x$ .

Exercice 2.- Concevoir un programme analogue pour d'autres fonctions, à une ou plusieurs variables.

## 3.2 L'affectation

L'instruction nouvelle que nous venons d'introduire est celle d'*affectation*. Explicitons-la un peu plus.

### 3.2.1 Variable

Qu'est-ce qu'une variable ? Vous connaissez déjà la notion de **variable** pour l'avoir rencontrée en mathématiques. Aucune définition ne vous en a été donnée. On peut seulement la comparer à une boîte qui peut prendre des valeurs différentes à des instants différents.

Nom d'une variable.- Lorsqu'on utilise plusieurs variables, donc plusieurs boîtes, il est intéressant, pour ne pas dire fondamental, de pouvoir les distinguer. Une façon de faire est de leur donner un nom (un nom différent par variable). En mathématiques, depuis Descartes au *XVII<sup>e</sup>* siècle, on utilise les lettres minuscules de la fin de l'alphabet  $x, y, z$ , éventuellement indexées  $x_1, x_2, \dots$  et, à vrai dire, sans règles bien précises. Dans un langage informatique ces règles doivent être tout à fait précises. On appelle **identificateur** les noms possibles pour ce qui a besoin d'être nommé, les variables en particulier, mais aussi les noms d'autres entités.

### 3.2.2 Identificateur

Écriture des identificateurs.- Il n'est pas facile d'écrire des indices dans un langage informatique, on ne les utilise donc pas. En fait ceci date des débuts des langages informatiques, car ceci serait possible actuellement, mais on continue la tradition. Les identificateurs seront donc des mots écrits linéairement sur un alphabet propre à chaque langage, comprenant les vingt-six lettres, ou caractères, (non accentuées) de 'A' à 'Z', et éventuellement en minuscule de 'a' à 'z', les chiffres de '0' à '9', et des **caractères spéciaux** (c'est-à-dire les autres symboles) tels que '\$', '%', '\_' ... Un certain nombre de mots ne peuvent pas être utilisés comme identificateurs ; on les appelle les **mots réservés**. Ils servent pour la combinaison des instructions élémentaires. Au vu des programmes précédents on se doute par exemple que les mots "float" et "void" sont des mots réservés du langage C.

Les identificateurs du langage C.- En langage C, les identificateurs sont des mots écrits linéairement sur l'alphabet comprenant les vingt-six lettres (non accentuées) majuscules de 'A' à 'Z', les vingt-six lettres minuscules de 'a' à 'z', les chiffres de '0' à '9', et le blanc souligné '\_', avec la restriction que ce mot ne doit ni commencer par un chiffre, ni être un mot réservé.

La longueur peut être quelconque mais le compilateur ne tient compte en général que des 32 premiers caractères, et même quelquefois que des 8 premiers.

Mots réservés.- Les mots réservés du langage C standard sont les suivants :

```

auto      break   case   char   const
continue  default  do     double else
enum      extern  float  for     goto
if        int     long   register return
short     signed  sizeof static  struct
switch    typedef union  unsigned void
volatile  while

```

Exercice 1.- Parmi les mots suivants indiquer ceux qui sont des identificateurs du langage C et ceux qui ne le sont pas :

```

A, A1, 1A,
n, n_1, n-1, n 1,
Racine_carrée, Racine_carree,
Reel, Real.

```

Exercice 2.- Parmi les paires suivantes d'identificateurs indiquer lesquelles représentent la même variable et lesquelles représentent des variables différentes :

```

Puissance, puissance,
Racine_carree, RacineCarree,
Constitution_1, Constitution_2.

```

### 3.2.3 Type

#### 3.2.3.1 Généralités sur les types

Intérêt des types.- Si nous poursuivons l'analogie d'une variable et d'une boîte, il peut être intéressant de savoir ce que doit contenir la boîte (pour qu'elle soit assez grande, ou qu'elle soit plus sophistiquée avec des séparations). Autrement dit il est intéressant d'indiquer le type de boîte que l'on veut. On dit que l'on attribue un **type** à une variable donnée.

Typage implicite et typage explicite.- Le **typage** des variables peut être **implicite** (comme dans le langage BASIC que vous connaissez peut-être) ou **explicite** (avec obligation de le déclarer). Le typage est explicite en langage C.

Intérêt du typage explicite.- Le typage explicite possède à la fois des avantages et des inconvénients.

Le principal avantage du typage pour les utilisateurs est que le compilateur vérifie la **cohérence des types**. Imaginons, par exemple, que l'on écrive l'instruction suivante :

```
A = B + C;
```

avec B et C déclarées comme variables réelles et A déclarée comme variable entière. Il y a nécessairement une mauvaise conception du programme, que le compilateur détectera.

Un autre avantage, bien que passant inaperçu à l'utilisateur *a priori*, est que la déclaration du type permet à l'ordinateur de ne réserver que la place mémoire nécessaire (place moins importante pour un entier que pour un réel, par exemple).

Un inconvénient est que le programmeur peut ressentir l'obligation de typer ses variables comme quelque chose de contraignant. D'autre part la vérification de la cohérence des types ralentit le compilateur.

### 3.2.3.2 Cas des entiers

Les types.- On a les quatre types :

#### char int short long

(évidemment pour les mots anglais *character*, *integer*) signifiant caractère, entier, court et long).

Chacun de ces quatre types peut tous être précédés du modificateur **signed** (valeur par défaut, sauf pour **char**) ou **unsigned** pour désigner des entiers naturels ou des entiers relatifs. Le mot **signed** ou **unsigned** tout seul est considéré comme suivi du mot **int**.

- Le type **char** est un entier codé sur un octet. Il prend donc les valeurs 0 à 255. Il sert, en particulier, à représenter la valeur entière d'un caractère.
- Le type **short** code les entiers sur un certain nombre d'octets, ce nombre n'étant pas défini par la norme. Sur un IBM-PC, il est souvent codé sur deux octets, les entiers de ce type allant donc :
  - de -32 768 à 32 767 pour **signed short** ;
  - de 0 à 65 535 pour **unsigned short**.
- Le type **long** code les entiers sur un certain nombre d'octets, ce nombre n'étant pas défini par la norme, mais il doit être égal ou supérieur au nombre d'octets pour le type **short**. Sur un IBM-PC, il est souvent de quatre octets, les entiers de ce type allant donc :
  - de - 2 147 843 648 à 2 147 843 647 pour **signed long** ;
  - de 0 à 4 294 967 295 pour **unsigned long**.
- Le type **int** correspond à **short** ou à **long**, le type exact auquel il se réfère n'étant pas défini par la norme.

Les constantes.- Lorsqu'une constante de type **long** est placée dans un programme, il faut la faire suivre de L (sans blanc). Par défaut une constante entière est de type **int**. On aura, par exemple :

```
n = 2123L;
```

Format.- Le format pour afficher une expression de type **long** est %Ld.

### 3.2.3.3 Cas des réels

Types.- On a les types **float** et **double**, et quelquefois **long double** :

- un réel de type **float** est codé sur un certain nombre d'octets, celui-ci n'étant pas défini par la norme. Pour un IBM-PC, on a généralement 4 octets (avec en général une mantisse de 23 bits, un exposant de 8 bits et un signe de 1 bit), ce qui donne, pour les réels positifs, des nombres appartenant à l'intervalle de  $3,4 \cdot 10^{-38}$  à  $3,4 \cdot 10^{38}$  ;
- un réel de type **double** est codé sur un certain nombre d'octets, celui-ci n'étant pas défini par la norme mais étant égal ou supérieur à celui pour le type **float**. Pour un IBM-PC, on a généralement 8 octets (avec en général une mantisse de 52 bits, un exposant de 11 bits et un signe de 1 bit), ce qui donne, pour les réels positifs, des nombres appartenant à l'intervalle de  $1,7 \cdot 10^{-308}$  à  $1,7 \cdot 10^{308}$  ;

- un réel de type **long double** peut, par exemple être codé sur 10 octets (avec en général une mantisse de 64 bits, un exposant de 15 bits et un signe de 1 bit), ce qui donne, pour les réels positifs, des nombres appartenant à l'intervalle de  $3,4 \cdot 10^{-4932}$  à  $3,4 \cdot 10^{4932}$ .

Constantes.- Lorsqu'une constante de type **float** (respectivement de type **long double**) est placée dans un programme, il faut la faire suivre de **f** ou de **F** (respectivement de **L** ou de **l**) sans espace. Par défaut une constante réelle est de type **double**. On aura, par exemple :

```
X = 2.0L;
```

Format.- Le format pour afficher une expression de type **long double** est **%Lf** (attention ! **%lf** n'est pas compris).

### 3.2.3.4 Cas des caractères

Type.- On a vu le type **char** ci-dessus. Les caractères sont codés par des entiers à l'aide d'un code, par exemple le code *ASCII* pour beaucoup de systèmes d'exploitation. Ainsi le caractère 'A' est rangé dans la machine en tant qu'entier 65.

Format.- Le format pour afficher les caractères est **%c**.

Constante.- Une constante caractère se place entre apostrophes verticales. Par exemple le caractère 'a' s'écrit 'a'. Lorsqu'il s'agit d'un caractère non affichable on indique son code *ascii* précédé d'une contre-oblique (*antislash* en anglais), par exemple pour CTRL-Z de code *ascii* 26 on aura '\26'.

### 3.2.3.5 Autres types

Existe-t-il d'autres types ?

Oui. Nous venons de voir les types de bases. Nous verrons au fur et à mesure de nos besoins des **constructeurs de types**, permettant d'obtenir des types plus complexes, tels que les types tableaux et les types structurés.

## 3.2.4 Déclaration des variables

Introduction.- En langage C une variable doit toujours être **déclarée** avant d'être utilisée. Ceci n'est pas le cas de tous les programmes informatiques (par exemple le BASIC).

L'intérêt est le même que pour le typage. Si une variable non déclarée est utilisée, il y a une erreur dans le programme. Il suffit de consulter les déclarations de variables pour connaître la liste des variables utilisées. De plus la déclaration de variables facilite la compilation (mais ceci semble moins important à l'utilisateur).

Syntaxe.- Les variables se déclarent par des lignes de la forme :

```
type identificateur1, identificateur2, ... , identificateurn;
```

où **identificateur1**, **identificateur2**, ... , **identificateurn** sont les noms des variables (à savoir X ou Y dans notre exemple ci-dessus) et où **type** indique de quelle nature est la variable, **float** dans l'exemple ci-dessus.

Emplacement des déclarations.- Ces lignes ne peuvent pas se placer n'importe où à l'intérieur du programme. On peut déclarer des variables à l'intérieur de chaque bloc, mais à la condition que

ces déclarations soient placées avant la première instruction du bloc. On peut aussi déclarer des variables hors de toute fonction, par exemple avant la fonction `main()`.

### 3.2.5 Affectation

Introduction.- L'**affectation** est une instruction élémentaire très importante. Elle indique qu'il faut donner (on dit **affecter**) telle valeur à telle variable.

Syntaxe.- Une telle instruction est de la forme :

```
Identificateur = Expression ;
```

où **Identificateur** est une variable (déclarée) et **Expression** une expression dont le type est le même que celui de **Identificateur**.

Sémantique.- La valeur de l'expression (ce qui suppose que les variables apparaissant dans celle-ci possèdent une valeur) est affectée à la variable.

Remarquons à ce propos que toute variable possède une valeur à tout moment (le contenu de la mémoire correspondante). Les variables doivent donc être **initialisées**, sinon le contenu risque d'apparaître à l'utilisateur comme aléatoire.

Exercice 1.- *Quel est le résultat de l'exécution du programme suivant :*

```
/* affect_1.c */
#include <stdio.h>

void main(void)
{
    int x, y;
    x = 2;
    y = 3;
    y = x;
    printf("%d %d", x, y);
}
```

Exercice 2.- *Quel est le résultat de l'exécution du programme suivant :*

```
/* affect_2.c */
#include <stdio.h>

void main(void)
{
    int x, y;
    x = 2;
    y = x + 1;
    printf("%d", y);
}
```

Exercice 3.- *Quel est le résultat de l'exécution du programme suivant :*

```
/* affect\_3.c */
#include <stdio.h>
```

```
void main(void)
{
    int x;
    x = 2;
    x = 2*x;
    printf("%d", x);
}
```

Exercice 4.- *Quels sont les types des expressions suivantes ?* (On ne vous demande pas de les évaluer) :

```
2 * 3
5 * (7 + 4/2)
2.0000
```

### 3.2.6 Initialisation des variables

Introduction.- Nous avons vu l'intérêt d'initialiser les variables. On peut déclarer une variable puis l'initialiser ensuite. On peut également, en langage C, initialiser une variable lors de sa déclaration.

Exemple de la première méthode.- Si on veut initialiser un entier à la valeur 2 on peut écrire :

```
int i;
- - -
i = 2;
```

Syntaxe de la deuxième méthode.- L'initialisation lors de la déclaration prend la forme suivante :

```
type identificateur = valeur;
```

Exemple.- Pour le problème précédent on peut donc aussi écrire :

```
int i = 2;
```

### 3.2.7 Partie entière et conversion de type

Introduction.- Nous avons déjà remarqué 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. Nous venons de voir, d'autre part, que les entiers et les réels ne sont pas codés de la même façon.

Il existe, en langage C, une façon de convertir des entités de types cohérents d'un type à l'autre, par exemple des `float` en `double`, des `int` en `float`...

Ceci se fait grâce à l'opération de **conversion de type**. Il existe une conversion de type implicite. Mais il existe aussi une **conversion de type explicite** (en anglais **cast**, à savoir "joue le rôle de", comme dans les films).

Syntaxe.- On écrit :

```
(type) expression
```

pour forcer la valeur de l'expression `expression` à être de type `type`.

Sémantique.- La sémantique est claire en général, avec quelquefois une valeur approchée qui a moins de précision (cas du passage du type `double` à `float`, par exemple).

Dans le cas du passage d'un type réel à un type entier on obtient la partie entière.

Exemple.- Le programme suivant :

```
/* cast.c */
#include <stdio.h>

void main(void)
{
    printf("%d", 3.14);
    printf("\n%d\n", (int) 3.14);
}
```

permet de constater comment il faut calculer la partie entière. En effet l'exécution du programme donne :

```
1374389535
3
```

### 3.3 Ordres de lecture et d'écriture

Nous avons déjà vu quelques instructions d'affichage à l'écran. Voyons maintenant comment saisir un paramètre au clavier.

#### 3.3.1 Un exemple

Introduction.- Reprenons notre problème initial, à savoir calculer un certain nombre de valeurs de la fonction :

$$f(x) = \frac{\sin x + \ln x}{e^x + 2}.$$

Nous avons vu plusieurs façons de faire (réécrire un programme par valeur, changer les valeurs grâce à l'éditeur de texte, ne changer qu'une seule fois la valeur grâce à l'affectation) mais, pour toutes ces méthodes, il faut un programme nouveau (et donc compiler à chaque fois) pour chaque valeur considérée. Les ordres de lecture vont nous faciliter la vie.

Un programme.- Voici un exemple de programme utilisant un ordre de lecture :

```
/* fonct_2.c */
#include <stdio.h>
#include <math.h>

void main(void)
{
    float x, y;
    printf("x = ");
    scanf("%f",&x);
    y = (sin(x) + log(x))/(exp(x) + 2);
    printf("f(%f) = %f", x, y);
}
```

Utilisation de ce programme.- Lorsqu'on fait exécuter ce programme, on obtient une ligne de la forme suivante :

```
x = _
```

Le symbole “\_” indiquant en fait le **curseur** qui clignote. Entrez alors un réel (le curseur se déplacera), pour obtenir, par exemple :

```
x = 3.7865_
```

puis appuyez sur la touche *Return*. On obtient alors les deux lignes :

```
x = 3.7865
f(3.786500) = 0.015841
```

Il suffit de faire exécuter à nouveau le programme (sans le recompiler) autant de fois que nécessaire pour obtenir les valeurs désirées.

Exercice.- *Écrire le programme précédent et faites quelques essais. Pour cela il suffit de compiler une seule fois (s'il n'y a pas d'erreurs, sinon il faut le mettre au point) et pour chaque valeur de le faire exécuter.*

### 3.3.2 Ordre d'écriture formatée

Introduction.- Nous connaissons l'intérêt des ordres d'écriture depuis notre tout premier programme. Nous venons de voir l'intérêt des ordres de lecture. Nous allons préciser la syntaxe de ces ordres en langage C.

Il va s'agir plus exactement de ces ordres concernant les périphériques par défaut, à savoir le clavier pour les lectures et l'écran pour l'écriture. Nous verrons plus tard comment faire pour d'autres périphériques, en particulier pour l'écriture sur l'imprimante.

Syntaxe.- Un ordre d'écriture formatée est de la forme :

```
printf(FORMAT, ARG1, ... , ARGn);
```

où **FORMAT** est une chaîne de caractères contenant  $n$  (le même  $n$  que celui de **ARGn**) **directives de format** (commençant par '%') et où **ARG1, ... , ARGn** sont des expressions.

Sémantique.- La chaîne de caractères sera affichée telle quelle sauf en ce qui concerne les séquences d'échappement et les directives de format.

Nous avons déjà vu comment sont affichées les séquences d'échappement, par exemple '\\ ' par '\ ' et '\n ' par un passage à la ligne.

La  $i$ -ième directive de format sera remplacée par la valeur du  $i$ -ième argument suivant les instructions de cette directive de format.

### 3.3.3 Ordre de saisie formatée

Syntaxe.- On a :

```
scanf(format, &variable);
```

où **format** est une directive de format (commençant par '%') et où **variable** est une variable.

L'esperluette (*ampersand* en anglais) & est appelée l'**opérateur d'adressage**.

Sémantique.- Lorsque cet ordre est rencontré, l'ordinateur attend que quelque chose soit entré au clavier, ce qu'il indique par un curseur clignotant à l'écran. Il affiche à l'écran en écho le mot entré de la même façon qu'un ordre de lecture. Il sait que ce mot est entré complètement

lorsqu'on appuie sur la touche *return*. Le curseur est alors placé (à l'écran), non pas juste après ce qui vient d'être entré, mais au début de la ligne suivante puis l'ordinateur passe à l'instruction suivante. Bien entendu, de plus, la valeur entrée est affectée à la variable. On dit que l'ordre de lecture est une **instruction bloquante** car on ne passe à l'instruction suivante que lorsque l'utilisateur a réagi.

Commentaires.- 1°) Une variable, par exemple *x*, contient la valeur de celle-ci alors que *&x* contient son adresse. En langage C la saisie a besoin de connaître l'adresse de la variable pour placer sa valeur.

2°) Une saisie se terminant par un retour chariot, le curseur va à la ligne suivante après celle-ci.

3°) Bien que cela ne soit pas syntaxiquement nécessaire, il faut toujours faire précéder la saisie d'une donnée par un **prompteur** ou **invite** (un texte à l'écran grâce à un appel à `printf()`) indiquant ce que l'on attend de l'utilisateur du programme. Cela aide beaucoup l'utilisateur pour lui permettre de savoir ce qu'il doit entrer.

4°) Bien que cela soit permis par la syntaxe du langage C, nous déconseillons fortement de saisir plusieurs données à la fois comme conséquence de ce qui précède ; d'ailleurs nous n'en indiquons pas la syntaxe.

5°) Si le type de la valeur saisie n'est pas cohérent avec l'indication de formatage alors la fonction `scanf()` s'arrête sans s'occuper des éventuelles saisies qui restent en attente.

Exercice 1.- Écrire un programme C qui lit un entier, le multiplie par 2 et affiche le résultat.

Exercice 2.- Écrire un programme C qui demande le rayon d'un cercle, donne la circonférence de ce cercle et l'aire du disque délimité par ce cercle.

Exercice 3.- Écrire un programme C qui demande le prix hors taxes d'un produit et affiche le prix toutes taxes comprises.

### 3.3.4 Cas des chaînes de caractères

Introduction.- Nous ne voyons pas comment entrer du texte avec les ordres de lecture vus précédemment, puisque les seuls types de variables sont, pour l'instant, ceux de nombres (entiers et réels).

Un exemple.- Le programme suivant se comprend aisément ; nous expliciterons après les aspects nouveaux :

```
/* prenom.c */
#include <stdio.h>

void main(void)
{
    char prenom[20];
    printf("Quel est votre prenom ? ");
    scanf("%s", prenom);
    printf("Bonjour %s", prenom);
}
```

Déclaration des variables chaînes de caractères.- Sans que nous cherchions pour l'instant à comprendre complètement pourquoi (nous y reviendrons plus tard), disons qu'un mot (on dit plutôt en informatique une **chaîne de caractères**, *string* en anglais) de nom `mot` d'au plus 19 caractères se déclare :

```
char mot[20];
```

(nous verrons également, plus tard, pourquoi 20 et non 19, c'est-à-dire un de plus).

Format.- Le format pour saisir ou afficher une chaîne de caractères (en tant que valeurs d'une variable ; nous avons vu que le format n'est pas nécessaire pour les constantes) est `%s`.

Saisie d'une chaîne de caractères.- La saisie se fait comme les autres types mais on remarquera l'absence de l'esperluette. Nous l'expliquerons plus tard.

Exercice.- *Faites exécuter le programme donné en exemple.*

### 3.3.5 Saisie d'un caractère

Intérêt.- Il n'y a rien de vraiment nouveau à propos de la saisie d'un caractère ; nous allons seulement donner un exemple et en expliquer l'intérêt. L'intérêt de saisir un caractère, même si cela n'apparaît pas à l'utilisateur, est la gestion de la mémoire. La place retenue en mémoire par la déclaration d'un caractère est évidemment (beaucoup) moindre que celle pour une chaîne de caractères.

Exemple.- L'exemple ci-dessous est caractéristique de l'utilisation du type `char` dans un programme :

```
/* char.c */
#include <stdio.h>

void main(void)
{
    char c;
    printf("Aimez-vous Brahms ? (O/N) ");
    scanf("%c", &c);
}
```

Exercice.- Indiquer ce que fait ce programme. *Le faire exécuter.*

Commentaire.- Le programme demande si l'on aime le compositeur en question et attend une réponse parmi les deux qui sont indiquées : 'O' pour oui, 'N' pour non. Le programme est cependant un peu décevant puisque, quelle que soit la réponse, il s'arrête (quel était l'intérêt de la question dans ces conditions ?). Nous verrons que c'est l'introduction des structures de contrôle qui nous permettra d'aller plus loin intelligemment.

## 3.4 Les constantes

L'intérêt des variables est qu'elles peuvent prendre des valeurs que l'on ne connaît pas *a priori*. Si l'on connaît la valeur alors autant la considérer tout de suite. Ce principe a des exceptions. Si on doit utiliser plusieurs fois cette valeur dans un programme, surtout si son écriture est un mot assez long, on a intérêt à lui donner un nom auquel on affecte une valeur, cette valeur ne pouvant plus être changée. Ceci conduit à la notion de **constante** au sens informatique.

### 3.4.1 Déclaration des constantes

Il y a deux façons de déclarer une constante en langage C.

Première méthode.- Dans la première version de C (le C K-R) il n'y avait pas de notion de constante proprement dite. On pouvait l'émuler en utilisant la commande du préprocesseur appelée `#define`.

La syntaxe est :

```
#define identificateur expression
```

où `expression` est une expression constante. Il vaut mieux placer cette commande en début de programme, comme toute commande du préprocesseur.

Le préprocesseur remplace toute occurrence de `identificateur` dans le programme par l'expression.

On remarquera qu'il n'y a pas de signe d'égalité entre l'identificateur et l'expression.

Exemple 1.- Puisque  $\pi$  est souvent utilisé, on peut définir :

```
#define pi 3.1415926
```

Deuxième méthode.- Cette méthode n'est valable que pour le C ANSI. C'est un emprunt au langage PASCAL via le langage C++. Les constantes se déclarent aux mêmes positions que les variables, de la façon suivante :

```
const type Identificateur = valeur ;
```

'const' (pour constante) est l'appel de déclaration de constantes. Chaque constante a un nom, qui est un identificateur, non utilisé par ailleurs et qui ne pourra plus être attribué à une autre entité (constante, variable...). La valeur de cette constante est donnée. Le type de la constante est précisé explicitement.

Exemple 2.- On peut définir  $\pi$  de la façon suivante :

```
const double pi = 3.1415926 ;
```

### 3.4.2 Exemples d'utilisation

Exemple 1.- Le programme suivant demande le rayon d'un cercle, donne la circonférence de ce cercle et l'aire du disque délimité par ce cercle :

```
/* disque.c */
#include <stdio.h>

void main(void)
{
    const double pi = 3.14159;
```

```
float rayon;
printf("Quel est le rayon du cercle ? ");
scanf("%f", &rayon);
printf("Sa circonference est %f.", 2*pi*rayon);
printf("\nSon aire est %f.\n", pi*rayon*rayon);
}
```

Exemple 2.- Un autre cas d'utilisation des constantes est celui d'une variable qui change peu souvent de valeur, par exemple le taux de la TVA pour une catégorie donnée d'articles. Dans ce cas on a intérêt à la déclarer comme constante plutôt que comme variable qu'il faudrait initialiser à chaque utilisation du programme. Il suffira de changer sa valeur de temps en temps et de recompiler le programme à ce moment-là.

Voilà un tel petit exemple :

```
/* prix.c */
#include <stdio.h>

void main(void)
{
    const float TVA = 19.6;
    float prixHT, prixTTC;
    printf("Donner le prix hors taxes : ");
    scanf("%f", &prixHT);
    prixTTC = prixHT*(1 + TVA/100);
    printf("Le prix toutes taxes comprises");
    printf(" est : %10.2f.\n", prixTTC);
}
```