Chapitre 1
Généralités sur le langage C

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur deux exemples commentés. Vous y découvrirez (pour l'instant, de façon encore informelle) comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux des structures fondamentales (boucle avec compteur, choix).

Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont l'édition, la compilation, l'édition de liens et l'exécution.

1- Présentation par l'exemple de quelques instructions du langage C

1.1 Un exemple de programme en langage C

Voici un exemple de programme en langage C, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonctionnement.

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5

main()
{ int i ;
  float x ;
  float racx ;

  printf ("Bonjour\n") ;
  printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;

  for (i=0 ; i<NFOIS ; i++)
    { printf ("Donnez un nombre : ") ;
      scanf ("%f", &x)  ;
      if (x < 0.0)
       printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
       else
        { racx = sqrt (x) ;
          printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
        }
    }
  printf ("Travail terminé - Au revoir") ;
}
```

```
Bonjour
Je vais vous calculer 5 racines carrées
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carrée : 2.000000
Donnez un nombre : 2
Le nombre 2.000000 a pour racine carrée : 1.414214
Donnez un nombre : -3
Le nombre -3.000000 ne possède pas de racine carrée
Donnez un nombre : 5.8
Le nombre 5.800000 a pour racine carrée : 2.408319
Donnez un nombre : 12.58
Le nombre 12.580000 a pour racine carrée : 3.546829
Travail terminé - Au revoir
```

Nous reviendrons un peu plus loin sur le rôle des trois premières lignes. Pour l'instant, admettez simplement que le symbole `NFOIS` est équivalent à la valeur `5`.

1.2 Structure d'un programme en langage C

La ligne :

```
main()
```

se nomme un « en-tête ». Elle précise que ce qui sera décrit à sa suite est en fait le *programme principal* (`main`). Lorsque nous aborderons l'écriture des fonctions en C, nous verrons que celles-ci possèdent également un tel en-tête ; ainsi, en C, le programme principal apparaîtra en fait comme une fonction dont le nom (`main`) est imposé.

Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades « `{` » et « `}` ». On dit que les instructions situées entre ces accolades forment un « bloc ». Ainsi peut-on dire que la fonction `main` est constituée d'un en-tête et d'un bloc ; il en ira de même pour toute fonction C. Notez qu'un bloc peut lui-même contenir d'autres blocs (c'est le cas de notre exemple). En revanche, nous verrons qu'une fonction ne peut jamais contenir d'autres fonctions.

1.3 Déclarations

Les trois instructions :

```
int i ;
float x ;
float racx ;
```

sont des « déclarations ».

La première précise que la variable nommée `i` est de type `int`, c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C il existe plusieurs types d'entiers.

Les deux autres déclarations précisent que les variables `x` et `racx` sont de type `float`, c'est-à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C il existe plusieurs types flottants.

En C, comme dans la plupart des langages actuels, les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme (on devrait plutôt dire : au début de la fonction `main`). Il en ira de même pour toutes les variables définies dans une fonction ; on les appelle « variables locales » (en toute rigueur, les variables définies dans notre exemple sont des variables locales de la fonction `main`). Nous verrons également (dans le chapitre consacré aux fonctions) qu'on peut définir des variables en dehors de toute fonction ; on parlera alors de variables globales.

Remarque C99 Suivant la norme C99, une déclaration peut figurer à n'importe quel emplacement, pour peu qu'elle apparaisse avant que la variable correspondante ne soit utilisée.

1.4 Pour écrire des informations : la fonction `printf`

L'instruction :

```
    printf ("Bonjour\n") ;
```

appelle en fait une fonction prédéfinie (fournie avec le langage, et donc que vous n'avez pas à écrire vous-même) nommée `printf`. Ici, cette fonction reçoit un argument qui est :

```
"Bonjour\n"
```

Les guillemets servent à délimiter une « chaîne de caractères » (suite de caractères). La notation `\n` est conventionnelle : elle représente un caractère de fin de ligne, c'est-à-dire un caractère qui, lorsqu'il est envoyé à l'écran, provoque le passage à la ligne suivante. Nous verrons que, de manière générale, le langage C prévoit une notation de ce type (\ suivi d'un caractère) pour un certain nombre de caractères dits « de contrôle », c'est-à-dire ne possédant pas de graphisme particulier.

Notez que, apparemment, bien que `printf` soit une fonction, nous n'utilisons pas sa valeur. Nous aurons l'occasion de revenir sur ce point, propre au langage C. Pour l'instant, admettez que nous pouvons, en C, utiliser une fonction comme ce que d'autres langages nomment une « procédure » ou un « sous-programme ».

L'instruction suivante :

```
    printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;
```

ressemble à la précédente avec cette différence qu'ici la fonction `printf` reçoit deux arguments. Pour comprendre son fonctionnement, il faut savoir qu'en fait le premier argument de `printf` est ce que l'on nomme un « format » ; il s'agit d'une sorte de guide qui précise comment afficher les informations qui sont fournies par les arguments suivants (le cas échéant). Ici, on demande à `printf` d'afficher suivant ce format :

```
"Je vais vous calculer %d racines carrées\n"
```

la valeur de `NFOIS`, c'est-à-dire, la valeur `5`.

Ce format est, comme précédemment, une chaîne de caractères. Toutefois, vous constatez la présence d'un caractère `%`. Celui-ci signifie que le caractère suivant est, non plus du texte à afficher tel quel, mais un « code de format ». Ce dernier précise qu'il faut considérer la valeur reçue (en argument suivant, donc ici `5`) comme un entier et l'afficher en décimal. Notez bien que tout ce qui, dans le format, n'est pas un code de format, est affiché tel quel ; il en va ainsi du texte « `racines carrées\n` ».

Il peut paraître surprenant d'avoir à spécifier à nouveau dans le code format que `NFOIS(5)` est un entier alors que l'on pourrait penser que le compilateur est bien capable de s'en apercevoir (quoiqu'il ne puisse pas deviner que nous voulons l'écrire en décimal et non pas, par exemple, en hexadécimal). Nous aurons l'occasion de revenir sur ce phénomène dont l'explication réside

essentiellement dans le fait que `printf` est une fonction, autrement dit que les instructions correspondantes seront incorporées, non pas à la compilation, mais lors de l'édition de liens.

Cependant, dès maintenant, sachez qu'il vous faudra toujours veiller à accorder le code de format au type de la valeur correspondante. Si vous ne respectez pas cette règle, vous risquez fort d'afficher des valeurs totalement fantaisistes.

1.5 Pour faire une répétition : l'instruction `for`

Comme nous le verrons, en langage C, il existe plusieurs façons de réaliser une répétition (on dit aussi une « boucle »). Ici, nous avons utilisé l'instruction `for` :

```
    for (i=0 ; i<NFOIS ; i++)
```

Son rôle est de répéter le bloc (délimité par des accolades « `{` » et « `}` ») figurant à sa suite, en respectant les consignes suivantes :

- avant de commencer cette répétition, réaliser :

```
i = 0
```

- avant chaque nouvelle exécution du bloc (tour de boucle), examiner la condition :

```
i < NFOIS
```

si elle est satisfaite, exécuter le bloc indiqué, sinon passer à l'instruction suivant ce bloc : à la fin de chaque exécution du bloc, réaliser :

```
 i++
```

Il s'agit là d'une notation propre au langage C qui est équivalente à :

```
i = i + 1
```

En définitive, vous voyez qu'ici notre bloc sera répété cinq fois.

1.6 Pour lire des informations : la fonction `scanf`

La première instruction du bloc répété par l'instruction `for` affiche simplement le message `Donnez un nombre:`. Notez qu'ici nous n'avons pas prévu de changement de ligne à la fin. La seconde instruction du bloc :

```
scanf ("%f", &x) ;
```

est un appel de la fonction prédéfinie `scanf` dont le rôle est de lire une information au clavier. Comme `printf`, la fonction `scanf` possède en premier argument un format exprimé sous forme d'une chaîne de caractères, ici :

```
"%f"
```

ce qui correspond à une valeur flottante (plus tard, nous verrons précisément sous quelle forme elle peut être fournie ; l'exemple d'exécution du programme vous en donne déjà une bonne idée !). Notez bien qu'ici, contrairement à ce qui se produisait pour `printf`, nous n'avons aucune raison de trouver, dans ce format, d'autres caractères que ceux qui servent à définir un code de format.

Comme nous pouvons nous y attendre, les arguments (ici, il n'y en a qu'un) précisent dans quelles variables on souhaite placer les valeurs lues. Il est fort probable que vous vous attendiez à trouver simplement `x` et non pas `&x`.

En fait, la nature même du langage C fait qu'une telle notation reviendrait à transmettre à la fonction `scanf` la valeur de la variable `x` (laquelle, d'ailleurs, n'aurait pas encore reçu de valeur précise). Or, manifestement, la fonction `scanf` doit être en mesure de ranger la valeur qu'elle aura lue dans l'emplacement correspondant à cette variable, c'est-à-dire à son adresse. Effectivement, nous verrons que `&` est un opérateur signifiant *adresse de*.

Notez bien que si, par mégarde, vous écrivez `x` au lieu de `&x`, le compilateur ne détectera pas d'erreur. Au moment de l'exécution, `scanf` prendra l'information reçue en deuxième argument (valeur de `x`) pour une adresse à laquelle elle rangera la valeur lue. Cela signifie qu'on viendra tout simplement écraser un emplacement indéterminé de la mémoire ; les conséquences pourront alors être quelconques.

1.7 Pour faire des choix : l'instruction `if`

Les lignes :

```
if (x < 0.0)
   printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
 else
   { racx = sqrt (x) ;
     printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
   }
```

constituent une instruction de choix basée sur la condition `x < 0.0`. Si cette condition est vraie, on exécute l'instruction suivante, c'est-à-dire :

```
    printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
```

Si elle est fausse, on exécute l'instruction suivant le mot `else`, c'est-à-dire, ici, le bloc :

```
   { racx = sqrt (x) ;
     printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
   }
```

Notez qu'il existe un mot `else` mais pas de mot `then`. La syntaxe de l'instruction `if` (notamment grâce à la présence de parenthèses qui encadrent la condition) le rend inutile.

La fonction `sqrt` fournit la valeur de la racine carrée d'une valeur flottante qu'on lui transmet en argument.

Remarque

Une instruction telle que :

```
racx = sqrt (x) ;
```

est une instruction classique d'affectation : elle donne à la variable `racx` la valeur de l'expression située à droite du signe égal. Nous verrons plus tard qu'en C l'affectation peut prendre des formes plus élaborées.

Notez que C dispose de trois sortes d'instructions :

- des instructions simples, terminées obligatoirement par un point-virgule,
- des instructions de structuration telles que `if` ou `for`,
- des blocs (délimités par { et }).

Les deux dernières ont une définition « récursive » puisqu'elles peuvent contenir, à leur tour, n'importe laquelle des trois formes.

Lorsque nous parlerons d'instruction, sans précisions supplémentaires, il pourra s'agir de n'importe laquelle des trois formes ci-dessus.

1.8 Les directives à destination du préprocesseur

Les trois premières lignes de notre programme :

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
```

sont en fait un peu particulières. Il s'agit de directives qui seront prises en compte avant la traduction (compilation) du programme. Ces directives, contrairement au reste du programme, doivent être écrites à raison d'une par ligne et elles doivent obligatoirement commencer en début de ligne. Leur emplacement au sein du programme n'est soumis à aucune contrainte (mais une directive ne s'applique qu'à la partie du programme qui lui succède). D'une manière générale, il est préférable de les placer au début, comme nous l'avons fait ici.

Les deux premières directives demandent en fait d'introduire (avant compilation) des instructions (en langage C) situées dans les fichiers `stdio.h` et `math.h`. Leur rôle ne sera complètement compréhensible qu'ultérieurement.

Pour l'instant, notez que, dès lors que vous faites appel à une fonction prédéfinie, il est nécessaire d'incorporer de tels fichiers, nommés « fichiers en-têtes », qui contiennent des déclarations appropriées concernant cette fonction : `stdio.h` pour `printf` et `scanf`, `math.h` pour `sqrt`. Fréquemment, ces déclarations permettront au compilateur d'effectuer des contrôles sur le nombre et le type des arguments que vous mentionnerez dans l'appel de votre fonction.

Notez qu'un même fichier en-tête contient des déclarations relatives à plusieurs fonctions. En général, il est indispensable d'incorporer `stdio.h`.

La troisième directive demande simplement de remplacer systématiquement, dans toute la suite du programme, le symbole `NFOIS` par `5`. Autrement dit, le programme qui sera réellement compilé comportera ces instrutions :

```
    printf ("Je vais vous calculer %d racines carrées\n", 5) ;


    for (i=0 ; i<5 ; i++)
```

Notez toutefois que le programme proposé est plus facile à adapter lorsque l'on emploie une directive `define`.

Remarque

Important : Dans notre exemple, la directive `#define` servait à définir la valeur d'un symbole. Nous verrons (dans le chapitre consacré au préprocesseur) que cette directive sert également à définir ce que l'on nomme une « macro ». Une macro s'utilise comme une fonction ; en particulier, elle peut posséder des arguments. Mais le préprocesseur remplacera chaque appel par la ou les instructions C correspondantes. Dans le cas d'une (vraie) fonction, une telle substitution n'existe pas ; au contraire, c'est l'éditeur de liens qui incorporera (une seule fois quel que soit le nombre d'appels) les instructions machine correspondantes.

1.9 Un second exemple de programme

Voici un second exemple de programme destiné à vous montrer l'utilisation du type « caractère ». Il demande à l'utilisateur de choisir une opération parmi l'addition ou la multiplication, puis de fournir deux nombres entiers ; il affiche alors le résultat correspondant.

```
#include <stdio.h>
main()
{
   char op ;
   int n1, n2 ;
   printf ("opération souhaitée (+ ou *) ? ") ;
   scanf ("%c", &op) ;
   printf ("donnez 2 nombres entiers : ") ;
   scanf ("%d %d", &n1, &n2) ;
   if (op == '+') printf ("leur somme  est : %d ", n1+n2) ;
         else  printf ("leur produit est : %d ", n1*n2) ;
}
```

Ici, nous déclarons que la variable `op` est de type caractère (`char`). Une telle variable est destinée à contenir un caractère quelconque (codé, bien sûr, sous forme binaire !).

L'instruction `scanf ("%c", &op)` permet de lire un caractère au clavier et de le ranger dans `op`. Notez le code `%c` correspondant au type `char` (n'oubliez pas le `&` devant `op`). L'instruction `if` permet d'afficher la somme ou le produit de deux nombres, suivant le caractère contenu dans `op`. Notez que :

- la relation d'égalité se traduit par le signe == (et non = qui représente l'affectation et qui, ici, comme nous le verrons plus tard, serait admis mais avec une autre signification !).
- la notation '+' représente une constante caractère. Notez bien que C n'utilise pas les mêmes délimiteurs pour les chaînes (il s'agit de ") et pour les caractères.

Remarquez que, tel qu'il a été écrit, notre programme calcule le produit, dès lors que le caractère fourni par l'utilisateur n'est pas +.

Remarques

On pourrait penser à inverser l'ordre des deux instructions de lecture en écrivant :

```
scanf ("%d %d", &n1, &n2) ;
   ...
scanf ("%c", &op) ;
```

Toutefois, dans ce cas, une petite difficulté apparaîtrait : le caractère lu par le second appel de `scanf` serait toujours différent de + (ou de *). Il s'agirait en fait du caractère de fin de ligne `\n` (fourni par la validation de la réponse précédente). Le mécanisme exact vous sera expliqué dans le chapitre relatif aux « entrées-sorties conversationnelles » ; pour l'instant, sachez que vous pouvez régler le problème en effectuant une lecture d'un caractère supplémentaire.

Au lieu de :

```
scanf ("%d", &op) ;
```

on pourrait écrire :

```
op = getchar () ;
```

Cette instruction affecterait à la variable `op` le résultat fourni par la fonction `getchar` (qui ne reçoit aucun argument - n'omettez toutefois pas les parenthèses !).

D'une manière générale, il existe une fonction symétrique `putchar` ; ainsi :

```
putchar (op) ;
```

affiche le caractère contenu dans `op`.

Notez que généralement `getchar` et `putchar` sont, non pas des vraies fonctions, mais des macros dont la définition figure dans `stdio.h`.

2- Quelques règles d'écriture

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme en langage C. Nous y parlerons précisément de ce que l'on appelle les « identificateurs » et les « mots-clés », du format libre dans lequel on écrit les instructions, ainsi que de l'usage des séparateurs et des commentaires.

2.1 Les identificateurs

Les identificateurs servent à désigner les différents « objets » manipulés par le programme : variables, fonctions, etc. (Nous rencontrerons ultérieurement les autres objets manipulés par le langage C : constantes, étiquettes de structure, d'union ou d'énumération, membres de structure ou d'union, types, étiquettes d'instruction `GOTO`, macros). Comme dans la plupart des langages, ils sont formés d'une suite de caractères choisis parmi les lettres ou les chiffres, le premier d'entre eux étant nécessairement une lettre.

En ce qui concerne les lettres :

- le caractère souligné (_) est considéré comme une lettre. Il peut donc apparaître au début d'un identificateur. Voici quelques identificateurs corrects :

```
lg_lig    valeur_5   _total   _89
```

- les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes. Ainsi, en C, les identificateurs *ligne* et *Ligne* désignent deux objets différents.

En ce qui concerne la longueur des identificateurs, la norme ANSI prévoit qu'au moins les 31 premiers caractères soient « significatifs » (autrement dit, deux identificateurs qui diffèrent par leurs 31 premières lettres désigneront deux objets différents).

2.2 Les mots-clés

Certains « mots-clés » sont réservés par le langage à un usage bien défini et ne peuvent pas être utilisés comme identificateurs. En voici la liste, classée par ordre alphabétique.

Les mots-clés du langage C

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

2.3 Les séparateurs

Dans notre langue écrite, les différents mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne.

Il en va quasiment de même en langage C dans lequel les règles vont donc paraître naturelles. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier (tel que : `, = ; * ( ) [ ] { }`) doivent impérativement être séparés soit par un espace, soit par une fin de ligne. En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme).

Ainsi, vous devrez impérativement écrire :

```
int x,y
```

et non :

```
intx,y
```

En revanche, vous pourrez écrire indifféremment :

```
int n,compte,total,p
```

ou plus lisiblement :

```
int n, compte, total, p
```

2.4 Le format libre

Le langage C autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites ; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

Bien entendu, cette liberté de mise en page possède des contreparties. Notamment, le risque existe, si l'on n'y prend garde, d'aboutir à des programmes peu lisibles.

À titre d'exemple, voyez comment pourrait être (mal) présenté notre programme précédent :

Exemple de programme mal présenté

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
main() {  int i ;  float
       x
  ; float racx ; printf ("Bonjour\n") ; printf
     ("Je vais vous calculer %d racines carrées\n", NFOIS) ;  for (i=
    0 ; i<NFOIS ; i++) { printf ("Donnez un nombre : ") ; scanf ("%f"
     , &x) ;  if (x < 0.0)
   printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;  else
 { racx = sqrt (x) ; printf ("Le nombre %f a pour racine carrée : %f\n",
       x, racx) ;  }     }   printf ("Travail terminé - Au revoir") ;}
```

2.5 Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.

Ils sont formés de caractères quelconques placés entre les symboles `/*` et `*/`. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de commentaires :

```
/* programme de calcul de racines carrées */

/* commentaire fantaisiste &ç§{<>} ?%!!!!!! */

/* commentaire s'étendant
   sur plusieurs lignes
   de programme source    */

/* ============================================
   *    commentaire quelque peu esthétique    *
   *    et encadré, pouvant servir,         *
   *    par exemple, d'en-tête de programme   *
   ============================================ */
```

Voici un exemple de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int i ;             /* compteur de boucle */
float x ;           /* nombre dont on veut la racine carrée */
float racx ;        /* racine carrée du nombre */
```

Remarque C99 La norme C99 autorise une seconde forme de commentaire, dit « de fin de ligne », que l'on retrouve également en C++. Un tel commentaire est introduit par // et tout ce qui suit ces deux caractères jusqu'à la fin de la ligne est considéré comme un commentaire. En voici un exemple :

```
printf ("bonjour\n") ;    // formule de politesse
```

3- Création d'un programme en langage C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir : édition du programme, compilation et édition de liens.

3.1 L'édition du programme

L'édition du programme (on dit aussi parfois « saisie ») consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme qu'on nomme « programme source ». En général, ce texte sera conservé dans un fichier que l'on nommera « fichier source ».

Chaque système possède ses propres conventions de dénomination des fichiers. En général, un fichier peut, en plus de son nom, être caractérisé par un groupe de caractères (au moins 3) qu'on appelle une « extension » (ou, parfois un « type ») ; la plupart du temps, en langage C, les fichiers source porteront l'extension C.

3.2 La compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En langage C, compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes :

- traitement par le préprocesseur : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en langage C pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.

- compilation proprement dite, c'est-à-dire traduction en langage machine du texte en langage C fourni par le préprocesseur.

Le résultat de la compilation porte le nom de module objet.

3.3 L'édition de liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque, au moins, les différents modules objet correspondant aux fonctions prédéfinies (on dit aussi « fonctions standard ») utilisées par votre programme (comme `printf`, `scanf`, `sqrt`).

C'est effectivement le rôle de l'éditeur de liens que d'aller rechercher dans la bibliothèque standard les modules objet nécessaires. Notez que cette bibliothèque est une collection de modules objet organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers.

Le résultat de l'édition de liens est ce que l'on nomme un programme exécutable, c'est-à-dire un ensemble autonome d'instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu'il soit nécessaire de faire appel à un quelconque composant de l'environnement de programmation en C.

3.4 Les fichiers en-tête

Nous avons vu que, grâce à la directive `#include`, vous pouviez demander au préprocesseur d'introduire des instructions (en langage C) provenant de ce que l'on appelle des fichiers « en-tête ». De tels fichiers comportent, entre autres choses :

- des déclarations relatives aux fonctions prédéfinies,
- des définitions de macros prédéfinies.

Lorsqu'on écrit un programme, on ne fait pas toujours la différence entre fonction et macro, puisque celles-ci s'utilisent de la même manière. Toutefois, les fonctions et les macros sont traitées de façon totalement différente par l'ensemble « préprocesseur + compilateur + éditeur de liens ».

En effet, les appels de macros sont remplacés (par du C) par le préprocesseur, du moins si vous avez incorporé le fichier en-tête correspondant. Si vous ne l'avez pas fait, aucun remplacement ne sera effectué, mais aucune erreur de compilation ne sera détectée : le compilateur croira simplement avoir affaire à un appel de fonction ; ce n'est que l'éditeur de liens qui, ne la trouvant pas dans la bibliothèque standard, vous fournira un message.

Les fonctions, quant à elles, sont incorporées par l'éditeur de liens. Cela reste vrai, même si vous omettez la directive `#include` correspondante ; dans ce cas, simplement, le compilateur n'aura pas disposé d'informations appropriées permettant d'effectuer des contrôles d'arguments (nombre et type) et de mettre en place d'éventuelles conversions ; aucune erreur ne sera signalée à la compilation ni à l'édition de liens ; les conséquences n'apparaîtront que lors de l'exécution : elles peuvent être invisibles dans le cas de fonctions comme `printf` ou, au contraire, conduire à des résultats erronés dans le cas de fonctions comme `sqrt`.

Chapitre 2

Les types de base du langage C

Les types `char`, `int` et `float` que nous avons déjà rencontrés sont souvent dits « scalaires » ou « simples », car, à un instant donné, une variable d'un tel type contient une seule valeur. Ils s'opposent aux types « structurés » (on dit aussi « agrégés ») qui correspondent à des variables qui, à un instant donné, contiennent plusieurs valeurs (de même type ou non). Ici, nous étudierons en détail ce que l'on appelle les types de base du langage C ; il s'agit des types scalaires à partir desquels pourront être construits tous les autres, dits « types dérivés », qu'il s'agisse :

- de types structurés comme les tableaux, les structures ou les unions,
- d'autres types simples comme les pointeurs ou les énumérations.

Auparavant, cependant, nous vous proposons de faire un bref rappel concernant la manière dont l'information est représentée dans un ordinateur et la notion de type qui en découle.

1 La notion de type

La mémoire centrale est un ensemble de positions binaires nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par ce qu'on nomme son adresse.

L'ordinateur, compte tenu de sa technologie actuelle, ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être codée sous cette forme. Dans ces conditions, on voit qu'il ne suffit pas de connaître le contenu d'un emplacement de la mémoire (d'un ou de plusieurs octets) pour être en mesure de lui attribuer une signification. Par exemple, si vous savez qu'un octet contient le « motif binaire » suivant :

```
01001101
```

vous pouvez considérer que cela représente le nombre entier `77` (puisque le motif ci-dessus correspond à la représentation en base 2 de ce nombre). Mais pourquoi cela représenterait-il un nombre ? En effet, toutes les informations (nombres entiers, nombres réels, nombres complexes, caractères, instructions de programme en langage machine, graphiques...) devront, au bout du compte, être codées en binaire.

Dans ces conditions, les huit bits ci-dessus peuvent peut-être représenter un caractère ; dans ce cas, si nous connaissons la convention employée sur la machine concernée pour représenter les caractères, nous pouvons lui faire correspondre un caractère donné (par exemple M, dans le cas du code ASCII). Ils peuvent également représenter une partie d'une instruction machine ou d'un nombre entier codé sur deux octets, ou d'un nombre réel codé sur 4 octets, ou...

On comprend donc qu'il n'est pas possible d'attribuer une signification à une information binaire tant que l'on ne connaît pas la manière dont elle a été codée. Qui plus est, en général, il ne sera même pas possible de « traiter » cette information. Par exemple, pour additionner deux informations, il faudra savoir quel codage a été employé afin de pouvoir mettre en œuvre les bonnes instructions (en langage machine). Par exemple, on ne fait pas appel aux mêmes circuits électroniques pour additionner deux nombres codés sous forme « entière » et deux nombres codés sous forme « flottante ».

D'une manière générale, la notion de type, telle qu'elle existe dans les langages évolués, sert à régler (entre autres choses) les problèmes que nous venons d'évoquer.

Les types de base du langage C se répartissent en trois grandes catégories en fonction de la nature des informations qu'ils permettent de représenter :

- nombres entiers (mot-clé `int`),
- nombres flottants (mot-clé `float` ou `double`),
- caractères (mot-clé `char`) ; nous verrons qu'en fait `char` apparaît (en C) comme un cas particulier de `int`.

2 Les types entiers

2.1 Leur représentation en mémoire

Le mot-clé `int` correspond à la représentation de nombres entiers relatifs. Pour ce faire : un bit est réservé pour représenter le signe du nombre (en général `0` correspond à un nombre

positif) ; les autres bits servent à représenter la valeur absolue du nombre (en toute rigueur, on la représente sous la forme de ce que l'on nomme le « complément à deux ». Nous y reviendrons dans le chapitre 13).

2.2 Les différents types d'entiers

Le C prévoit que, sur une machine donnée, on puisse trouver jusqu'à trois tailles différentes d'entiers, désignées par les mots-clés suivants :

- `short int` (qu'on peut abréger en `short`),
- `int` (c'est celui que nous avons rencontré dans le chapitre précédent),
- `long int` (qu'on peut abréger en `long`).

Chaque taille impose naturellement ses limites. Toutefois, ces dernières dépendent, non seulement du mot-clé considéré, mais également de la machine utilisée : tous les `int` n'ont pas la même taille sur toutes les machines ! Fréquemment, deux des trois mots-clés correspondent à une même taille (par exemple, sur PC, `short` et `int` correspondent à 16 bits, tandis que `long` correspond à 32 bits).

À titre indicatif, avec 16 bits, on représente des entiers s'étendant de `-32 768` à `32 767` ; avec 32 bits, on peut couvrir les valeurs allant de `-2 147 483 648` à `2 147 483 647`.

Remarque En toute rigueur, chacun des trois types (`short`, `int` et `long`) peut être nuancé par l'utilisation du qualificatif `unsigned` (non signé). Dans ce cas, il n'y a plus de bit réservé au signe et on ne représente plus que des nombres positifs. Son emploi est réservé à des situations particulières. Nous y reviendrons dans le chapitre 13.

Remarque C99 La norme C99 introduit le type `long long`, ainsi que des types permettant de choisir :

- soit la taille correspondante, par exemple `int16` pour des entiers codés sur 16 bits ou `int32` pour des entiers codés sur 32 bits ;
- soit une taille minimale, par exemple `int_least32_t` pour un entier d'au moins 32 bits.

2.3 Notation des constantes entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire simplement sous forme décimale, avec ou sans signe, comme dans ces exemples :

```
+533    48    -273
```

Il est également possible d'utiliser une notation octale ou hexadécimale. Nous en reparlerons dans le chapitre 13.

3 Les types flottants

3.1 Les différents types et leur représentation en mémoire

Les types flottants permettent de représenter, de manière approchée, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation scientifique (ou exponentielle) bien connue qui consiste à écrire un nombre sous la forme $1.5\ 10^{22}$ ou $0.472\ 10^{-8}$; dans une telle notation, on nomme « mantisses » les quantités telles que `1.5` ou `0.472` et « exposants » les quantités telles que `22` ou `-8`.

Plus précisément, un nombre réel sera représenté en flottant en déterminant deux quantités M (mantisse) et E (exposant) telles que la valeur

$$M . B^{E}$$

représente une approximation de ce nombre. La base B est généralement unique pour une machine donnée (il s'agit souvent de 2 ou de 16) et elle ne figure pas explicitement dans la représentation machine du nombre.

Le C prévoit trois types de flottants correspondant à des tailles différentes : `float`, `double` et `long double`. La connaissance des caractéristiques exactes du système de codage n'est généralement pas indispensable, sauf lorsque l'on doit faire une analyse fine des erreurs de calcul. En revanche, il est important de noter que de telles représentations sont caractérisées par deux éléments :

- *la précision* : lors du codage d'un nombre décimal quelconque dans un type flottant, il est nécessaire de ne conserver qu'un nombre fini de bits. Or la plupart des nombres s'exprimant avec un nombre limité de décimales ne peuvent pas s'exprimer de façon exacte dans un tel codage. On est donc obligé de se limiter à une représentation approchée en faisant ce que l'on nomme une erreur de troncature. Quelle que soit la machine utilisée, on est assuré que cette erreur (relative) ne dépassera pas 10^{-6} pour le type `float` et 10^{-10} pour le type `long double`.
- *le domaine couvert*, c'est-à-dire l'ensemble des nombres représentables à l'erreur de troncature près. Là encore, quelle que soit la machine utilisée, on est assuré qu'il s'étendra au moins de 10^{-37} à 10^{+37}.

3.2 Notation des constantes flottantes

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale,
- exponentielle.

La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais, bien sûr, pas toutes les deux en même temps !). En voici quelques exemples corrects :

```
12.43     -0.38      -.38      4.      .27
```

En revanche, la constante `47` serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance, si ce n'est au niveau du temps d'exécution, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant).

La notation exponentielle utilise la lettre `e` (ou `E`) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès que l'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents) :

```
   4.25E4          4.25e+4          42.5E3
54.27E-32        542.7E-33        5427e-34
   48e13            48.e13          48.0E13
```

Par défaut, toutes les constantes sont créées par le compilateur dans le type `double`. Il est toutefois possible d'imposer à une constante flottante :

- d'être du type `float`, en faisant suivre son écriture de la lettre `F` (ou `f`) : cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision (le gain en place et la perte en précision dépendant de la machine concernée).
- d'être du type `long double`, en faisant suivre son écriture de la lettre `L` (ou `l`) : cela permet de gagner éventuellement en précision, en contrepartie d'une perte de place mémoire (le gain en précision et la perte en place dépendant de la machine concernée).

4 Les types caractères

4.1 La notion de caractère en langage C

Comme la plupart des langages, C permet de manipuler des caractères codés en mémoire sur un octet. Bien entendu, le code employé, ainsi que l'ensemble des caractères représentables, dépend de l'environnement de programmation utilisé (c'est-à-dire à la fois de la machine concernée et du compilateur employé). Néanmoins, on est toujours certain de disposer des lettres (majuscules et minuscules), des chiffres, des signes de ponctuation et des différents séparateurs (en fait, tous ceux que l'on emploie pour écrire un programme !). En revanche, les caractères nationaux (caractères accentués ou ç) ou les caractères semi-graphiques ne figurent pas dans tous les environnements.

Par ailleurs, la notion de caractère en C dépasse celle de caractère imprimable, c'est-à-dire auquel est obligatoirement associé un graphisme (et qu'on peut donc imprimer ou afficher sur un écran). C'est ainsi qu'il existe certains caractères de changement de ligne, de tabulation, d'activation d'une alarme sonore (cloche),... Nous avons d'ailleurs déjà utilisé le premier (sous la forme `\n`).

Remarque

De tels caractères sont souvent nommés « caractères de contrôle ». Dans le code ASCII (restreint ou non), ils ont des codes compris entre 0 et 31.

4.2 Notation des constantes caractères

Les constantes de type « caractère », lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quotes) le caractère voulu, comme dans ces exemples :

```
'a'        'Y'        '+'        '$'
```

Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère « \ », nommé « antislash » (en anglais, il se nomme « back-slash », en français, on le nomme aussi « barre inverse » ou « contre-slash »). Dans cette catégorie, on trouve également quelques caractères (`\`, `'`, `"` et `?`) qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteur qui les empêche d'être notés de manière classique entre deux apostrophes.

Voici la liste de ces caractères.

Caractères disposant d'une notation spéciale

NOTATION EN C	CODE ASCII (hexadécimal)	ABRÉVIATION USUELLE	SIGNIFICATION
\a	07	BEL	cloche ou bip (alert ou audible bell)
\b	08	BS	Retour arrière (Backspace)
\f	0C	FF	Saut de page (Form Feed)
\n	0A	LF	Saut de ligne (Line Feed)
\r	0D	CR	Retour chariot (Carriage Return)
\t	09	HT	Tabulation horizontale (Horizontal Tab)
\v	0B	VT	Tabulation verticale (Vertical Tab)
\\	5C	\	
\'	2C	'	
\"	22	"	
\?	3F	?	

De plus, il est possible d'utiliser directement le code du caractère en l'exprimant, toujours à la suite du caractère « antislash » :

- soit sous forme octale,
- soit sous forme hexadécimale précédée de `x`.

Voici quelques exemples de notations équivalentes, dans le code ASCII :

```
'A'     '\x41'   '\101'
'\r'    '\x0d'   '\15'    '\015'
'\a'    '\x07'   '\x7'    '\07'    '\007'
```

Remarques

En fait, il existe plusieurs versions de code ASCII, mais toutes ont en commun la première moitié des codes (correspondant aux caractères qu'on trouve dans toutes les implémentations) ; les exemples cités ici appartiennent bien à cette partie commune.

Le caractère `\`, suivi d'un caractère autre que ceux du tableau ci-dessus ou d'un chiffre de `0` à `7` est simplement ignoré. Ainsi, dans le cas où l'on a affaire au code ASCII, `\9` correspond au caractère `9` (de code ASCII 57), tandis que `\7` correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».

En fait, la norme prévoit deux types : `signed char` et `unsigned char` (`char` correspondant soit à l'un, soit à l'autre, suivant le compilateur utilisé). Là encore, nous y reviendrons dans le chapitre 13. Pour l'instant, sachez que cet attribut de signe n'agit pas sur la représentation d'un caractère en mémoire. En revanche, il pourra avoir un rôle dans le cas où l'on s'intéresse à la valeur numérique associée à un caractère.

5 Initialisation et constantes

1 Nous avons déjà vu que la directive `#define` permettait de donner une valeur à un symbole. Dans ce cas, le préprocesseur effectue le remplacement correspondant avant la compilation.

2 Par ailleurs, il est possible d'initialiser une variable lors de sa déclaration comme dans :

```
int n = 15 ;
```

Ici, pour le compilateur, `n` est une variable de type `int` dans laquelle il placera la valeur `15` ; mais rien n'empêche que cette valeur initiale évolue lors de l'exécution du programme. Notez d'ailleurs que la déclaration précédente pourrait être remplacée par une déclaration ordinaire (`int n`), suivie un peu plus loin d'une affectation (`n=15`) ; la seule différence résiderait dans l'instant où `n` recevrait la valeur `15`.

3 En fait, il est possible de déclarer que la valeur d'une variable ne doit pas changer lors de l'exécution du programme. Par exemple, avec :

```
const int n = 20 ;
```

on déclare `n` de type `int` et de valeur (initiale) `20` mais, de surcroît, les éventuelles instructions modifiant la valeur de `n` seront rejetées par le compilateur.

On pourrait penser qu'une déclaration (par `const`) remplace avantageusement l'emploi de `define`. En fait, nous verrons que les choses ne sont pas aussi simples, car les variables ainsi déclarées ne pourront pas intervenir dans ce qu'on appelle des « expressions constantes » (notamment, elles ne pourront pas servir de dimension d'un tableau !).

6 Autres types introduits par la norme C99

Outre les nouveaux types entiers dont nous avons parlé, la norme C99 introduit :

- le type booléen, sous le nom `bool` ; une variable de ce type ne peut prendre que l'une des deux valeurs : vrai (noté `true`) et faux (noté `false`) ;
- des types complexes, sous les noms `float complex`, `double complex` et `long double complex` ; la constante `I` correspond alors à la constante mathématique `i` (racine de –1).

Chapitre 3

Les opérateurs et les expressions en langage C

1 L'originalité des notions d'opérateur et d'expression en langage C

Le langage C est certainement l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (*arithmétiques*, *relationnels*, *logiques*) ou moins classiques (*manipulations de bits*). Mais, de surcroît, le C dispose d'un important éventail d'opérateurs originaux *d'affectation* et *d'incrémentation*.

Ce dernier aspect nécessite une explication. En effet, dans la plupart des langages, on trouve, comme en C :

- d'une part, des *expressions* formées (entre autres) à l'aide d'opérateurs,
- d'autre part, des *instructions* pouvant éventuellement faire intervenir des expressions, comme, par exemple, l'instruction d'affectation :

```
y = a * x +b ;
```

ou encore l'instruction d'affichage :

```
printf ("valeur %d", n + 2*p) ;
```

dans laquelle apparaît l'expression `n + 2 * p`.

Mais, généralement, dans les langages autres que C, l'expression possède une valeur mais ne réalise aucune action, en particulier aucune affectation d'une valeur à une variable. Au contraire, l'affectation y réalise une affectation d'une valeur à une variable mais ne possède pas de valeur. On a affaire à deux notions parfaitement disjointes. En langage C, il en va différemment puisque :

- d'une part, les (nouveaux) opérateurs d'incrémentation pourront non seulement intervenir au sein d'une expression (laquelle, au bout du compte, possédera une valeur), mais également agir sur le contenu de variables. Ainsi, l'expression (car, comme nous le verrons, il s'agit bien d'une expression en C) :

  ```
  ++i
  ```

  réalisera une action, à savoir : augmenter la valeur `i` de `1` ; en même temps, elle aura une valeur, à savoir celle de `i` après incrémentation.

- d'autre part, une affectation apparemment classique telle que :

  ```
  i = 5
  ```

  pourra, à son tour, être considérée comme une expression (ici, de valeur `5`). D'ailleurs, en C, l'affectation (=) est un opérateur. Par exemple, la notation suivante :

  ```
  k = i = 5
  ```

  représente une expression en C (ce n'est pas encore une instruction - nous y reviendrons). Elle sera interprétée comme :

  ```
  k = (i = 5)
  ```

  Autrement dit, elle affectera à `i` la valeur `5` puis elle affectera à `k` la valeur de l'expression `i = 5`, c'est-à-dire `5`.

En fait, en C, les notions d'expression et d'instruction sont étroitement liées puisque la principale instruction de ce langage est une expression terminée par un point-virgule. On la nomme souvent « instruction expression ». Voici des exemples de telles instructions qui reprennent les expressions évoquées ci-dessus :

```
++i ;
i = 5 ;
k = i = 5 ;
```

Les deux premières ont l'allure d'une affectation telle qu'on la rencontre classiquement dans la plupart des autres langages. Notez que, dans les deux cas, il y a évaluation d'une expression (`++i` ou `i=5`) dont la valeur est finalement inutilisée. Dans le dernier cas, la valeur de l'expression `i=5`, c'est-à-dire `5`, est à son tour affectée à `k` ; par contre, la valeur finale de l'expression complète est, là encore, inutilisée.

Ce chapitre vous présente la plupart des opérateurs du C ainsi que les règles de priorité et de conversion de type qui interviennent dans les évaluations des expressions. Les (quelques) autres opérateurs concernent essentiellement les pointeurs, l'accès aux tableaux et aux structures et les manipulations de bits. Ils seront exposés dans la suite de cet ouvrage.

2 Les opérateurs arithmétiques en C

2.1 Présentation des opérateurs

Comme tous les langages, C dispose d'opérateurs classiques « binaires » (c'est-à-dire portant sur deux « opérandes »), à savoir l'addition (`+`), la soustraction (`-`), la multiplication (`*`) et la division (`/`), ainsi que d'un opérateur « unaire » (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé noté `-` (comme dans `-n` ou dans `-x+y`).

Les opérateurs binaires ne sont a priori définis que pour deux opérandes ayant le même type parmi : `int`, `long int`, `float`, `double`, `long double` et ils fournissent un résultat de même type que leurs opérandes.

Remarque

En machine, il n'existe, par exemple, que des additions de deux entiers de même taille ou de flottants de même taille. Il n'existe pas d'addition d'un entier et d'un flottant ou de deux flottants de taille différente.

Mais nous verrons, dans le paragraphe 2, que, par le jeu des conversions implicites, le compilateur saura leur donner une signification :

- soit lorsqu'ils porteront sur des opérateurs de type différent,
- soit lorsqu'ils porteront sur des opérandes de type `char` ou `short`.

De plus, il existe un opérateur de modulo noté `%` qui ne peut porter que sur des entiers et qui fournit le reste de la division de son premier opérande par son second. Par exemple, `11%4` vaut `3`, `23%6` vaut `5`.

La norme ANSI ne définit les opérateurs `%` et `/` que pour des valeurs positives de leurs deux opérandes. Dans les autres cas, le résultat dépend de l'implémentation (C99 lève cette ambiguïté).

Notez bien qu'en C le quotient de deux entiers fournit un entier. Ainsi, `5/2` vaut `2` ; en revanche, le quotient de deux flottants (noté, lui aussi, `/`) est bien un flottant (`5.0/2.0` vaut bien approximativement `2.5`).

Remarque

Il n'existe pas d'opérateur d'élévation à la puissance. Il est nécessaire de faire appel soit à des produits successifs pour des puissances entières pas trop grandes (par exemple, on calculera x^3 comme `x*x*x`), soit à la fonction `power` de la bibliothèque standard (voyez éventuellement l'annexe).

2.2 Les priorités relatives des opérateurs

Lorsque plusieurs opérateurs apparaissent dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En C, comme dans les autres langages, les règles sont naturelles et rejoignent celles de l'algèbre traditionnelle (du moins, en ce qui concerne les opérateurs arithmétiques dont nous parlons ici).

Les opérateurs unaires + et − ont la priorité la plus élevée. On trouve ensuite, à un même niveau, les opérateurs *, / et %. Enfin, sur un dernier niveau, apparaissent les opérateurs binaires + et −.

En cas de priorités identiques, les calculs s'effectuent de gauche à droite. On dit que l'on a affaire à une *associativité de gauche à droite* (nous verrons que quelques opérateurs, autres qu'arithmétiques, utilisent une associativité de droite à gauche).

Enfin, des parenthèses permettent d'outrepasser ces règles de priorité, en forçant le calcul préalable de l'expression qu'elles contiennent. Notez que ces parenthèses peuvent également être employées pour assurer une meilleure lisibilité d'une expression.

Voici quelques exemples dans lesquels l'expression de droite, où ont été introduites des parenthèses superflues, montre dans quel ordre s'effectuent les calculs (les deux expressions proposées conduisent donc aux mêmes résultats) :

```
a + b * c                    a + ( b * c )
a * b + c % d                ( a * b ) + ( c % d )
- c % d                      ( - c ) % d
- a + c % d                  ( - a ) + ( c % d )
- a / - b + c                ( ( - a ) / ( - b ) ) + c
- a / - ( b + c )            ( - a ) / ( - ( b + c ) )
```

Remarques

Les règles de priorité interviennent pour définir la signification exacte d'une expression. Néanmoins, lorsque deux opérateurs sont théoriquement commutatifs, on ne peut être certain de l'ordre dans lequel ils seront finalement exécutés. Par exemple, une expression telle que `a+b+c` pourra aussi bien être calculée en ajoutant `c` à la somme de `a` et `b`, qu'en ajoutant `a` à la somme de `b` et `c`. Même l'emploi de parenthèses dans ce cas ne suffit pas à « forcer » l'ordre des calculs. Notez bien qu'une telle remarque n'a d'importance que lorsque l'on cherche à maîtriser parfaitement les erreurs de calcul.

Il est tout à fait possible qu'une opération portant sur deux valeurs entières conduise à un résultat non représentable dans le type concerné, parce que en dehors des limites permises, lesquelles, rappelons-le, dépendent de la machine employée. Dans ce cas, la plupart du temps, on obtient un résultat aberrant : les bits excédentaires sont ignorés, le résultat est ana- logue à celui obtenu lorsque la somme de deux nombres à 3 chiffres est un nombre à quatre chiffres dont on élimine le chiffre de gauche ; l'exécution du programme se poursuit sans que l'utilisateur ait été informé d'une quelconque anomalie. Notez bien à ce propos qu'un opérateur appliqué par exemple à deux opérandes de type `int` fournit toujours un résultat de type `int`,
même s'il n'est plus représentable dans ce type et qu'un type `long` aurait pu convenir.

De la même manière, il se peut qu'à un moment donné vous cherchiez à diviser un entier par zéro. Cette fois, la plupart du temps, cette anomalie est effectivement détectée : un message d'erreur est fourni à l'utilisateur, et l'exécution du programme est interrompue.

Comme les opérations entières, les opérations sur les flottants peuvent conduire à des résultats non représentables dans le type concerné (de valeur absolue trop grande ou trop petite). Dans ce cas, le comportement dépend des environnements de programmation utilisés ; en particulier, il peut y avoir arrêt de l'exécution du programme. Là encore, il faut bien noter qu'un opérateur

appliqué par exemple à deux opérandes de type `float` fournit toujours un résultat de type `float`, même s'il n'est plus représentable dans ce type et qu'un type `double` aurait pu convenir.

En ce qui concerne la division par zéro des flottants, elle conduit toujours à un message et à l'arrêt du programme.

3 Les conversions implicites pouvant intervenir dans un calcul d'expression

3.1 Notion d'expression mixte

Comme nous l'avons dit, les opérateurs arithmétiques ne sont définis que lorsque leurs deux opérandes sont de même type. Mais vous pouvez écrire ce que l'on nomme des « expressions mixtes » dans lesquelles interviennent des opérandes de types différents. Voici un exemple d'expression autorisée, dans laquelle `n` et `p` sont supposés de type `int`, tandis que `x` est supposé de type `float` :

```
n * x + p
```

Dans ce cas, le compilateur sait, compte tenu des règles de priorité, qu'il doit d'abord effectuer le produit `n*x`. Pour que cela soit possible, il va mettre en place des instructions de conversion de la valeur de `n` dans le type `float` (car on considère que ce type `float` permet de représenter à peu près convenablement une valeur entière, l'inverse étant naturellement faux). Au bout du compte, la multiplication portera sur deux opérandes de type `float` et elle fournira un résultat de type `float`.

Pour l'addition, on se retrouve à nouveau en présence de deux opérandes de types différents (`float` et `int`). Le même mécanisme sera mis en place, et le résultat final sera de type `float`.

Remarque

Attention, le compilateur ne peut que prévoir les instructions de conversion (qui seront donc exécutées en même temps que les autres instructions du programme) ; il ne peut pas effectuer lui-même la conversion d'une valeur que généralement il ne peut pas connaître.

3.2 Les conversions d'ajustement de type

Une conversion telle que `int -> float` se nomme une « conversion d'ajustement de type ». Une telle conversion ne peut se faire que suivant une hiérarchie qui permet de ne pas dénaturer la valeur initiale (on dit parfois que de telles conversions respectent l'intégrité des données), à savoir :

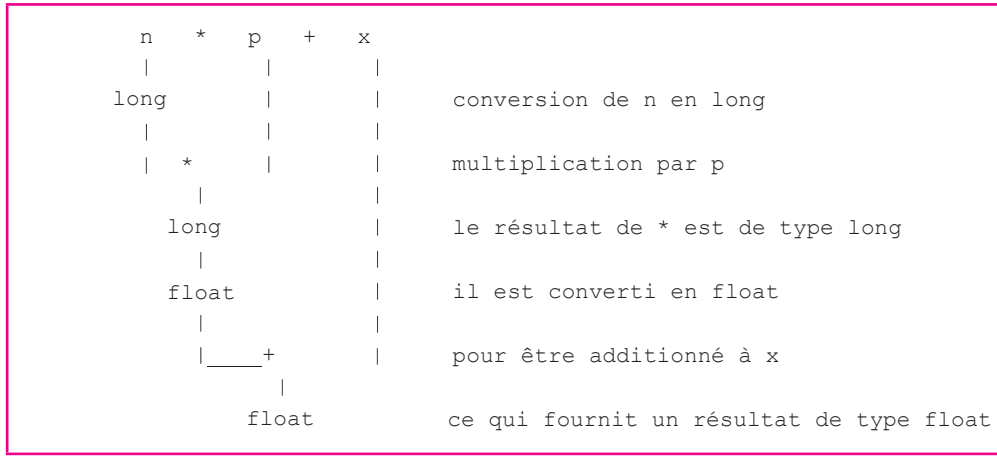
```
int -> long -> float -> double -> long double
```

On peut bien sûr convertir directement un `int` en `double` ; en revanche, on ne pourra pas convertir un `double` en `float` ou en `int`.

Notez que le choix des conversions à mettre en œuvre est effectué en considérant un à un les opérandes concernés et non pas l'expression de façon globale. Par exemple, si `n` est de type `int`, `p` de type `long` et `x` de type `float`, l'expression :

```
n * p + x
```

sera évaluée suivant ce schéma :



3.3 Les promotions numériques

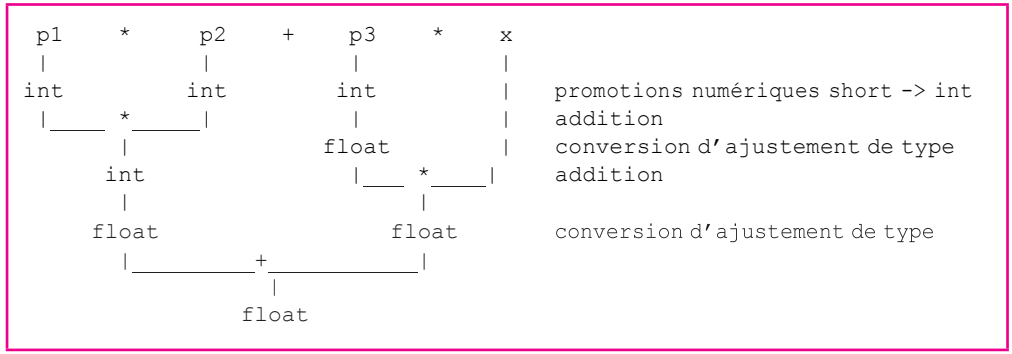
Les conversions d'ajustement de type ne suffisent pas à régler tous les cas. En effet, comme nous l'avons déjà dit, les opérateurs numériques ne sont pas définis pour les types `char` et `short`.

En fait, le langage C prévoit tout simplement que toute valeur de l'un de ces deux types apparaissant dans une expression est d'abord convertie en `int`, et cela sans considérer les types des éventuels autres opérandes. On parle alors, dans ce cas, de « promotions numériques » (ou encore de « conversions systématiques »).

Par exemple, si `p1`, `p2` et `p3` sont de type `short` et `x` de type `float`, l'expression :

```
p1 * p2 + p3 * x
```

est évaluée comme l'indique le schéma ci-après :



Notez bien que les valeurs des trois variables de type `short` sont d'abord soumises à la promotion numérique `short -> int` ; après quoi, on applique les mêmes règles que précédemment.

Remarque

En principe, comme nous l'avons déjà dit, les types entiers peuvent être non signés (`unsigned`). Nous y reviendrons dans le chapitre 13. Pour l'instant, sachez que nous vous déconseillons fortement de mélanger, dans une même expression, des types signés et des types non signés, dans la mesure où les conversions qui en résultent sont généralement dénuées de sens (et simplement faites pour préserver un motif binaire).

3.4 Le cas du type `char`

A priori, vous pouvez être surpris de l'existence d'une conversion systématique (promotion numérique) de `char` en `int` et vous interroger sur sa signification. En fait, il ne s'agit que d'une question de point de vue. En effet, une valeur de type caractère peut être considérée de deux façons :

- comme le caractère concerné : `a`, `Z`, fin de ligne,
- comme le code de ce caractère, c'est-à-dire un motif de 8 bits ; or à ce dernier on peut toujours faire correspondre un nombre entier (le nombre qui, codé en binaire, fournit le motif en question) ; par exemple, dans le code ASCII, le caractère `E` est représenté par le motif binaire `01000101`, auquel on peut faire correspondre le nombre `65`.

Effectivement, on peut dire qu'en quelque sorte le langage C confond facilement un caractère avec la valeur (entier) du code qui le représente. Notez bien que, comme toutes les machines n'emploient pas le même code pour les caractères, l'entier associé à un caractère donné ne sera pas toujours le même.

Voici quelques exemples d'évaluation d'expressions, dans lesquels on suppose que `c1` et `c2` sont de type `char`, tandis que `n` est de type `int`.

L'expression `c1+1` fournit donc un résultat de type `int`, correspondant à la valeur du code du caractère contenu dans `c1` augmenté d'une unité.

Ici, bien que les deux opérandes soient de type `char`, il y a quand même conversion préalable de leurs valeurs en `int` (promotions numériques).

Remarques

Théoriquement, en plus de ce qui vient d'être dit, il faut tenir compte de l'attribut de signe des caractères. Ainsi, lorsque l'on convertit un `unsigned char` en `int`, on obtient toujours un nombre entre 0 et 255, tandis que lorsque l'on convertit un `signed char` en `int`, on obtient un nombre compris entre -127 et 128. Nous y reviendrons en détail dans le chapitre 13.

Dans la première version de C (telle qu'elle a été définie initialement par Kernighan et Ritchie, c'est-à-dire avant la normalisation par le comité ANSI), il était prévu une promotion numérique `float -> double`. Certains compilateurs l'appliquent encore.

Les arguments d'appel d'une fonction peuvent être également soumis à des conversions. Le mécanisme exact est toutefois assez complexe dans ce cas, car il tient compte de la manière

dont la fonction a été déclarée dans le programme qui l'utilise (on peut trouver : aucune déclaration, une déclaration partielle ne mentionnant pas le type des arguments ou une déclaration complète dite prototype mentionnant le type des arguments).

Lorsque le type des arguments n'a pas été déclaré, les valeurs transmises en argument sont soumises aux règles précédentes (donc, en particulier, aux promotions numériques) auxquelles il faut ajouter la promotion numérique `float -> double`. Or, précisément, c'est ainsi que sont traitées les valeurs que vous transmettez à `printf` (ses arguments n'étant pas d'un type connu à l'avance, il est impossible au compilateur d'en connaître le type !). Ainsi :

- tout argument de type `char` ou `short` est converti en `int` ; autrement dit, le code `%c` s'applique aussi à un `int` : il affichera tout simplement le caractère ayant le code correspondant ; de même on obtiendra la valeur numérique du code d'un caractère `c` en écrivant : `printf("%d",c)`,

- tout argument de type `float` sera converti en `double` (et cela dans toutes les versions du C) ; ainsi le code `%f` pour `printf` correspond-il à un `double`, et il n'est pas besoin de prévoir un code pour un `float`.

4 Les opérateurs relationnels

Comme tout langage, C permet de comparer des expressions à l'aide d'opérateurs classiques de comparaison. En voici un exemple :

```
2 * a > b + 5
```

En revanche, C se distingue de la plupart des autres langages sur deux points :

- le résultat de la comparaison est, non pas une valeur booléenne (on dit aussi logique) prenant l'une des deux valeurs *vrai* ou *faux*, mais un entier valant :
  - 0 si le résultat de la comparaison est faux,
  - 1 si le résultat de la comparaison est vrai.

  Ainsi, la comparaison ci-dessus devient en fait une *expression de type entier*. Cela signifie qu'elle pourra éventuellement intervenir dans des calculs arithmétiques ;
- les expressions comparées pourront être d'un type de base quelconque et elles seront soumises aux règles de conversion présentées dans le paragraphe précédent. Cela signifie qu'au bout du compte on ne sera amené à comparer que des expressions de type numérique.

Voici la liste des opérateurs relationnels existant en C. Remarquez bien la notation (==) de l'opérateur d'égalité, le signe = étant, comme nous le verrons, réservé aux affectations. Notez également que = utilisé par mégarde à la place de == ne conduit généralement pas à un diagnostic de compilation, dans la mesure où l'expression ainsi obtenue possède un sens (mais qui n'est pas celui voulu).

Les opérateurs relationnels

OPÉRATEUR	SIGNIFICATION
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
!=	différent de

En ce qui concerne leurs priorités, il faut savoir que les quatre premiers opérateurs (<, <=, >, >=) sont de même priorité. Les deux derniers (== et !=) possèdent également la même priorité, mais celle-ci est inférieure à celle des précédents. Ainsi, l'expression :

```
a < b == c < d
```

est interprétée comme :

```
( a < b) == (c < d)
```

ce qui, en C, a effectivement une signification, étant donné que les expressions `a < b` et `c < d` sont, finalement, des quantités entières. En fait, cette expression prendra la valeur `1` lorsque les relations `a < b` et `c < d` auront toutes les deux la même valeur, c'est-à-dire soit lorsqu'elles seront toutes les deux vraies, soit lorsqu'elles seront toutes les deux fausses. Elle prendra la valeur `0` dans le cas contraire.

D'autre part, ces opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques. Cela permet souvent d'éviter certaines parenthèses dans des expressions.

Ainsi :

```
x + y < a + 2
```

est équivalent à :

```
( x + y ) < ( a + 2 )
```

Remarques

Important : *comparaisons de caractères.* Compte tenu des règles de conversion, une comparaison peut porter sur deux caractères. Bien entendu, la comparaison d'égalité ne pose pas de problème particulier ; par exemple (`c1` et `c2` étant de type `char`) :

`c1 == c2` sera vraie si `c1` et `c2` ont la même valeur, c'est-à-dire si `c1` et `c2` contiennent des caractères de même code, donc si `c1` et `c2` contiennent le même caractère,

`c1 == 'e'` sera vraie si le code de `c1` est égal au code de `'e'`, donc si `c1` contient le caractère `e`.

Autrement dit, dans ces circonstances, l'existence d'une conversion `char --> int` n'a guère d'influence. En revanche, pour les comparaisons d'inégalité, quelques précisions s'imposent. En effet, par exemple `c1 < c2` sera vraie si le code du caractère de `c1` a une valeur inférieure au code du caractère de `c2`. Le résultat d'une telle comparaison peut donc varier suivant le codage employé. Cependant, il faut savoir que, quel que soit ce codage :

l'ordre alphabétique est respecté pour les minuscules d'une part, pour les majuscules d'autre part ; on a toujours `'a' < 'c'`, `'C' < 'S'`...

- les chiffres sont classés par ordre naturel ; on a toujours `'2' < '5'`...

En revanche, aucune hypothèse ne peut être faite sur les places relatives des chiffres, des majuscules et des minuscules, pas plus que sur la place relative des caractères accentués (lorsqu'ils existent) par rapport aux autres caractères !

5 Les opérateurs logiques

C dispose de trois opérateurs logiques classiques : et (noté `&&`), ou (noté `||`) et non (noté `!`). Par exemple :

- `(a<b) && (c<d)`

  prend la valeur `1` (vrai) si les deux expressions `a<b` et `c<d` sont toutes deux vraies (de valeur `1`), et prend la valeur `0` (faux) dans le cas contraire.

- `(a<b) || (c<d)`

  prend la valeur `1` (vrai) si l'une au moins des deux conditions `a<b` et `c<d` est vraie (de valeur `1`), et prend la valeur `0` (faux) dans le cas contraire.

- `! (a<b)`

  prend la valeur `1` (vrai) si la condition `a<b` est fausse (de valeur `0`) et prend la valeur `0` (faux) dans le cas contraire. Cette expression est équivalente à : `a>=b`.

Il est important de constater que, ne disposant pas de type logique, C se contente de représenter vrai par `1` et faux par `0`. C'est pourquoi ces opérateurs produisent un résultat numérique (de type `int`).

De plus, on pourrait s'attendre à ce que les opérandes de ces opérateurs ne puissent être que des expressions prenant soit la valeur `0`, soit la valeur `1`. En fait, ces opérateurs acceptent n'importe quel opérande numérique, y compris les types flottants, avec les règles de conversion implicite déjà rencontrées. Leur signification reste celle évoquée ci-dessus, à condition de considérer que :

- `0` correspond à faux,
- toute valeur non nulle (et donc pas seulement la valeur `1`) correspond à vrai.

Le tableau suivant récapitule la situation.

Fonctionnement des opérateurs logiques en C

OPERANDE 1	OPERATEUR	OPERANDE 2	RESULTAT
0	&&	0	0
0	&&	non nul	0
non nul	&&	0	0
non nul	&&	non nul	1
0	||	0	0
0	||	non nul	1
non nul	||	0	1
non nul	||	non nul	1
	!	0	1
	!	non nul	0

Ainsi, en C, si `n` et `p` sont des entiers, des expressions telles que :

```
n && p          n || p          !n
```

sont acceptées par le compilateur. Notez que l'on rencontre fréquemment l'écriture :

```
if (!n)
```

plus concise (mais pas forcément plus lisible) que :

```
if ( n == 0 )
```

L'opérateur `!` a une priorité supérieure à celle de tous les opérateurs arithmétiques binaires et aux opérateurs relationnels. Ainsi, pour écrire la condition contraire de :

```
a == b
```

il est nécessaire d'utiliser des parenthèses en écrivant :

```
! ( a == b )
```

En effet, l'expression :

```
! a == b
```

serait interprétée comme :

```
( ! a ) == b
```

L'opérateur `||` est moins prioritaire que `&&`. Tous deux sont de priorité inférieure aux opérateurs arithmétiques ou relationnels. Ainsi, les expressions utilisées comme exemples en début de ce paragraphe auraient pu, en fait, être écrites sans parenthèses :

```
a<b && c<d        équivaut à      (a<b) && (c<d)
a<b || c<d        équivaut à      (a<b) || (c<d)
```

Enfin, les deux opérateurs `&&` et `||` jouissent en C d'une propriété intéressante : leur second opérande (celui qui figure à droite de l'opérateur) n'est évalué que si la connaissance de sa valeur est indispensable pour décider si l'expression correspondante est vraie ou fausse. Par exemple, dans une expression telle que :

```
a<b && c<d
```

on commence par évaluer `a<b`. Si le résultat est faux (`0`), il est inutile d'évaluer `c<d` puisque, de toute façon, l'expression complète aura la valeur faux (`0`).

La connaissance de cette propriété est indispensable pour maîtriser des « constructions » telles que :

```
if ( i<max && ( (c=getchar()) != '\n' ) )
```

En effet, le second opérande de l'opérateur `&&`, à savoir :

```
c = getchar() != '\n'
```

fait appel à la lecture d'un caractère au clavier. Celle-ci n'aura donc lieu que si la première condition (`i<max`) est vraie.

6 L'opérateur d'affectation ordinaire

Nous avons déjà eu l'occasion de remarquer que :

```
i = 5
```

était une expression qui :

- réalisait une action : l'affectation de la valeur `5` à `i`,
- possédait une valeur : celle de `i` après affectation, c'est-à-dire `5`.

Cet opérateur d'affectation (=) peut faire intervenir d'autres expressions comme dans :

```
c = b + 3
```

La faible priorité de cet opérateur = (elle est inférieure à celle de tous les opérateurs arithmétiques et de comparaison) fait qu'il y a d'abord évaluation de l'expression `b + 3`. La valeur ainsi obtenue est ensuite affectée à `c`.

En revanche, il n'est pas possible de faire apparaître une expression comme premier opérande de cet opérateur =. Ainsi, l'expression suivante n'aurait pas de sens :

```
c + 5 = x
```

6.1 Notion de lvalue

Nous voyons donc que cet opérateur d'affectation impose des restrictions sur son premier opérande. En effet, ce dernier doit être une référence à un emplacement mémoire dont on pourra effectivement modifier la valeur.

Dans les autres langages, on désigne souvent une telle référence par le nom de « variable » ; on précise généralement que ce terme recouvre par exemple les éléments de tableaux ou les composantes d'une structure. En langage C, cependant, la syntaxe du langage est telle que cette notion de variable n'est pas assez précise. Il faut introduire un mot nouveau : la lvalue. Ce terme désigne une « valeur à gauche », c'est-à-dire tout ce qui peut apparaître à gauche d'un opérateur d'affectation.

Certes, pour l'instant, vous pouvez trouver que dire qu'à gauche d'un opérateur d'affectation doit apparaître une *lvalue* n'apporte aucune information. En fait, d'une part, nous verrons qu'en C d'autres opérateurs que = font intervenir une *lvalue* ; d'autre part, au fur et à mesure que nous rencontrerons de nouveaux types d'objets, nous préciserons s'ils peuvent être ou non utilisés comme *lvalue*.

Pour l'instant, les seules *lvalue* que nous connaissons restent les variables de n'importe quel type de base déjà rencontré.

6.2 L'opérateur d'affectation possède une associativité de droite à gauche

Contrairement à tous ceux que nous avons rencontrés jusqu'ici, cet opérateur d'affectation possède une associativité de *droite à gauche*. C'est ce qui permet à une expression telle que :

```
i = j = 5
```

d'évaluer d'abord l'expression `j = 5` avant d'en affecter la valeur (`5`) à la variable `j`. Bien entendu, la valeur finale de cette expression est celle de i après affectation, c'est-à-dire `5`.

6.3 L'affectation peut entraîner une conversion

Là encore, la grande liberté offerte par le langage C en matière de mixage de types se traduit par la possibilité de fournir à cet opérateur d'affectation des opérandes de types différents.

Cette fois, cependant, contrairement à ce qui se produisait pour les opérateurs rencontrés précédemment et qui mettaient en jeu des conversions implicites, il n'est plus question, ici, d'effectuer une quelconque conversion de la *lvalue* qui apparaît à gauche de cet opérateur.

Une telle conversion reviendrait à changer le type de la *lvalue* figurant à gauche de cet opérateur, ce qui n'a pas de sens.

En fait, lorsque le type de l'expression figurant à droite n'est pas du même type que la *lvalue* figurant à gauche, il y a conversion systématique de la valeur de l'expression (qui est évaluée suivant les règles habituelles) dans le type de la *lvalue*. Une telle conversion imposée ne respecte plus nécessairement la hiérarchie des types qui est de rigueur dans le cas des conversions implicites. Elle peut donc conduire, suivant les cas, à une dégradation plus ou moins importante de l'information (par exemple lorsque l'on convertit un `double` en `int`, on perd la partie décimale du nombre).

Nous ferons le point sur ces différentes possibilités de conversions imposées par les affectations dans le paragraphe 9.

7 Les opérateurs d'incrémentation et de décrémentation

7.1 Leur rôle

Dans des programmes écrits dans un langage autre que C, on rencontre souvent des expressions (ou des instructions) telles que :

```
i = i + 1
n = n - 1
```

qui incrémentent ou qui décrémentent de `1` la valeur d'une variable (ou plus généralement d'une *lvalue*).

En C, ces actions peuvent être réalisées par des opérateurs « unaires » portant sur cette *lvalue*. Ainsi, l'expression :

```
++i
```

a pour effet d'incrémenter de `1` la valeur de `i`, et sa valeur est celle de `i` après incrémentation.

Là encore, comme pour l'affectation, nous avons affaire à une expression qui non seulement possède une valeur, mais qui, de surcroît, réalise une action (incrémentation de `i`).

Il est important de voir que la valeur de cette expression est celle de `i` après incrémentation. Ainsi, si la valeur de `i` est `5`, l'expression :

```
n = ++i - 5
```

affectera à `i` la valeur `6` et à `n` la valeur `1`.

En revanche, lorsque cet opérateur est placé *après* la *lvalue* sur laquelle il porte, la valeur de l'expression correspondante est celle de la variable avant incrémentation.

Ainsi, si `i` vaut `5`, l'expression :

```
n = i++ - 5
```

affectera à `i` la valeur `6` et à `n` la valeur `0` (car ici la valeur de l'expression `i++` est `5`).

On dit que `++` est :

- un opérateur de préincrémentation lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de postincrémentation lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

Bien entendu, lorsque seul importe l'effet d'incrémentation d'une *lvalue*, cet opérateur peut être indifféremment placé avant ou après. Ainsi, ces deux instructions (ici, il s'agit bien d'instructions car les expressions sont terminées par un point-virgule - leur valeur se trouve donc inutilisée) sont équivalentes :

```
i++ ;
++i ;
```

De la même manière, il existe un opérateur de décrémentation noté `--` qui, suivant les cas, sera :

- un opérateur de prédécrémentation lorsqu'il est placé à gauche de la *lvalue* sur laquelle il porte,
- un opérateur de postdécrémentation lorsqu'il est placé à droite de la *lvalue* sur laquelle il porte.

7.2 Leurs priorités

Les priorités élevées de ces opérateurs unaires (voir tableau en fin de chapitre) permettent d'écrire des expressions assez compliquées sans qu'il soit nécessaire d'employer des parenthèses pour isoler la *lvalue* sur laquelle ils portent. Ainsi, l'expression suivante a un sens :

```
3 * i++ * j-- + k++
```

(si * avait été plus prioritaire que la postincrémentation, ce dernier aurait été appliqué à l'expression `3*i` qui n'est pas une *lvalue* ; l'expression n'aurait alors pas eu de sens).

Remarque

Il est toujours possible (mais non obligatoire) de placer un ou plusieurs espaces entre un opérateur et les opérandes sur lesquels il porte. Nous utilisons souvent cette latitude pour accroître la lisibilité de nos instructions. Cependant, dans le cas des opérateurs d'incrémentation, nous avons plutôt tendance à ne pas le faire, cela pour mieux rapprocher l'opérateur de la *lvalue* sur laquelle il porte.

7.3 Leur intérêt

Ces opérateurs allègent l'écriture de certaines expressions et offrent surtout le grand avantage d'éviter la redondance qui est de mise dans la plupart des autres langages. En effet, dans une notation telle que :

```
i++
```

on ne cite qu'une seule fois la *lvalue* concernée alors qu'on est amené à le faire deux fois dans la notation :

```
i = i + 1
```

Les risques d'erreurs de programmation s'en trouvent ainsi quelque peu limités. Bien entendu, cet aspect prendra d'autant plus d'importance que la *lvalue* correspondante sera d'autant plus complexe.

D'une manière générale, nous utiliserons fréquemment ces opérateurs dans la manipulation de tableaux ou de chaînes de caractères. Ainsi, anticipant sur les chapitres suivants, nous pouvons indiquer qu'il sera possible de lire l'ensemble des valeurs d'un tableau nommé `t` en répétant la seule instruction :

```
t [i++] = getchar() ;
```

Celle-ci réalisera à la fois :

- la lecture d'un caractère au clavier,
- l'affectation de ce caractère à l'élément de rang `i` du tableau `t`,
- l'incrémentation de `1` de la valeur de `i` (qui sera ainsi préparée pour la lecture du prochain élément).

8 Les opérateurs d'affectation élargie

Nous venons de voir comment les opérateurs d'incrémentation permettaient de simplifier l'écriture de certaines affectations. Par exemple :

```
i++
```

remplaçait avantageusement :

```
i = i + 1
```

Mais C dispose d'opérateurs encore plus puissants. Ainsi, vous pourrez remplacer :

```
i = i + k
```

par :

```
i += k
```

ou, mieux encore :

```
a = a * b
```

par :

```
a *= b
```

D'une manière générale, C permet de condenser les affectations de la forme :

```
lvalue   =   lvalue   opérateur   expression
```

en :

```
lvalue   opérateur=   expression
```

Cette possibilité concerne tous les opérateurs binaires arithmétiques et de manipulation de bits. Voici la liste complète de tous ces nouveaux opérateurs nommés « opérateurs d'affectation élargie » :

```
+=   -=   *=   /=   %=   |=   ^=   &=   <<=   >>=
```

Remarque

Les cinq derniers correspondent en fait à des « opérateurs de manipulation de bits » (`|`, `^`, `&`, `<<` et `>>`) que nous n'aborderons que dans le chapitre 13.

Ces opérateurs, comme ceux d'incrémentation, permettent de condenser l'écriture de certaines instructions et contribuent à éviter la redondance introduite fréquemment par l'opérateur d'affectation classique.

Remarque

Ne confondez pas l'opérateur de comparaison `<=` avec un opérateur d'affectation élargie. Notez bien que les opérateurs de comparaison ne sont pas concernés par cette possibilité.

9 Les conversions forcées par une affectation

Nous avons déjà vu comment le compilateur peut être amené à introduire des conversions implicites dans l'évaluation des expressions. Dans ce cas, il applique les règles de promotions numériques et d'ajustement de type.

Par ailleurs, une affectation introduit une conversion d'office dans le type de la *lvalue* réceptrice, dès lors que cette dernière est d'un type différent de celui de l'expression correspondante. Par exemple, si `n` est de type `int` et `x` de type `float`, l'affectation :

```
n = x + 5.3 ;
```

entraînera tout d'abord l'évaluation de l'expression située à droite, ce qui fournira une valeur de type `float` ; cette dernière sera ensuite convertie en `int` pour pouvoir être affectée à `n`.

D'une manière générale, lors d'une affectation, toutes les conversions (d'un type numérique vers un autre type numérique) sont acceptées par le compilateur mais le résultat en est plus ou moins satisfaisant. En effet, si aucun problème ne se pose (autre qu'une éventuelle perte de précision) dans le cas de conversion ayant lieu suivant le bon sens de la hiérarchie des types, il n'en va plus de même dans les autres cas.

Par exemple, la conversion `float -> int` (telle que celle qui est mise en jeu dans l'instruction précédente) ne fournira un résultat acceptable que si la partie entière de la valeur flottante est représentable dans le type `int`. Si une telle condition n'est pas réalisée, non seulement le résultat obtenu pourra être différent d'un environnement à un autre mais, de surcroît, on pourra aboutir, dans certains cas, à une erreur d'exécution.

De la même manière, la conversion d'un `int` en `char` sera satisfaisante si la valeur de l'entier correspond à un code d'un caractère.

Sachez, toutefois, que les conversions d'un type entier vers un autre type entier ne conduisent, au pis, qu'à une valeur inattendue mais jamais à une erreur d'exécution.

10 L'opérateur de cast

S'il le souhaite, le programmeur peut forcer la conversion d'une expression quelconque dans un type de son choix, à l'aide d'un opérateur un peu particulier nommé en anglais « cast ».

Si, par exemple, `n` et `p` sont des variables entières, l'expression :

```
(double) ( n/p )
```

aura comme valeur celle de l'expression entière `n/p` convertie en `double`.

La notation `(double)` correspond en fait à un opérateur unaire dont le rôle est d'effectuer la conversion dans le type `double` de l'expression sur laquelle il porte. Notez bien que cet opérateur force la conversion du *résultat* de l'expression et non celle des différentes valeurs qui concourent à son évaluation. Autrement dit, ici, il y a d'abord calcul, dans le type `int`, du quotient de `n` par `p` ; c'est seulement ensuite que le résultat sera converti en `double`. Si `n` vaut `10` et que `p` vaut `3`, cette expression aura comme valeur `3`.

D'une manière générale, il existe autant d'opérateurs de « cast » que de types différents (y compris les types dérivés comme les pointeurs que nous rencontrerons ultérieurement). Leur priorité élevée (voir tableau en fin de chapitre) fait qu'il est généralement nécessaire de placer entre parenthèses l'expression concernée. Ainsi, l'expression :

```
(double) n/p
```

conduirait d'abord à convertir `n` en `double` ; les règles de conversions implicites amèneraient alors à convertir `p` en `double` avant qu'ait lieu la division (en `double`). Le résultat serait alors différent de celui obtenu par l'expression proposée en début de ce paragraphe (avec les mêmes valeurs de `n` et de `p`, on obtiendrait une valeur de l'ordre de `3.33333`...).

Bien entendu, comme pour les conversions forcées par une affectation, toutes les conversions numériques sont réalisables par un opérateur de « cast », mais le résultat en est plus ou moins satisfaisant (revoyez éventuellement le paragraphe précédent).

11 L'opérateur conditionnel

Considérons l'instruction suivante :

```
if ( a>b )
       max = a ;
   else
       max = b ;
```

Elle attribue à la variable `max` la plus grande des deux valeurs de `a` et de `b`. La valeur de `max` pourrait être définie par cette phrase :

```
Si a>b alors a sinon b
```

En langage C, il est possible, grâce à l'aide de l'opérateur conditionnel, de traduire presque littéralement la phrase ci-dessus de la manière suivante :

```
max = a>b ? a : b
```

L'expression figurant à droite de l'opérateur d'affectation est en fait constituée de trois expressions (`a>b`, `a` et `b`) qui sont les trois opérandes de l'opérateur conditionnel, lequel se matérialise par *deux symboles séparés :* `?` et `:`.

D'une manière générale, cet opérateur évalue la première expression qui joue le rôle d'une condition. Comme toujours en C, celle-ci peut être en fait de n'importe quel type. Si sa valeur est différente de zéro, il y a évaluation du second opérande, ce qui fournit le résultat ; si sa valeur est nulle, en revanche, il y a évaluation du troisième opérande, ce qui fournit le résultat.

Voici un autre exemple d'une expression calculant la valeur absolue de `3*a + 1` :

```
3*a+1 > 0  ?  3*a+1 :  -3*a-1
```

L'opérateur conditionnel dispose d'une faible priorité (il arrive juste avant l'affectation), de sorte qu'il est rarement nécessaire d'employer des parenthèses pour en délimiter les différents opérandes (bien que cela puisse parfois améliorer la lisibilité du programme). Voici, toutefois, un cas où les parenthèses sont indispensables :

```
z = (x=y) ? a : b
```

Le calcul de cette expression amène tout d'abord à affecter la valeur de `y` à `x`. Puis, si cette valeur est non nulle, on affecte la valeur de `a` à `z`. Si, au contraire, cette valeur est nulle, on affecte la valeur de `b` à `z`.

Il est clair que cette expression est différente de :

```
z = x = y ? a : b
```

laquelle serait évaluée comme :

```
z = x = ( y ? a : b )
```

Bien entendu, une expression conditionnelle peut, comme toute expression, apparaître à son tour dans une expression plus complexe. Voici, par exemple, une instruction (notez qu'il s'agit effectivement d'une instruction, car elle se termine par un point-virgule) affectant à `z` la plus grande des valeurs de `a` et de `b` :

```
z = ( a>b ? a : b ) ;
```

De même, rien n'empêche que l'expression conditionnelle soit évaluée sans que sa valeur soit utilisée comme dans cette instruction :

```
a>b ? i++ : i-- ;
```

Ici, suivant que la condition `a>b` est vraie ou fausse, on incrémentera ou on décrémentera la variable `i`.

12 L'opérateur séquentiel

Nous avons déjà vu qu'en C la notion d'expression était beaucoup plus générale que dans la plupart des autres langages. L'opérateur dit « séquentiel » va élargir encore un peu plus cette notion d'expression. En effet, celui-ci permet, en quelque sorte, d'exprimer *plusieurs calculs successifs au sein d'une même expression.* Par exemple :

```
a * b , i + j
```

est une expression qui évalue d'abord `a*b`, puis `i+j` et qui prend comme valeur la dernière calculée (donc ici celle de `i+j`). Certes, dans ce cas d'école, le calcul préalable de `a*b` est inutile puisqu'il n'intervient pas dans la valeur de l'expression globale et qu'il ne réalise aucune action.

En revanche, une expression telle que :

```
i++, a + b
```

peut présenter un intérêt puisque la première expression (dont la valeur ne sera pas utilisée) réalise en fait une incrémentation de la variable `i`.

Il en est de même de l'expression suivante :

```
i++, j = i + k
```

dans laquelle, il y a :

- évaluation de l'expression `i++`,
- évaluation de l'affectation `j = i + k`. Notez qu'alors on utilise la valeur de `i` après incrémentation par l'expression précédente.

Cet opérateur séquentiel, qui dispose d'une associativité de gauche à droite, peut facilement faire intervenir plusieurs expressions (sa faible priorité évite l'usage de parenthèses) :

```
i++, j = i+k, j--
```

Certes, un tel opérateur peut être utilisé pour réunir plusieurs instructions en une seule. Ainsi, par exemple, ces deux formulations sont équivalentes :

```
i++, j = i+k, j-- ;
i++ ; j = i+k ; j-- ;
```

Dans la pratique, ce n'est cependant pas là le principal usage que l'on fera de cet opérateur séquentiel. En revanche, ce dernier pourra fréquemment intervenir dans les instructions de choix ou dans les boucles ; là où celles-ci s'attendent à trouver une seule expression, l'opérateur séquentiel permettra d'en placer plusieurs, et donc d'y réaliser plusieurs calculs ou plusieurs actions. En voici deux exemples :

```
if (i++, k>0) ......
```

remplace :

```
i++ ; if (k>0) ......
```

et :

```
for (i=1, k=0 ; ... ; ... ) .......
```

remplace :

```
i=1 ; for (k=0 ; ... ; ... ) ......
```

Compte tenu de ce que l'appel d'une fonction n'est en fait rien d'autre qu'une expression, la construction suivante est parfaitement valide en C :

```
for (i=1, k=0, printf("on commence") ; ... ; ...) ......
```

Nous verrons même que, dans le cas des boucles conditionnelles, cet opérateur permet de réaliser des constructions ne possédant pas d'équivalent simple.

13 L'opérateur `sizeof`

L'opérateur `sizeof`, dont l'emploi ressemble à celui d'une fonction, fournit la taille en octets (n'oubliez pas que l'octet est, en fait, la plus petite partie adressable de la mémoire). Par exemple, dans une implémentation où le type `int` est représenté sur 2 octets et le type `double` sur 8 octets, si l'on suppose que l'on a affaire à ces déclarations :

```
int n ;
double z ;
```

- l'expression `sizeof(n)` vaudra `2`,
- l'expression `sizeof(z)` vaudra `8`.

Cet opérateur peut également s'appliquer à un type de nom donné. Ainsi, dans l'implémentation précédemment citée :

- `sizeof(int)` vaudra `2`,
- `sizeof(double)` vaudra `8`.

Quelle que soit l'implémentation, `sizeof(char)` vaudra toujours `1` (par définition, en quelque sorte).

Cet opérateur offre un intérêt :

- lorsque l'on souhaite écrire des programmes portables dans lesquels il est nécessaire de connaître la taille exacte de certains objets,
- pour éviter d'avoir à calculer soi-même la taille d'objets d'un type relativement complexe pour lequel on n'est pas certain de la manière dont il sera implémenté par le compilateur. Ce sera notamment le cas des structures.

14 Récapitulatif des priorités de tous les opérateurs

Le tableau ci-après fournit la liste complète des opérateurs du langage C, classés par ordre de priorité décroissante, accompagnés de leur mode d'associativité.

Les opérateurs du langage C et leurs priorités

CATEGORIE	OPERATEURS	ASSOCIATIVITE
référence	() [] -> .	--->
unaire	+ - ++ -- ! ~ * & (cast) sizeof	<---
arithmétique	* / %	--->
arithmétique	+ -	--->
décalage	<< >>	--->
relationnels	< <= > >=	--->
relationnels	== !=	--->
manip. de bits	&	--->
manip. de bits	^	--->
manip de bits	|	--->
logique	&&	--->
logique	||	--->
conditionnel	? :	--->
affectation	= += -= *= /= %= &= ^= |= <<= >>=	<---
séquentiel	,	--->

En langage C, un certain nombre de notations servant à référencer des objets sont considérées comme des opérateurs et, en tant que tels, soumises à des règles de priorité. Ce sont essentiellement :

- les références à des éléments d'un tableau réalisées par `[]`
- des références à des champs d'une structure : opérateurs `->` et `.`
- des opérateurs d'adressage : `*` et `&`

Ces opérateurs seront étudiés ultérieurement dans les chapitres correspondant aux tableaux, structures et pointeurs. Néanmoins, ils figurent dans le tableau proposé. De même, vous y trouverez les opérateurs de manipulation de bits dont nous ne parlerons que dans le chapitre 13.

Chapitre 4

Les entrées-sorties conversationnelles

Jusqu'ici, nous avons utilisé de façon intuitive les fonctions `printf` et `scanf` pour afficher des informations à l'écran ou pour en lire au clavier. Nous vous proposons maintenant d'étudier en détail les différentes possibilités de ces fonctions, ce qui nous permettra de répondre à des questions telles que :

- quelles sont les écritures autorisées pour des nombres fournis en données ? Que se passe-t-il lorsque l'utilisateur ne les respecte pas ?
- comment organiser les données lorsque l'on mélange les types numériques et les types caractères ?
- que se produit-il lorsque, en réponse à `scanf`, on fournit trop ou trop peu d'informations ?
- comment agir sur la présentation des informations à l'écran ?

Nous nous limiterons ici à ce que nous avons appelé les « entrées-sorties conversationnelles ». Plus tard, nous verrons que ces mêmes fonctions (moyennant la présence d'un argument supplémentaire) permettent également d'échanger des informations avec des fichiers.

En ce qui concerne la lecture au clavier, nous serons amené à mettre en évidence certaines lacunes de `scanf` en matière de comportement lors de réponses incorrectes et à vous fournir quelques idées sur la manière d'y remédier.

1 Les possibilités de la fonction `printf`

Nous avons déjà vu que le premier argument de `printf` est une chaîne de caractères qui spécifie à la fois :

- des caractères à afficher tels quels,
- des codes de format repérés par %. Un code de conversion (tel que `c`, `d` ou `f`) y précise le type de l'information à afficher.

D'une manière générale, il existe d'autres caractères de conversion soit pour d'autres types de valeurs, soit pour agir sur la précision de l'information que l'on affiche. De plus, un code de format peut contenir des informations complémentaires agissant sur le cadrage, le gabarit ou la précision. Ici, nous nous limiterons aux possibilités les plus usitées de `printf` . Nous avons toutefois mentionné le code de conversion relatif aux chaînes (qui ne seront abordées que dans le chapitre 8) et les entiers non signés (chapitre 13). Sachez cependant que le paragraphe 1.2 de l'annexe vous en fournit un panorama complet.

1.1 Les principaux codes de conversion

`c` `char` : caractère affiché « en clair » (convient aussi à `short` ou à `int` compte tenu des conversions systématiques)

`d` `int` (convient aussi à `char` ou à `int`, compte tenu des conversions systématiques)

`u` `unsigned int` (convient aussi à `unsigned char` ou à `unsigned short`, compte tenu des conversions systématiques)

`ld` `long`

`lu` `unsigned long`

`f` `double` ou `float` (compte tenu des conversions systématiques `float` -> `double`) écrit en notation décimale avec six chiffres après le point (par exemple : `1.234500` ou `123.456789`)

`e` `double` ou `float` (compte tenu des conversions systématiques `float` -> `double`) écrit en notation exponentielle (mantisse entre `1` inclus et `10` exclu) avec six chiffres après le point décimal, sous la forme `x.xxxxxxe+yyy` ou `x.xxxxxx-yyy` pour les nombres positifs et `-x.xxxxxxe+yyy` ou `-x.xxxxxxe-yyy` pour les nombres négatifs

`s` chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

1.2 Action sur le gabarit d'affichage

Par défaut, les entiers sont affichés avec le nombre de caractères nécessaires (sans espaces avant ou après). Les flottants sont affichés avec six chiffres après le point (aussi bien pour le code `e` que `f`).

Un nombre placé après % dans le code de format précise un gabarit d'affichage, c'est-à-dire un nombre minimal de caractères à utiliser. Si le nombre peut s'écrire avec moins de caractères, `printf` le fera précéder d'un nombre suffisant d'espaces ; en revanche, si le nombre ne peut s'afficher convenablement dans le gabarit imparti, `printf` utilisera le nombre de caractères nécessaires.

Voici quelques exemples, dans lesquels nous fournissons, à la suite d'une instruction `printf`, à la fois des valeurs possibles des expressions à afficher et le résultat obtenu à l'écran. Notez que le symbole ^ représente un espace.

```
printf ("%3d", n) ;      /* entier avec 3 caractères minimum */
     n = 20                   ^20
     n = 3                    ^^3
     n = 2358                 2358
     n = -5200                -5200

printf ("%f", x) ;       /* notation décimale gabarit par défaut  */
                         /*   (6 chiffres après point)            */
     x = 1.2345               1.234500
     x = 12.3456789           12.345679

printf ("%10f", x) ;     /* notation décimale - gabarit mini 10 */
                         /*   (toujours 6 chiffres après point) */
     x = 1.2345               ^^1.234500
     x = 12.345               ^12.345000
     x = 1.2345E5             123450.000000

printf ("%e", x) ; /* notation exponentielle - gabarit par défaut */
                     /*      (6 chiffres après point)               */
     x = 1.2345               1.234500e+000
     x = 123.45               1.234500e+002
     x = 123.456789E8         1.234568e+010
     x = -123.456789E8        -1.234568e+010
```

1.3 Actions sur la précision

Pour les types flottants, on peut spécifier un nombre de chiffres (éventuellement inférieur à 6) après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle). Ce nombre doit apparaître, précédé d'un point, avant le code de format (et éventuellement après le gabarit).

Voici quelques exemples :

```
printf ("%10.3f", x) ;  /* notation décimale, gabarit mini 10 */
                        /*    et 3 chiffres après point       */
     x = 1.2345              ^^^^^1.235
     x = 1.2345E3            ^^1234.500
     x = 1.2345E7            12345000.000

printf ("%12.4e", x) ;  /* notation exponentielle, gabarit mini 12*/
                        /*    et 4 chiffres après point          */
     x = 1.2345              ^1.2345e+000
     x = 123.456789E8        ^1.2346e+010
```

Remarques

Le signe moins (`-`), placé immédiatement après le symbole `%` (comme dans `%-4d` ou `%-10.3f`), demande de cadrer l'affichage à gauche au lieu de le cadrer (par défaut) à droite ; les éventuels espaces supplémentaires sont donc placés à droite et non plus à gauche de l'information affichée.

Le caractère * figurant à la place d'un gabarit ou d'une précision signifie que la valeur effective est fournie dans la liste des arguments de `printf`. En voici un exemple dans lequel nous appliquons ce mécanisme à la précision :

```
printf ("%8.*f", n, x) ;
n = 1    x = 1.2345              ^^^^1.2
n = 3    x = 1.2345              ^^^1.234
```

La fonction `printf` fournit en fait une valeur de retour. Il s'agit du nombre de caractères qu'elle a réellement affichés (ou la valeur `-1` en cas d'erreur). Par exemple, avec l'instruction suivante, on s'assure que l'opération d'affichage s'est bien déroulée :

```
if (printf ("....", ...) != -1 ) .....
```

De même, on obtient le nombre de caractères effectivement affichés par :

```
n = printf ("....", ....)
```

1.4 La syntaxe de `printf`

D'une manière générale, nous pouvons dire que l'appel à `printf` se présente ainsi :

La fonction printf

```
printf  ( format,  liste_d'expressions )
```

- `format` :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- `liste_d'expressions` : suite d'expressions séparées par des virgules d'un type en accord avec le code format correspondant.

Remarque Nous verrons que les deux notions de constante chaîne et de pointeur sur une chaîne sont identiques.

1.5 En cas d'erreur de programmation

Deux types d'erreur de programmation peuvent apparaître dans l'emploi de `printf`.

a) code de format en désaccord avec le type de l'expression

Lorsque le code de format, bien qu'erroné, correspond à une information de même taille (c'est-à-dire occupant la même place en mémoire) que celle relative au type de l'expression, les conséquences de l'erreur se limitent à une *mauvaise interprétation de l'expression*. C'est ce qui se passe, par exemple, lorsque l'on écrit une valeur de type `int` en `%u` ou une valeur de type `unsigned int` en `%d`.

En revanche, lorsque le code format correspond à une information de taille différente de celle relative au type de l'expression, les conséquences sont généralement plus désastreuses, du moins si d'autres valeurs doivent être affichées à la suite. En effet, tout se passe alors comme si, dans la suite d'octets (correspondant aux différentes valeurs à afficher) reçue par `printf`, le repérage des emplacements des valeurs suivantes se trouvait soumis à un décalage.

b) nombre de codes de format différent du nombre d'expressions de la liste

Dans ce cas, il faut savoir que C cherche toujours à satisfaire le contenu du format.

Ce qui signifie que, si des expressions de la liste n'ont pas de code format, elles ne seront pas affichées. C'est le cas dans cette instruction où la valeur de `p` ne sera pas affichée :

```
printf ("%d", n, p) ;
```

En revanche, si vous prévoyez trop de codes de format, les conséquences seront là encore assez désastreuses puisque `printf` cherchera à afficher n'importe quoi. C'est le cas dans cette instruction où deux valeurs seront affichées, la seconde étant (relativement) aléatoire :

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

1.6 La macro putchar

L'expression :

```
putchar (c)
```

joue le même rôle que :

```
printf ("%c", c)
```

Son exécution est toutefois plus rapide, dans la mesure où elle ne fait pas appel au mécanisme d'analyse de format. Notez qu'en toute rigueur putchar n'est pas une vraie fonction mais une macro. Ses instructions (écrites en C) seront incorporées à votre programme par la directive :

```
#include <stdio.h>
```

Alors que cette directive était facultative pour printf (qui est une fonction), elle devient absolument nécessaire pour putchar. En son absence, l'éditeur de liens serait amené à rechercher une fonction putchar en bibliothèque et, ne la trouvant pas, il vous gratifierait d'un message d'erreur En toute rigueur, la fonction recherchée pourra porter un nom légèrement différent, par exemple _putchar ; c'est ce nom qui figurera dans le message d'erreur fourni par l'éditeur de liens.

2 Les possibilités de la fonction scanf

Nous avons déjà rencontré quelques exemples d'appels de scanf. Nous y avons notamment vu la nécessité de recourir à l'opérateur & pour désigner l'adresse de la variable (plus généralement de la *lvalue*) pour laquelle on souhaite lire une valeur. Vous avez pu remarquer que cette fonction possédait une certaine ressemblance avec printf et qu'en particulier elle faisait, elle aussi, appel à des « codes de format ».

Cependant, ces ressemblances masquent également des différences assez importantes au niveau :

- de la signification des codes de format. Certains codes correspondront à des types différents, suivant qu'ils sont employés avec printf ou avec scanf ;
- de l'interprétation des caractères du format qui ne font pas partie d'un code de format.

Ici, nous allons vous montrer le fonctionnement de scanf. Comme nous l'avons fait pour printf, nous vous présenterons d'abord les principaux codes de conversion. (Le paragraphe 1.2 de l'annexe vous en fournira un panorama complet). Là encore, nous avons également mentionné les codes de conversion relatifs aux chaînes et aux entiers non signés.

En revanche, compte tenu de la complexité de scanf, nous vous en exposerons les différentes possibilités de façon progressive, à l'aide d'exemples. Notamment, ce n'est qu'à la fin de ce chapitre que vous serez en mesure de connaître toutes les conséquences de données incorrectes.

2.1 Les principaux codes de conversion de `scanf`

Pour chaque code de conversion, nous précisons le type de la *lvalue* correspondante.

`c`	`char`
`d`	`int`
`u`	`unsigned int`
`hd`	`short int`
`hu`	`unsigned short`
`ld`	`long int`
`lu`	`unsigned long`
`f` ou `e`	`loat` écrit indifféremment dans l'une des deux notations : décimale (éventuel- lement sans point, c'est-à-dire comme un entier) ou exponentielle (avec la lettre `e` ou `E`)
`lf` ou `le`	`double` avec la même présentation que ci-dessus
`s`	chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

Remarque

Contrairement à ce qui se passait pour `printf`, il ne peut plus y avoir ici de conversion automatique puisque l'argument transmis à `scanf` est l'adresse d'un emplacement mémoire. C'est ce qui justifie l'existence d'un code `hd` par exemple pour le type `short` ou encore celle des codes `lf` et *le* pour le type `double`.

2.2 Premières notions de tampon et de séparateurs

Lorsque `scanf` attend que vous lui fournissiez des données, l'information frappée au clavier est rangée temporairement dans l'emplacement mémoire nommé « tampon ». Ce dernier est exploré, caractère par caractère par `scanf`, au fur et à mesure des besoins. Il existe un pointeur qui précise quel est le prochain caractère à prendre en compte.

D'autre part, certains caractères dits « séparateurs » (ou « espaces blancs ») jouent un rôle particulier dans les données. Les deux principaux sont l'espace et la fin de ligne (`\n`). Il en existe trois autres d'un usage beaucoup moins fréquent : la tabulation horizontale (`\t`), la tabulation verticale (`\v`) et le changement de page (`\f`).

2.3 Les premières règles utilisées par `scanf`

Les codes de format correspondant à un nombre (c'est-à-dire tous ceux de la liste précédente, excepté `%c` et `%s`) entraînent d'abord l'avancement éventuel du pointeur jusqu'au premier caractère différent d'un séparateur. Puis `scanf` prend en compte tous les caractères suivants jusqu'à la rencontre d'un séparateur (en y plaçant le pointeur), du moins lorsque aucun gabarit

n'est précisé (comme nous apprendrons à le faire dans le paragraphe 2.4) et qu'aucun caractère invalide n'est présent dans la donnée (nous y reviendrons au paragraphe 2.6).

Quant au code de format `%c`, il entraîne la prise en compte du caractère désigné par le pointeur (même s'il s'agit d'un séparateur comme espace ou fin de ligne), et le pointeur est simplement avancé sur le caractère suivant du tampon.

Voici quelques exemples dans lesquels nous supposons que `n` et `p` sont de type `int`, tandis que `c` est de type `char`. Nous fournissons, pour chaque appel de `scanf`, des exemples de réponses possibles (`^` désigne un espace et `@` une fin de ligne) et, en regard, les valeurs effectivement lues.

```
scanf ("%d%d", &n, &p) ;
      12^25@              n = 12      p = 25
      ^12^^25^^@          n = 12      p = 25

      12@
      @
      ^25@                n = 12      p = 25

scanf ("%c%d", &c, &n) ;
      a25@                c = 'a'     n = 25
      a^^25@              c = 'a'     n = 25

scanf ("%d%c", &n, &c) ;
      12 a@               n = 12      c = ' '
```

Notez que, dans ce cas, on obtient bien le caractère « espace » dans `c`. Nous verrons dans le paragraphe 2.5 comment imposer à `scanf` de sauter quand même les espaces dans ce cas.

Remarque

Le code de format précise la nature du travail à effectuer pour transcoder une partie de l'information frappée au clavier, laquelle n'est en fait qu'une suite de caractères (codés chacun sur un octet) pour fabriquer la valeur (binaire) de la variable correspondante. Par exemple, `%d` entraîne en quelque sorte une double conversion : suite de caractères -> nombre écrit en décimal -> nombre codé en binaire ; la première conversion revient à faire correspondre un nombre entre 0 et 9 à un caractère représentant un chiffre. En revanche, le code `%c` demande simplement de ne rien faire puisqu'il suffit de recopier tel quel l'octet contenant le caractère concerné.

2.4 Imposition d'un gabarit maximal

Comme dans les codes de format de `printf`, on peut, dans un code de format de `scanf`, préciser un gabarit. Dans ce cas, le traitement d'un code de format s'interrompt soit à la rencontre d'un séparateur, soit lorsque le nombre de caractères indiqués a été atteint (attention, les séparateurs éventuellement sautés auparavant ne sont pas comptabilisés !).

Voici un exemple :

```
scanf ("%3d%3d", &n, &p)
     12^25@                    n = 12   p = 25

     ^^^^^12345@               n = 123  p = 45

     12@
     25@                       n = 12   p = 25
```

2.5 Rôle d'un espace dans le format

Un espace entre deux codes de format demande à `scanf` de faire avancer le pointeur au prochain caractère différent d'un séparateur. Notez que c'est déjà ce qui se passe lorsque l'on a affaire à un code de format correspondant à un type numérique. En revanche, cela n'était pas le cas pour les caractères, comme nous l'avons vu au paragraphe 2.3.

Voici un exemple :

```
scanf ("%d^%c", &n, &c) ;   /* ^ désigne un espace          */
                            /* %d^%c est différent de %d%c */
     12^a@               n = 12   c = 'a'
     12^^^a@             n = 12   c = 'a'
     12@a@               n = 12   c = 'a'
```

2.6 Cas où un caractère invalide apparaît dans une donnée

Voyez cet exemple, accompagné des valeurs obtenues dans les variables concernées :

```
scanf ("%d^%c", &n, &c) ;    /* ^ désigne un espace */
     12a@               n = 12   c = 'a'
```

Ce cas fait intervenir un mécanisme que nous n'avons pas encore rencontré. Il s'agit d'un troisième critère d'arrêt du traitement d'un code format (les deux premiers étaient : rencontre d'un séparateur ou gabarit atteint).

Ici, lors du traitement du code `%d`, `scanf` rencontre les caractères `1`, puis `2`, puis `a`. Ce caractère `a` ne convenant pas à la fabrication d'une valeur entière, `scanf` interrompt son exploration et fournit donc la valeur `12` pour `n`. L'espace qui suit `%d` dans le format n'a aucun effet puisque le caractère courant est le caractère `a` (différent d'un séparateur). Le traitement du code suivant, c'est-à-dire `%c`, amène `scanf` à prendre ce caractère courant (`a`) et à l'affecter à la variable `c`.

D'une manière générale, dans le traitement d'un code de format, `scanf` arrête son exploration du tampon dès que l'une des trois conditions est satisfaite :

- rencontre d'un caractère séparateur,

- gabarit maximal atteint (s'il y en a un de spécifié),
- rencontre d'un caractère invalide, par rapport à l'usage qu'on veut en faire (par exemple un point pour un entier, une lettre autre que `E` ou `e` pour un flottant,...). Notez bien l'aspect relatif de cette notion de caractère invalide.

2.7 Arrêt prématuré de `scanf`

Voyez cet exemple, dans lequel nous utilisons, pour la première fois, la valeur de retour de la fonction `scanf`.

```
compte = scanf ("%d^%d^%c", &n, &p, &c) ;    /* ^ désigne un espace */
```

```
12^25^b@     n = 12         p = 25         c = 'b'         compte = 3
12b@         n = 12         p inchangé     c inchangé      compte = 1
b@           n indéfini     p inchangé     c inchangé      compte = 0
```

La valeur fournie par `scanf` n'est pas comparable à celle fournie par `printf` puisqu'il s'agit cette fois du nombre de valeurs convenablement lues. Ainsi, dans le premier cas, il n'est pas surprenant de constater que cette valeur est égale à `3`.

En revanche, dans le deuxième cas, le caractère `b` a interrompu le traitement du premier code `%d`. Dans le traitement du deuxième code (`%d`), `scanf` a rencontré d'emblée ce caractère `b`, toujours invalide pour une valeur numérique. Dans ces conditions, `scanf` se trouve dans l'incapacité d'attribuer une valeur à `p` (puisque ici, contrairement à ce qui s'est passé pour `n`, elle ne dispose d'aucun caractère correct). Dans un tel cas, `scanf` s'interrompt sans chercher à lire d'autres valeurs et fournit, en retour, le nombre de valeurs correctement lues jusqu'ici, c'est-à-dire `1`. Les valeurs de `p` et de `c` restent inchangées (éventuellement indéfinies).

Dans le troisième cas, le même mécanisme d'arrêt prématuré se produit dès le traitement du premier code de format, et le nombre de valeurs correctement lues est `0`.

Ne confondez pas cet arrêt prématuré de *scanf* avec le troisième critère d'arrêt de traite- ment d'un code de format. En effet, les deux situations possèdent bien la même cause (un caractère invalide par rapport à l'usage que l'on souhaite en faire), mais seul le cas où `scanf` n'est pas en mesure de fabriquer une valeur conduit à l'arrêt prématuré.

Remarque

Ici, nous avons vu la signification d'un espace introduit entre deux codes de format. En toute rigueur, vous pouvez introduire à un tel endroit n'importe quel caractère de votre choix. Dans ce cas, sachez que lorsque `scanf` rencontre un caractère (`x` par exemple) dans le format, il le compare avec le caractère courant (celui désigné par le pointeur) du tampon. S'ils sont égaux, il poursuit son travail (après avoir avancé le pointeur) mais, dans le cas contraire, il y a arrêt prématuré. Une telle possibilité ne doit toutefois être réservée qu'à des cas bien particuliers.

2.8 La syntaxe de `scanf`

D'une manière générale, l'appel de `scanf` se présente ainsi :

La fonction `scanf`

```
scanf (format, liste_d_adresses)
```

- `format` :
  - constante chaîne (entre " "),
  - pointeur sur une chaîne de caractères (cette notion sera étudiée ultérieurement).
- `liste_d_adresses` : liste de *lvalue*, séparées par des virgules, d'un type en accord avec le code de format correspondant.

2.9 Problèmes de synchronisation entre l'écran et le clavier

Voyez cet exemple de programme accompagné de son exécution alors que nous avons répondu :

```
12^25@
```

à la première question posée.

L'écran et le clavier semblent mal synchronisés

```
#include <stdio.h>
main()
{
    int n, p ;
    printf ("donnez une valeur pour n : ") ;
    scanf ("%d", &n) ;
    printf ("merci pour %d\n", n) ;
    printf ("donnez une valeur pour p : ") ;
    scanf ("%d", &p) ;
    printf ("merci pour %d", p) ;
}
```

```
donnez une valeur pour n : 12 25
merci pour 12
donnez une valeur pour p : merci pour 25
```

Vous constatez que la seconde question (donnez une valeur pour p) est apparue à l'écran, mais le programme n'a pas attendu que vous frappiez votre réponse pour vous afficher la suite. Vous notez alors qu'il a bien pris pour p la seconde valeur entrée au préalable, à savoir 25.

En fait, comme nous l'avons vu, les informations frappées au clavier ne sont pas traitées instantanément par scanf mais mémorisées dans un tampon. Jusqu'ici, cependant, nous n'avions pas précisé quand scanf s'arrêtait de mémoriser pour commencer à traiter. Il le fait tout naturellement à la rencontre d'un caractère de fin de ligne généré par la frappe de la touche « return », dont le rôle est aussi classiquement celui d'une validation. Notez que, bien qu'il joue le rôle d'une validation, ce caractère de fin de ligne est quand même recopié dans le tampon ; il pourra donc éventuellement être lu en tant que tel.

L'élément nouveau réside donc dans le fait que scanf reçoit une information découpée en lignes (nous appelons ainsi une suite de caractères terminée par une fin de ligne). Tant que son traitement n'est pas terminé, elle attend une nouvelle ligne (c'est d'ailleurs ce qui se produisait dans notre premier exemple dans lequel nous commencions par frapper « return »).

Par contre, lorsque son traitement est terminé, s'il existe une partie de ligne non encore utilisée, celle-ci est conservée pour une prochaine lecture.

Autrement dit, le tampon n'est pas vidé à chaque nouvel appel de scanf. C'est ce qui explique le comportement du programme précédent.

2.10 En cas d'erreur

Dans le cas de printf, la source unique d'erreur résidait dans les fautes de programmation. Dans le cas de scanf, en revanche, il peut s'agir, non seulement d'une faute de programmation, mais également d'une mauvaise réponse de l'utilisateur.

2.10.1 Erreurs de programmation

Comme dans le cas de printf, ces erreurs peuvent être de deux types :

a) Code de format en désaccord avec le type de l'expression

Si le code de format, bien qu'erroné, correspond à un type de longueur égale à celle de la *lvalue* mentionnée dans la liste, les conséquences se limitent, là encore, à l'introduction d'une mauvaise valeur. Si, en revanche, la *lvalue* a une taille inférieure à celle correspondant au type mentionné dans le code format, il y aura écrasement d'un emplacement mémoire consécutif à cette *lvalue*. Les conséquences en sont difficilement prévisibles.

b) Nombre de codes de format différent du nombre d'éléments de la liste

Comme dans le cas de printf, il faut savoir que scanf cherche toujours à satisfaire le contenu du format. Les conséquences sont limitées dans le cas où le format comporte moins de codes que la liste ; ainsi, dans cette instruction, on ne cherchera à lire que la valeur de n :

```
scanf ("%d", &n, &p) ;
```

En revanche, dans le cas où le format comporte plus de codes que la liste, on cherchera à affecter des valeurs à des emplacements (presque) aléatoires de la mémoire. Là encore, les conséquences en seront pratiquement imprévisibles.

2.10.2 Mauvaise réponse de l'utilisateur

Nous avons déjà vu ce qui se passait lorsque l'utilisateur fournissait trop ou trop peu d'information par rapport à ce qu'attendait `scanf`.

De même, nous avons vu comment, en cas de rencontre d'un caractère invalide, il y avait arrêt prématuré. Dans ce cas, il faut bien voir que ce caractère non exploité reste dans le tampon pour une prochaine fois. Cela peut conduire à des situations assez cocasses telles que celle qui est présentée dans cet exemple (l'impression de `^C` représente, dans l'environnement utilisé, une interruption du programme par l'utilisateur) :

Boucle infinie sur un caractère invalide

```
main()
{
   int n ;
   do
      { printf ("donnez un nombre : ") ;
        scanf ("%d", &n) ;
        printf ("voici son carré : %d\n", n*n) ;
      }
   while (n) ;
}
```

```
donnez un nombre : 12
voici son carré : 144
donnez un nombre : &
voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
donnez un nombre : voici son carré : 144
 ^C
```

Fort heureusement, il existe un remède à cette situation. Nous ne pourrons vous l'exposer complètement que lorsque nous aurons étudié les chaînes de caractères.

2.11 La macro `getchar`

L'expression :

```
c = getchar()
```

joue le même rôle que :

```
scanf ("%c", &c)
```

tout en étant plus rapide puisque ne faisant pas appel au mécanisme d'analyse d'un format.

Notez bien que `getchar` utilise le même tampon (image d'une ligne) que `scanf`.

En toute rigueur, `getchar` est une macro (comme `putchar`) dont les instructions figurent dans `stdio.h`. Là encore, l'omission d'une instruction `#include` appropriée conduit à une erreur à l'édition de liens.